Документация к Postgres Pro Standard 17.6.1

Общество с ограниченной ответственностью «Постгрес Профессиональный»
https://postgrespro.ru

Документация к Postgres Pro Standard 17.6.1

Общество с ограниченной ответственностью «Постгрес Профессиональный»

Авторские права © 2016–2025 Постгрес Профессиональный

Юридическое уведомление

Представленная документация предназначена исключительно для использования с программным обеспечением «СУБД Postgres Pro» и для пользователей данной СУБД.

Не допускается использование документации для сторонних продуктов или в составе документации к иным продуктам.

Иные условия использования данной документации приведены в Пользовательском соглашении.

Postgres Pro © 2016–2025 — Общество с ограниченной ответственностью «Постгрес Профессиональный».

Общество с ограниченной ответственностью «Постгрес Профессиональный» (Postgres Professional) не несёт никакой ответственности за любые повреждения, включая потерю дохода, нанесённые прямым или непрямым, специальным или случайным использованием программного обеспечения «СУБД Postgres Pro» во всех версиях и/или его документации, даже если общество с ограниченной ответственностью «Постгрес Профессиональный» было извещено о возможности таких повреждений.

Общество с ограниченной ответственностью «Постгрес Профессиональный» (Postgres Professional) специально отказывается предоставлять любые гарантии, включая, но не ограничиваясь только этими гарантиями: неявные гарантии пригодности товара или пригодности для отдельной цели. Программное обеспечение «СУБД Postgres Pro» во всех версиях и/или документация к нему предоставляется на основе принципа «как есть», и общество с ограниченной ответственностью «Постгрес Профессиональный» не обязано предоставлять сопровождение, поддержку, обновления, расширения или изменения.

Для распространения этой документации или её частей требуется явное письменное разрешение от Общества с ограниченной ответственностью «Постгрес Профессиональный».

Документация программного обеспечения «СУБД Postgres Pro» во всех версиях основана на документации PostgreSQL, которая распространяется по следующей лицензии:

Система управления базами данных PostgreSQL (также известная как Postgres, ранее известная как Postgres95)

Частичные авторские права © 1996-2025 — PostgreSQL Global Development Group.

Частичные авторские права © 1994 — Регенты университета Калифорнии.

Настоящим разрешается использование, копирование, модификация и распространение данного программного обеспечения и документации для любых целей, бесплатно и без письменного разрешения, при условии сохранения во всех копиях приведённого выше уведомления об авторских правах и данного параграфа вместе с двумя последующими параграфами.

УНИВЕРСИТЕТ КАЛИФОРНИИ НИ В КОЕЙ МЕРЕ НЕ НЕСЁТ ОТВЕТСТВЕННОСТИ ЗА ПРЯМОЙ, КОСВЕННЫЙ, НАМЕРЕННЫЙ, СЛУЧАЙНЫЙ ИЛИ СПРОВОЦИРОВАННЫЙ УЩЕРБ, В ТОМ ЧИСЛЕ ПОТЕРЯННЫЕ ПРИБЫЛИ, СВЯЗАННЫЙ С ИСПОЛЬЗОВАНИЕМ ЭТОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ЕГО ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ УНИВЕРСИТЕТ КАЛИФОРНИИ БЫЛ УВЕДОМЛЁН О ВОЗМОЖНОСТИ ТАКОГО УЩЕРБА.

УНИВЕРСИТЕТ КАЛИФОРНИИ ЯВНЫМ ОБРАЗОМ ОТКАЗЫВАЕТСЯ ОТ ЛЮБЫХ ГАРАНТИЙ, В ЧАСТНОСТИ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ КОММЕРЧЕСКОЙ ВЫГОДЫ ИЛИ ПРИГОДНОСТИ ДЛЯ КАКОЙ-ЛИБО ЦЕЛИ. НАСТОЯЩЕЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПРЕДОСТАВЛЯЕТСЯ В ВИДЕ «КАК ЕСТЬ», И УНИВЕРСИТЕТ КАЛИФОРНИИ НЕ ДАЁТ НИКАКИХ ОБЯЗАТЕЛЬСТВ ПО ЕГО ОБСЛУЖИВАНИЮ, ПОДДЕРЖКЕ, ОБНОВЛЕНИЮ, УЛУЧШЕНИЮ ИЛИ МОДИФИКАЦИИ.

Предисловие

Эта книга является официальной документацией Postgres Pro Standard. Она была написана разработчиками Postgres Pro, сообществом PostgreSQL и другими добровольцами параллельно с разработкой программного продукта PostgreSQL и Postgres Pro. В ней описывается вся функциональность, которую официально поддерживает текущая версия Postgres Pro.

Чтобы такой большой объём информации о Postgres Pro был удобоварим, эта книга разделена на части. Каждая часть предназначена определённой категории пользователей или пользователям на разных стадиях освоения Postgres Pro:

- Часть I представляет собой неформальное введение для начинающих.
- Часть II описывает язык SQL и его среду, включая типы данных и функции, а также оптимизацию производительности на уровне пользователей. Это будет полезно прочитать всем пользователям Postgres Pro.
- Часть III описывает установку и администрирование сервера. Эта часть будет полезна всем, кто использует сервер Postgres Pro для самых разных целей.
- Часть IV описывает программные интерфейсы для клиентских приложений Postgres Pro.
- Часть V предназначена для более опытных пользователей и содержит сведения о возможностях расширения сервера. Эта часть включает темы, посвящённые, в частности, пользовательским типам данных и функциям.
- Часть VI содержит справочную информацию о командах SQL, а также клиентских и серверных программах. Эта часть дополняет другие информацией, структурированной по командам и программам.
- Часть VII содержит разнообразную информацию, полезную для разработчиков Postgres Pro.

1. Что такое Postgres Pro Standard?

Postgres Pro Standard — это объектно-реляционная система управления базами данных (ОРСУБД, ORDBMS), разработанная Postgres Professional в рамках проекта Postgres Pro на основе *PostgreSQL*, в свою очередь, основанном на *POSTGRES, Version 4.2* — программе, разработанной на факультете компьютерных наук Калифорнийского университета в Беркли. В POSTGRES появилось множество новшеств, которые были реализованы в некоторых коммерческих СУБД гораздо позднее.

Postgres Pro Standard работает во всех основных операционных системах Linux. Как и PostgreSQL, Postgres Pro Standard соответствует требованиям *ACID*.

Postgres Pro Standard, как и PostgreSQL поддерживает большую часть стандарта SQL и предлагает множество современных функций:

- сложные запросы
- внешние ключи
- триггеры
- изменяемые представления
- транзакционная целостность
- многоверсионное управление конкурентным доступом

Кроме того, пользователи могут всячески расширять Postgres Pro, так же как и PostgreSQL, например, создавая свои

- типы данных
- функции
- операторы
- агрегатные функции
- методы индексирования

- процедурные языки

2. Различия между Postgres Pro Standard и PostgreSQL

Postgres Pro предоставляет наиболее актуальную версию PostgreSQL с дополнительными изменениями и расширениями. Этот продукт включает все новые возможности, реализованные компанией Postgres Professional, а также сторонние доработки, которые уже приняты сообществом PostgreSQL и попадут в новые версии PostgreSQL. Таким образом, пользователи Postgres Pro Standard получают ранний доступ к важным нововведениям и исправлениям.

Примечание

Postgres Pro Standard предоставляется по следующей лицензии: *https://postgrespro.ru/products/postgrespro/eula*. Обязательно ознакомьтесь с условиями лицензии, прежде чем загружать и использовать Postgres Pro Standard.

Postgres Pro Standard отличают от PostgreSQL следующие усовершенствования:

- Улучшенный механизм проверки блокировок, не оказывающий отрицательного влияния на производительность.
- Увеличенная скорость и эффективность планирования для различных типов запросов.
- Уменьшенное потребление памяти при обработке сложных запросов со множеством таблиц.
- Добавление времени планирования в информацию, выводимую модулем auto_explain.
- Возможность замены нулевого байта заданным ASCII-символом при загрузке данных с помощью команды `COPY FROM`. (См. описание параметра nul_byte_replacement_on_import.)

  Возможность замены символа `'\u0000'` заданным символом Unicode при вызове функции, обрабатывающей JSONB. (См. описание параметра unicode_nul_character_replacement_in_jsonb.)
- Реализация механизма PTRACK, позволяющего программе pg_probackup отслеживать изменения страниц при создании инкрементальных резервных копий.
- Поддержка чтения файлов `pg_control` предыдущих версий PostgreSQL/Postgres Pro утилитой pgpro_controldata.
- Проверка совместимости кластера, позволяющая определить, совместима ли текущая версия Postgres Pro с определённым кластером, и понять, какие параметры влияют на совместимость, не запуская сервер. (См. описание параметра postgres `-Z` и описание pgpro_controldata.)
- Возможность изменения параметра `restore_command` без перезапуска сервера.
- Унифицированная структура пакетов двоичных файлов для всех дистрибутивов Linux, упрощающая миграцию между ними и позволяющая устанавливать продукты на базе PostgreSQL совместно без каких-либо конфликтов. (См. Главу 16.)
- Журнал операций, в котором хранится информация о критически важных системных событиях, таких как обновление, выполнение pg_resetwal и тому подобное. Эта информация важна для осуществления технической поддержки со стороны поставщика. Запись в журнал операций производится только на системном уровне, а для его чтения используются функции SQL. (См. Подраздел 9.28.11.)
- Расширенные политики аутентификации, обеспечивающие эффективное управление паролями и контроль доступа. (См. CREATE PROFILE и ALTER ROLE).
- Встроенные механизмы защиты данных, которые позволяют стерилизовать объекты, перед удалением заполняя их нулями. Обнуление объектов может производиться перед удалением файлов на диске и перед удалением устаревших версий строк (очисткой страниц), освобождением ОЗУ и удалением или перезаписью файлов WAL. (Только сертифицированная редакция.)
- Статистика очистки таблиц, индексов и баз данных в системных представлениях.
- Предопределённые роли, позволяющие создавать табличные пространства и управлять профилями без прав суперпользователя.
- Сбор информации о сбоях сервера, который включается параметром конфигурации crash_info и находится под управлением нескольких параметров, включая этот.

- Оптимизированное потребление памяти при оценке избирательности для каждого элемента массива.
- Улучшенное управление секционированными таблицами с использованием подкоманд `SPLIT PARTITION` и `MERGE PARTITIONS` команды `ALTER TABLE`: первая разделяет одну секцию на несколько, вторая объединяет несколько секций в одну.
- Экспериментальная функциональность, которая включает механизм параллельного выполнения запросов для запросов, использующих хотя бы одну временную таблицу. Эта функциональность полезна для 1С. (См. описание параметра enable_parallel_temptables.)
- Оценка приблизительной стоимости сброса страниц временных таблиц на диск. Эта функциональность полезна для 1С. (См. описание параметра write_page_cost.)
- Возможность преобразования подзапросов `ANY` в соединения `LATERAL`. (См. описание параметра enable_any_to_lateral_transformation.)

Postgres Pro Standard также включает следующие дополнительные модули и приложения:

- Расширение aqo для адаптивной оптимизации запросов.
- Расширение daterange_inclusive, позволяющее добавлять в вывод верхнюю границу временного диапазона.
- Модуль dump_stat, позволяющий сохранять статистику данных при резервном копировании и восстановлении.
- Модуль fasttrun, который предоставляет транзакционно-небезопасную функцию для усечения временных таблиц, что предотвращает разрастание каталога `pg_class`.
- Модуль fulleq, предоставляющий дополнительный оператор равенства для совместимости с Microsoft SQL Server.
- Модуль hunspell-dict, предоставляющий словари для ряда языков.
- Расширение hypopg, обеспечивающее поддержку гипотетических индексов в Postgres Pro.
- Модуль jsquery, реализующий специальный язык запросов для эффективного поиска по индексу в структурированных данных JSONB.
- Служба мониторинга mamonsu, исполненная в виде агента Zabbix.
- Модуль mchar, предоставляющий дополнительный тип данных для совместимости с Microsoft SQL Server.
- Модуль online_analyze, привносящий набор функций, которые немедленно обновляют статистику в целевых таблицах после операций `INSERT`, `UPDATE`, `DELETE` или `SELECT INTO` в них.
- pgbadger — инструмент для быстрого анализа журналов Postgres Pro с созданием подробных отчётов и графиков.
- Пул соединений pgbouncer.
- Модуль pg_integrity_check, осуществляющий вычисление и проверку контрольных сумм отслеживаемых файлов. (Только сертифицированная редакция.)
- Расширение pg_proaudit, позволяющее отслеживать различные события, связанные с безопасностью.
- Расширение pgpro_autopart, реализующее динамическое создание секций.
- pg_probackup — менеджер резервного копирования и восстановления.
- Решение *pg_probackup3* для резервного копирования и восстановления кластеров баз данных Postgres Pro.
- Модуль pgpro_bindump для управления операциями резервного копирования и восстановления. В модуле реализованы специализированные команды репликации для расширенного протокола репликации и собственный формат архивирования. Модуль не требует подключения по SSH. Он разработан специально для использования вместе с утилитой *pg_probackup3*.
- Утилита pgpro_controldata, показывающая управляющую информацию кластера БД PostgreSQL/Postgres Pro и параметры совместимости кластера и/или сервера.
- Расширение pgpro_pwr, позволяющее получать отчёты по нагрузке, полезные для выявления наиболее ресурсоёмких операций в базе данных.
- Расширение pgpro_stats, которое собирает статистику выполнения SQL-операторов, подсчитывает статистику событий ожидания и предоставляет прочие полезные метрики, которые в PostgreSQL нельзя получить никак иначе. Оно также обеспечивает возможность трассировки сеансов приложений и может создавать представления, эмулирующие другие расширения для сбора статистики.

- Утилита pgpro_tune для автоматической настройки параметров конфигурации Postgres Pro. Она предоставляет набор встроенных пресетов, включая пресет для настройки, специфичной для 1С.
- Модуль pg_query_state, дающий возможность узнавать текущее состояние выполнения запросов в обслуживающем процессе.
- Утилита pg_repack для реорганизации таблиц.
- Модуль pg_tsparser — альтернативный анализатор текстового поиска.
- Модуль pg_variables, предоставляющий функции для работы с переменными различных типов. Для облегчения миграции кода Oracle, обрабатывающего коллекции, доступны следующие виды функций: функции, позволяющие работать с общими переменными-коллекциями, к элементам которых можно получить доступ по ключу, имеющему целочисленный или текстовый тип; функции, обеспечивающие функциональность итератора для любых коллекций.
- Расширение pgvector, предоставляющее поиск сходства векторов в Postgres Pro.
- Расширение pg_wait_sampling, предназначенное для периодического сбора статистики по событиям ожидания. Воспользовавшись этим расширением, вы сможете понять характер активности сервера, в том числе увидеть текущие события ожидания во всех обычных и фоновых рабочих процессах.
- Модуль plantuner, добавляющий поддержку указаний для планировщика, подключающих или отключающих определённые индексы при выполнении запроса.
- Модуль rum, предоставляющий индекс RUM на базе GIN.
- Модуль shared_ispell, позволяющий разместить словари в общей памяти.

Postgres Pro предоставляет расширенные возможности и функциональность, поэтому предъявляет более строгие требования, ограничивающие множество поддерживаемых операционных систем.

Важно

Postgres Pro Standard работает на всех основных операционных системах Linux. Любые упоминания Windows или другой операционной системы, отличной от Linux, в этой документации неприменимы к Postgres Pro.

Выпуски Postgres Pro Standard следуют за выпусками PostgreSQL, хотя иногда могут выпускаться чаще. Схема версионирования Postgres Pro Standard основана на схеме версионирования PostgreSQL и включает дополнительную цифру.

3. Краткая история PostgreSQL

Объектно-реляционная система управления базами данных, именуемая сегодня PostgreSQL, произошла от пакета POSTGRES, написанного в Беркли, Калифорнийском университете. После десятилетий разработки PostgreSQL стал самой развитой СУБД с открытым исходным кодом.

3.1. Проект POSTGRES в Беркли

Проект POSTGRES, возглавляемый профессором Майклом Стоунбрейкером, спонсировали агентство DARPA при Минобороны США, Управление военных исследований (ARO), Национальный Научный Фонд (NSF) и компания ESL, Inc. Реализация POSTGRES началась в 1986 г. Первоначальные концепции системы были представлены в документе ston86, а описание первой модели данных появилось в rowe87. Проект системы правил тогда был представлен в ston87a. Суть и архитектура менеджера хранилища были расписаны в ston87b.

С тех пор POSTGRES прошёл несколько этапов развития. Первая «демоверсия» заработала в 1987 и была показана в 1988 на конференции ACM-SIGMOD. Версия 1, описанная в ston90a, была выпущена для нескольких внешних пользователей в июне 1989. В ответ на критику первой системы правил (ston89), она была переделана (ston90b), и в версии 2, выпущенной в июне 1990, была уже новая система правил. В 1991 вышла версия 3, в которой появилась поддержка различных менеджеров хранилища, улучшенный исполнитель запросов и переписанная система правил. Последующие выпуски до Postgres95 (см. ниже) в основном были направлены на улучшение портируемости и надёжности.

POSTGRES применялся для реализации множества исследовательских и производственных задач. В их числе: система анализа финансовых данных, пакет мониторинга работы реактивных двигателей, база данных наблюдений за астероидами, база данных медицинской информации, несколько географических информационных систем. POSTGRES также использовался для обучения в нескольких университетах. Наконец, компания Illustra Information Technologies (позже ставшая частью *Informix*, которая сейчас принадлежит *IBM*) воспользовалась кодом и нашла ему коммерческое применение. В конце 1992 POSTGRES стал основной СУБД научного вычислительного проекта Sequoia 2000, описанного в ston92.

В 1993 число внешних пользователей удвоилось. Стало очевидно, что обслуживание кода и поддержка занимает слишком много времени, и его не хватает на исследования. Для снижения этой нагрузки проект POSTGRES в Беркли был официально закрыт на версии 4.2.

3.2. Postgres95

В 1994 Эндри Ю и Джолли Чен добавили в POSTGRES интерпретатор языка SQL. Уже с новым именем Postgres95 был опубликован в Интернете и начал свой путь как потомок разработанного в Беркли POSTGRES, с открытым исходным кодом.

Код Postgres95 был приведён в полное соответствие с ANSI C и уменьшился на 25%. Благодаря множеству внутренних изменений он стал быстрее и удобнее. Postgres95 версии 1.0.x работал примерно на 30–50% быстрее POSTGRES версии 4.2 (по тестам Wisconsin Benchmark). Помимо исправления ошибок, произошли следующие изменения:

- На смену языку запросов PostQUEL пришёл SQL (реализованный в сервере). (Интерфейсная библиотека libpq унаследовала своё имя от PostQUEL.) Подзапросы не поддерживались до выхода PostgreSQL (см. ниже), хотя их можно было имитировать в Postgres95 с помощью пользовательских функций SQL. Были заново реализованы агрегатные функции. Также появилась поддержка предложения `GROUP BY`.
- Для интерактивных SQL-запросов была разработана новая программа (psql), которая использовала GNU Readline. Старая программа monitor стала не нужна.
- Появилась новая клиентская библиотека `libpgtcl` для поддержки Tcl-клиентов. Пример оболочки, `pgtclsh`, представлял новые команды Tcl для взаимодействия программ Tcl с сервером Postgres95.
- Был усовершенствован интерфейс для работы с большими объектами. Единственным механизмом хранения таких данных стали инверсионные объекты. (Инверсионная файловая система была удалена.)
- Удалена система правил на уровне экземпляров; перезаписывающие правила сохранились.
- С исходным кодом стали распространяться краткие описания возможностей стандартного SQL, а также самого Postgres95.
- Для сборки использовался GNU make (вместо BSD make). Кроме того, стало возможно скомпилировать Postgres95 с немодифицированной версией GCC (было исправлено выравнивание данных).

3.3. PostgreSQL

В 1996 г. стало понятно, что имя «Postgres95» не выдержит испытание временем. Мы выбрали новое имя, PostgreSQL, отражающее связь между оригинальным POSTGRES и более поздними версиями с поддержкой SQL. В то же время, мы продолжили нумерацию версий с 6.0, вернувшись к последовательности, начатой в проекте Беркли POSTGRES.

Postgres по-прежнему считается официальным именем проекта не только по традиции, но и потому, что Postgres произносить проще, чем PostgreSQL.

В процессе разработки Postgres95 основными задачами были поиск и понимание существующих проблем в серверном коде. С переходом к PostgreSQL акценты сместились к реализации новых функций и возможностей, хотя работа продолжается во всех направлениях.

4. Соглашения

Следующие соглашения используются в описаниях команд: квадратные скобки (`[` и `]`) обозначают необязательные части. Фигурные скобки (`{` и `}`) и вертикальная черта (`|`) обозначают выбор одного из предложенных вариантов. Многоточие (`...`) означает, что предыдущий элемент можно повторить.

Иногда для ясности команды SQL предваряются приглашением `=>`, а команды оболочки — приглашением `$`.

Под *администратором* здесь обычно понимается человек, ответственный за установку и запуск сервера, тогда как *пользователь* — кто угодно, кто использует или желает использовать любой компонент системы Postgres Pro. Эти роли не следует воспринимать слишком узко, так как в этой книге нет определённых допущений относительно процедур системного администрирования.

5. Как правильно сообщить об ошибке

Если вы найдёте ошибку в Postgres Pro, дайте нам знать о ней. Благодаря вашему отчёту об ошибке, Postgres Pro станет ещё более надёжным, ведь даже при самом высоком качестве кода нельзя гарантировать, что каждый блок и каждая функция Postgres Pro будет работать везде и при любых обстоятельствах.

Следующие предложения призваны помочь вам в составлении отчёта об ошибке, который можно будет обработать эффективно. Мы не требуем их неукоснительного выполнения, но всё же лучше следовать им для общего блага.

5.1. Диагностика ошибок

Прежде чем сообщать об ошибке, пожалуйста, прочитайте и перечитайте документацию и убедитесь, что вообще возможно сделать то, что вы хотите. Если из документации неясно, можно это сделать или нет, пожалуйста, сообщите и об этом (тогда это ошибка в документации). Если выясняется, что программа делает что-то не так, как написано в документации, это тоже ошибка. Вот лишь некоторые примеры возможных ошибок:

- Программа завершается с аварийным сигналом или сообщением об ошибке операционной системы, указывающей на проблему в программе. (В качестве контрпримера можно привести сообщение «Нет места на диске» — эту проблему вы должны решить сами.)
- Программа выдаёт неправильный результат для любых вводимых данных.
- Программа отказывается принимать допустимые (согласно документации) данные.
- Программа принимает недопустимые данные без сообщения об ошибке или предупреждения. Но помните: то, что вы считаете недопустимым, мы можем считать приемлемым расширением или совместимым с принятой практикой.
- Не удаётся установить Postgres Pro на поддерживаемой платформе, выполняя соответствующие инструкции.

Здесь под «программой» подразумевается произвольный исполняемый файл, а не исключительно серверный процесс.

Медленная работа или высокая загрузка ресурсов — это не обязательно ошибка. Также может не быть ошибкой какое-то несоответствие стандарту SQL, если только явно не декларируется соответствие в данном аспекте.

5.2. Что сообщать

Сообщая о проблеме, постарайтесь изложить все факты. В каждом отчёте об ошибке следует указать:

- Точную последовательность действий для воспроизведения проблемы, начиная с *запуска программы*. Она должна быть самодостаточной; если вывод зависит от данных в таблицах, то недостаточно сообщить один лишь `SELECT`, без предшествующих операторов `CREATE TABLE` и `INSERT`.

  Лучший формат теста для проблем с SQL — файл, который можно передать программе psql и увидеть проблему. (И убедитесь, что в вашем файле `~/.psqlrc` ничего нет.) Самый простой способ получить такой файл — выгрузить объявления таблиц и данные, необходимые для создания полигона, с помощью pg_dump, а затем добавить проблемный запрос. Постарайтесь сократить размер вашего тестового примера, хотя это не абсолютно необходимо. Если ошибка воспроизводится, мы найдём её в любом случае.

  Если ваше приложение использует какой-то другой клиентский интерфейс, например PHP, пожалуйста, попытайтесь свести ошибку к проблемным запросам.

- Результат, который вы получаете. Если есть сообщение об ошибке, покажите его. Если программа завершается ошибкой операционной системы, сообщите о ней. Или если ничего не происходит, отразите это. Даже если в результате вашего теста происходит сбой программы или что-то очевидное, мы можем не наблюдать этого у себя. Проще всего будет скопировать текст с терминала, если это возможно.

  Примечание

  Если вы упоминаете сообщение об ошибке, пожалуйста, укажите его в наиболее полной форме. Например, в psql, для этого сначала выполните `\set VERBOSITY verbose`. Если вы цитируете сообщения из журнала событий сервера, присвойте параметру выполнения log_error_verbosity значение `verbose`, чтобы журнал был наиболее подробным.

  Примечание

  В случае критических ошибок сообщение на стороне клиента может не содержать всю необходимую информацию. Пожалуйста, также изучите журнал сервера баз данных.

- Очень важно отметить, какой результат вы ожидали получить. Пожалуйста, приводите ожидаемый результат, если это уместно.
- Все параметры командной строки и другие параметры запуска, включая все связанные переменные окружения или файлы конфигурации, которые вы изменяли.
- Всё, что вы делали не так, как написано в инструкциях по установке.
- Версию Postgres Pro. Чтобы выяснить версию сервера, к которому вы подключены, можно выполнить команду `SELECT pgpro_version();`. Большинство исполняемых программ также поддерживают параметр `--version`; как минимум должно работать `postgres --version` и `psql --version`.
- Сведения о платформе, включая название и версию ядра, библиотеки C, характеристики процессора, памяти и т. д.

5.3. Куда сообщать

В общем случае посылать сообщения об ошибках следует по адресу `<bugs@postgrespro.ru>`. Мы просим вас написать информативную тему письма, возможно включив в неё часть сообщения об ошибке.

Не отправляйте отчёты об ошибках, специфичных для Postgres Pro, на адрес поддержки PostgreSQL, так как Postgres Pro не поддерживается сообществом PostgreSQL. Однако вы можете сообщать по адресу `<pgsql-bugs@lists.postgresql.org>` обо всех ошибках, связанных с PostgreSQL.

Даже если ваша ошибка не связана с особенностями Postgres Pro, не посылайте сообщения в списки рассылки для пользователей, например в `<pgsql-sql@lists.postgresql.org>` или `<pgsql-general@lists.postgresql.org>`. Эти рассылки предназначены для ответов на вопросы пользователей, и их подписчики обычно не хотят получать сообщения об ошибках, более того, они вряд ли исправят их.

Также, пожалуйста, *не* отправляйте отчёты об ошибках в список `<pgsql-hackers@lists.postgresql.org>`. Этот список предназначен для обсуждения разработки PostgreSQL, и будет лучше, если сообщения об ошибках будут существовать отдельно. Мы перенесём обсуждение вашей ошибки в `pgsql-hackers`, если связанная с PostgreSQL проблема потребует дополнительного рассмотрения.

Часть I. Введение

Добро пожаловать на встречу с Postgres Pro. Во введении вы сможете получить общее представление о Postgres Pro, реляционных базах данных и языке SQL. Этот материал будет понятен даже тем, кто обладает общими знаниями о компьютерах и не имеет опыта программирования и навыков использования Unix-систем. Цель вводной части — познакомить вас на практике с ключевыми аспектами системы Postgres Pro, поэтому затрагиваемые в ней темы не будут рассматриваться максимально глубоко и полно.

После прочтения этого введения рекомендуется перейти к разделу Часть II, чтобы лучше понять язык SQL, или к Части IV, посвящённой разработке приложений с Postgres Pro. Для изучения SQL можно воспользоваться демонстрационной базой данных, описанной в Приложении M. Тем же, кто устанавливает и администрирует сервер самостоятельно, следует также прочитать Часть III.

Глава 1. Начало

1.1. Установка

Прежде чем вы сможете использовать Postgres Pro, вы конечно должны его установить. Однако возможно, что Postgres Pro уже установлен у вас, либо потому что он включён в вашу операционную систему, либо его установил системный администратор. Если это так, обратитесь к документации по операционной системе или к вашему администратору и узнайте, как получить доступ к Postgres Pro.

Если вы устанавливаете Postgres Pro самостоятельно, воспользуйтесь инструкциями по установке (Глава 16), а закончив установку, вернитесь к этому введению. Обязательно прочитайте и выполните указания по установке соответствующих переменных окружения.

Если ваш администратор выполнил установку не с параметрами по умолчанию, вам может потребоваться проделать дополнительную работу. Например, если сервер баз данных установлен на удалённом компьютере, вам нужно будет указать в переменной окружения `PGHOST` имя этого компьютера. Вероятно, также придётся установить переменную окружения `PGPORT`. То есть, если вы пытаетесь запустить клиентское приложение и оно сообщает, что не может подключиться к базе данных, вы должны обратиться к вашему администратору. Если это вы сами, вам следует обратиться к документации и убедиться в правильности настройки окружения. Если вы не поняли, о чём здесь идёт речь, перейдите к следующему разделу.

1.2. Основы архитектуры

Прежде чем продолжить, вы должны разобраться в основах архитектуры системы Postgres Pro. Составив картину взаимодействия частей Postgres Pro, вы сможете лучше понять материал этой главы.

Говоря техническим языком, Postgres Pro реализован в архитектуре клиент-сервер. Рабочий сеанс Postgres Pro включает следующие взаимодействующие процессы (программы):

- Главный серверный процесс, управляющий файлами баз данных, принимающий подключения клиентских приложений и выполняющий различные запросы клиентов к базам данных. Эта программа сервера БД называется `postgres`.
- Клиентское приложение пользователя, желающее выполнять операции в базе данных. Клиентские приложения могут быть очень разнообразными: это может быть текстовая утилита, графическое приложение, веб-сервер, использующий базу данных для отображения веб-страниц, или специализированный инструмент для обслуживания БД. Некоторые клиентские приложения поставляются в составе дистрибутива Postgres Pro, однако большинство создают сторонние разработчики.

Как и в других типичных клиент-серверных приложениях, клиент и сервер могут располагаться на разных компьютерах. В этом случае они взаимодействуют по сети TCP/IP. Важно не забывать это и понимать, что файлы, доступные на клиентском компьютере, могут быть недоступны (или доступны только под другим именем) на компьютере-сервере.

Сервер Postgres Pro может обслуживать одновременно несколько подключений клиентов. Для этого он запускает («порождает») отдельный процесс для каждого подключения. Можно сказать, что клиент и серверный процесс общаются, не затрагивая главный процесс `postgres`. Таким образом, главный серверный процесс всегда работает и ожидает подключения клиентов, принимая которые, он организует взаимодействие клиента и отдельного серверного процесса. (Конечно всё это происходит незаметно для пользователя, а эта схема рассматривается здесь только для понимания.)

1.3. Создание базы данных

Первое, как можно проверить, есть ли у вас доступ к серверу баз данных, — это попытаться создать базу данных. Работающий сервер Postgres Pro может управлять множеством баз данных, что позволяет создавать отдельные базы данных для разных проектов и пользователей.

Возможно, ваш администратор уже создал базу данных для вас. В этом случае вы можете пропустить этот этап и перейти к следующему разделу.

Для создания базы данных, в этом примере названной `mydb`, выполните следующую команду:

```
$ createdb mydb
```

Если вы не увидите никаких сообщений, значит операция была выполнена успешно и продолжение этого раздела можно пропустить.

Если вы видите сообщение типа:

```
createdb: command not found
```

значит Postgres Pro не был установлен правильно. Либо он не установлен вообще, либо в путь поиска команд оболочки не включён его каталог. Попробуйте вызвать ту же команду, указав абсолютный путь:

```
$ /usr/local/pgsql/bin/createdb mydb
```

У вас этот путь может быть другим. Свяжитесь с вашим администратором или проверьте, как были выполнены инструкции по установке, чтобы исправить ситуацию.

Ещё один возможный ответ:

```
createdb: ошибка: не удалось подключиться к серверу через сокет "/tmp/.s.PGSQL.5432":
No such file or directory
        Он действительно работает локально и принимает
        соединения через этот сокет?
```

Это означает, что сервер не работает или `createdb` не может к нему подключиться. И в этом случае пересмотрите инструкции по установке или обратитесь к администратору.

Также вы можете получить сообщение:

```
createdb: ошибка: не удалось подключиться к серверу через сокет"/
tmp/.s.PGSQL.5432":ВАЖНО:  роль "joe" не существует
```

где фигурирует ваше имя пользователя. Это говорит о том, что администратор не создал учётную запись Postgres Pro для вас. (Учётные записи Postgres Pro отличаются от учётных записей пользователей операционной системы.) Если вы сами являетесь администратором, прочитайте Главу 20, где написано, как создавать учётные записи. Для создания нового пользователя вы должны стать пользователем операционной системы, под именем которого был установлен Postgres Pro (обычно это `postgres`). Также возможно, что вам назначено имя пользователя Postgres Pro, не совпадающее с вашим именем в ОС; в этом случае вам нужно явно указать ваше имя пользователя Postgres Pro, используя ключ `-U` или установив переменную окружения `PGUSER`.

Если у вас есть учётная запись пользователя, но нет прав на создание базы данных, вы увидите сообщение:

```
createdb: ошибка: создать базу данных не удалось:
  ОШИБКА:  нет прав на создание базы данных
```

Создавать базы данных разрешено не всем пользователям. Если Postgres Pro отказывается создавать базы данных для вас, значит вам необходимо соответствующее разрешение. В этом случае обратитесь к вашему администратору. Если вы устанавливали Postgres Pro сами, то для целей этого введения вы должны войти в систему с именем пользователя, запускающего сервер БД. [1]

---

[1] Объяснить это поведение можно так: Учётные записи пользователей Postgres Pro отличаются от учётных записей операционной системы. При подключении к базе данных вы можете указать, с каким именем пользователя Postgres Pro нужно подключаться. По умолчанию же используется имя, с которым вы зарегистрированы в операционной системе. При этом получается, что в Postgres Pro всегда есть учётная запись с именем, совпадающим с именем системного

Вы также можете создавать базы данных с другими именами. Postgres Pro позволяет создавать сколько угодно баз данных. Имена баз данных должны начинаться с буквы и быть не длиннее 63 символов. В качестве имени базы данных удобно использовать ваше текущее имя пользователя. Многие утилиты предполагают такое имя по умолчанию, так что вы сможете упростить ввод команд. Чтобы создать базу данных с таким именем, просто введите:

```
$ createdb
```

Если вы больше не хотите использовать вашу базу данных, вы можете удалить её. Например, если вы владелец (создатель) базы данных `mydb`, вы можете уничтожить её, выполнив следующую команду:

```
$ dropdb mydb
```

(Эта команда не считает именем БД по умолчанию имя текущего пользователя, вы должны явно указать его.) В результате будут физически удалены все файлы, связанные с базой данных, и так как отменить это действие нельзя, не выполняйте его, не подумав о последствиях.

Узнать о командах `createdb` и `dropdb` больше можно в справке createdb и dropdb.

1.4. Подключение к базе данных

Создав базу данных, вы можете обратиться к ней:

- Запустив терминальную программу Postgres Pro под названием *psql*, в которой можно интерактивно вводить, редактировать и выполнять команды SQL.
- Используя существующие графические инструменты, например, pgAdmin или офисный пакет с поддержкой ODBC или JDBC, позволяющий создавать и управлять базой данных. Эти возможности здесь не рассматриваются.
- Написав собственное приложение, используя один из множества доступных языковых интерфейсов. Подробнее это рассматривается в Части IV.

Чтобы работать с примерами этого введения, начните с `psql`. Подключиться с его помощью к базе данных `mydb` можно, введя команду:

```
$ psql mydb
```

Если имя базы данных не указать, она будет выбрана по имени пользователя. Об этом уже рассказывалось в предыдущем разделе, посвящённом команде `createdb`.

В `psql` вы увидите следующее сообщение:

```
psql (17.6.1)
Type "help" for help.

mydb=>
```

Последняя строка может выглядеть и так:

```
mydb=#
```

Что показывает, что вы являетесь суперпользователем, и так скорее всего будет, если вы устанавливали экземпляр Postgres Pro сами. В этом случае на вас не будут распространяться никакие ограничения доступа, но для целей данного введения это не важно.

Если вы столкнулись с проблемами при запуске `psql`, вернитесь к предыдущему разделу. Команды `createdb` и `psql` подключаются к серверу одинаково, так что если первая работает, должна работать и вторая.

Последняя строка в выводе `psql` — это приглашение, которое показывает, что `psql` ждёт ваших команд и вы можете вводить SQL-запросы в рабочей среде `psql`. Попробуйте эти команды:

```
mydb=> SELECT pgpro_version();
```

пользователя, запускающего сервер, и к тому же этот пользователь всегда имеет права на создание баз данных. И чтобы подключиться с именем этого пользователя Postgres Pro, необязательно входить с этим именем в систему, достаточно везде передавать его с параметром `-U`.

```
                                          version
--------------------------------------------------------------------
-----------------------
 PostgresPro 17.6.1 on x86_64-pc-linux-gnu, compiled by gcc (Debian 4.9.2-10) 4.9.2,
 64-bit
(1 row)

mydb=> SELECT current_date;
    date
------------
 2016-01-07
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

В программе `psql` есть множество внутренних команд, которые не являются SQL-операторами. Они начинаются с обратной косой черты, «`\`». Например, вы можете получить справку по различным SQL-командам Postgres Pro, введя:

```
mydb=> \h
```

Чтобы выйти из `psql`, введите:

```
mydb=> \q
```

и `psql` завершит свою работу, а вы вернётесь в командную оболочку операционной системы. (Чтобы узнать о внутренних командах, введите `\?` в приглашении командной строки `psql`.) Все возможности `psql` документированы в справке psql. В этом руководстве мы не будем использовать эти возможности явно, но вы можете изучить их и применять при удобном случае.

Глава 2. Язык SQL

2.1. Введение

В этой главе рассматривается использование SQL для выполнения простых операций. Она призвана только познакомить вас с SQL, но ни в коей мере не претендует на исчерпывающее руководство. Про SQL написано множество книг, включая melt93 и date97. При этом следует учитывать, что некоторые возможности языка Postgres Pro являются расширениями стандарта.

В следующих примерах мы предполагаем, что вы создали базу данных `mydb`, как описано в предыдущей главе, и смогли запустить psql.

2.2. Основные понятия

Postgres Pro — это *реляционная система управления базами данных* (РСУБД). Это означает, что это система управления данными, представленными в виде *отношений* (relation). Отношение — это математически точное обозначение *таблицы*. Хранение данных в таблицах так распространено сегодня, что это кажется самым очевидным вариантом, хотя есть множество других способов организации баз данных. Например, файлы и каталоги в Unix-подобных операционных системах образуют иерархическую базу данных, а сегодня активно развиваются объектно-ориентированные базы данных.

Любая таблица представляет собой именованный набор *строк*. Все строки таблицы имеют одинаковый набор именованных *столбцов*, при этом каждому столбцу назначается определённый тип данных. Хотя порядок столбцов во всех строках фиксирован, важно помнить, что SQL не гарантирует какой-либо порядок строк в таблице (хотя их можно явно отсортировать при выводе).

Таблицы объединяются в базы данных, а набор баз данных, управляемый одним экземпляром сервера Postgres Pro, образует *кластер* баз данных.

2.3. Создание таблицы

Вы можете создать таблицу, указав её имя и перечислив все имена столбцов и их типы:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- минимальная температура дня
    temp_hi         int,           -- максимальная температура дня
    prcp            real,          -- уровень осадков
    date            date
);
```

Весь этот текст можно ввести в `psql` вместе с символами перевода строк. `psql` понимает, что команда продолжается до точки с запятой.

В командах SQL можно свободно использовать пробельные символы (пробелы, табуляции и переводы строк). Это значит, что вы можете ввести команду, выровняв её по-другому или даже уместив в одной строке. Два минуса («`--`») обозначают начало комментария. Всё, что идёт за ними до конца строки, игнорируется. SQL не чувствителен к регистру в ключевых словах и идентификаторах, за исключением идентификаторов, взятых в кавычки (в данном случае это не так).

`varchar(80)` определяет тип данных, допускающий хранение произвольных символьных строк длиной до 80 символов. `int` — обычный целочисленный тип. `real` — тип для хранения чисел с плавающей точкой одинарной точности. `date` — тип даты. (Да, столбец типа `date` также называется `date`. Это может быть удобно или вводить в заблуждение — как посмотреть.)

Postgres Pro поддерживает стандартные типы SQL: `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp` и `interval`, а также другие универсальные типы и богатый набор геометрических типов. Кроме того, Postgres Pro можно расширять, создавая набор собственных типов данных. Как следствие, имена типов не являются ключевыми словами в данной записи, кроме тех случаев, когда это требуется для реализации особых конструкций стандарта SQL.

Во втором примере мы сохраним в таблице города и их географическое положение:

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

Здесь `point` — пример специфического типа данных Postgres Pro.

Наконец, следует сказать, что если вам больше не нужна какая-либо таблица, или вы хотите пересоздать её по-другому, вы можете удалить её, используя следующую команду:

```
DROP TABLE имя_таблицы;
```

2.4. Добавление строк в таблицу

Для добавления строк в таблицу используется оператор `INSERT`:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Заметьте, что для всех типов данных применяются довольно очевидные форматы. Константы, за исключением простых числовых значений, обычно заключаются в апострофы (`'`), как в данном примере. Тип `date` на самом деле очень гибок и принимает разные форматы, но в данном введении мы будем придерживаться простого и однозначного.

Тип `point` требует ввода пары координат, например таким образом:

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

Показанный здесь синтаксис требует, чтобы вы запомнили порядок столбцов. Можно также применить альтернативную запись, перечислив столбцы явно:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Вы можете перечислить столбцы в другом порядке, если желаете опустить некоторые из них, например, если уровень осадков (столбец prcp) неизвестен:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Многие разработчики предпочитают явно перечислять столбцы, а не полагаться на их порядок в таблице.

Пожалуйста, введите все показанные выше команды, чтобы у вас были данные, с которыми можно будет работать дальше.

Вы также можете загрузить большой объём данных из обычных текстовых файлов, применив команду `COPY`. Обычно это будет быстрее, так как команда `COPY` оптимизирована для такого применения, хотя и менее гибка, чем `INSERT`. Например, её можно использовать так:

```
COPY weather FROM '/home/user/weather.txt';
```

здесь подразумевается, что данный файл доступен на компьютере, где работает серверный процесс, а не на клиенте, так как указанный файл будет прочитан непосредственно на сервере. Данные, добавленные выше в таблицу weather, можно также добавить из файла, содержащего следующее (значения разделены символом табуляции):

```
San Francisco    46    50    0.25    1994-11-27
San Francisco    43    57    0.0    1994-11-29
Hayward    37    54    \N    1994-11-29
```

Подробнее об этом вы можете узнать в описании команды COPY.

2.5. Выполнение запроса

Чтобы получить данные из таблицы, нужно выполнить *запрос*. Для этого предназначен SQL-оператор `SELECT`. Он состоит из нескольких частей: выборки (в которой перечисляются столбцы, которые должны быть получены), списка таблиц (в нём перечисляются таблицы, из которых будут получены данные) и необязательного условия (определяющего ограничения). Например, чтобы получить все строки таблицы `weather`, введите:

```
SELECT * FROM weather;
```

Здесь `*` — это краткое обозначение «всех столбцов».[1] Таким образом, это равносильно записи:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

В результате должно получиться:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

В списке выборки вы можете писать не только ссылки на столбцы, но и выражения. Например, вы можете написать:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

И получить в результате:

```
     city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Обратите внимание, как предложение `AS` позволяет переименовать выходной столбец. (Само слово `AS` можно опускать.)

Запрос можно дополнить «условием», добавив предложение `WHERE`, ограничивающее множество возвращаемых строк. В предложении `WHERE` указывается логическое выражение (проверка истинности), которое служит фильтром строк: в результате оказываются только те строки, для которых это выражение истинно. В этом выражении могут присутствовать обычные логические операторы (`AND`, `OR` и `NOT`). Например, следующий запрос покажет, какая погода была в Сан-Франциско в дождливые дни:

```
SELECT * FROM weather
    WHERE city = 'San Francisco' AND prcp > 0.0;
```

Результат:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

Вы можете получить результаты запроса в определённом порядке:

```
SELECT * FROM weather
    ORDER BY city;
```

```
     city      | temp_lo | temp_hi | prcp |    date
```

---

[1] Хотя запросы `SELECT *` часто пишут экспромтом, это считается плохим стилем в производственном коде, так как результат таких запросов будет меняться при добавлении новых столбцов.

```
---------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

В этом примере порядок сортировки определён не полностью, поэтому вы можете получить строки Сан-Франциско в любом порядке. Но вы всегда получите результат, показанный выше, если напишете:

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

Если требуется, вы можете убрать дублирующиеся строки из результата запроса:

```
SELECT DISTINCT city
    FROM weather;
```

```
     city
---------------
 Hayward
 San Francisco
(2 rows)
```

И здесь порядок строк также может варьироваться. Чтобы получать неизменные результаты, соедините предложения `DISTINCT` и `ORDER BY`: [2]

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

2.6. Соединения таблиц

До этого все наши запросы обращались только к одной таблице. Однако запросы могут также обращаться сразу к нескольким таблицам или обращаться к той же таблице так, что одновременно будут обрабатываться разные наборы её строк. Запросы, обращающиеся к разным таблицам (или нескольким экземплярам одной таблицы), называются *соединениями* (JOIN). Такие запросы содержат выражение, указывающее, какие строки одной таблицы нужно объединить со строками другой таблицы. Например, чтобы вернуть все погодные события вместе с координатами соответствующих городов, база данных должна сравнить столбец `city` каждой строки таблицы `weather` со столбцом `name` всех строк таблицы `cities` и выбрать пары строк, для которых эти значения совпадают. [3] Это можно сделать с помощью следующего запроса:

```
SELECT * FROM weather JOIN cities ON city = name;
```

```
     city     |temp_lo|temp_hi| prcp|    date    |     name      | location
--------------+-------+-------+-----+------------+---------------+----------
 San Francisco|     46|     50| 0.25| 1994-11-27| San Francisco| (-194,53)
 San Francisco|     43|     57|    0| 1994-11-29| San Francisco| (-194,53)
(2 rows)
```

Обратите внимание на две особенности полученных данных:

- В результате нет строки с городом Хейуорд (Hayward). Так получилось потому, что в таблице `cities` нет строки для данного города, а при соединении все строки таблицы `weather`, для которых не нашлось соответствие, опускаются. Вскоре мы увидим, как это можно исправить.
- Название города оказалось в двух столбцах. Это правильно и объясняется тем, что столбцы таблиц `weather` и `cities` были объединены. Хотя на практике это нежелательно, поэтому лучше перечислить нужные столбцы явно, а не использовать `*`:

  ```
  SELECT city, temp_lo, temp_hi, prcp, date, location
  ```

---

[2] В некоторых СУБД, включая старые версии Postgres Pro, реализация предложения `DISTINCT` автоматически упорядочивает строки, так что `ORDER BY` добавлять не обязательно. Но стандарт SQL этого не требует и текущая версия Postgres Pro не гарантирует определённого порядка строк после `DISTINCT`.

[3] Это не совсем точная модель. Обычно соединения выполняются эффективнее (сравниваются не все возможные пары строк), но это скрыто от пользователя.

```
      FROM weather JOIN cities ON city = name;
```

Так как все столбцы имеют разные имена, анализатор запроса автоматически понимает, к какой таблице они относятся. Если бы имена столбцов в двух таблицах повторялись, вам пришлось бы *дополнить* имена столбцов, конкретизируя, что именно вы имели в виду:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather JOIN cities ON weather.city = cities.name;
```

Вообще хорошим стилем считается указывать полные имена столбцов в запросе соединения, чтобы запрос не поломался, если позже в таблицы будут добавлены столбцы с повторяющимися именами.

Запросы соединения, которые вы видели до этого, можно также записать в другом виде:

```
SELECT *
    FROM weather, cities
    WHERE city = name;
```

Этот синтаксис появился до синтаксиса `JOIN`/`ON`, принятого в SQL-92. Таблицы просто перечисляются в предложении `FROM`, а выражение сравнения добавляется в предложение `WHERE`. Результаты, получаемые с использованием старого неявного синтаксиса и нового явного синтаксиса `JOIN`/`ON`, будут одинаковыми. Однако, читая запрос, понять явный синтаксис проще: условие соединения вводится с помощью специального ключевого слова, а раньше это условие включалось в предложение `WHERE` наряду с другими условиями.

Сейчас мы выясним, как вернуть записи о погоде в городе Хейуорд. Мы хотим, чтобы запрос просканировал таблицу `weather` и для каждой её строки нашёл соответствующую строку в таблице `cities`. Если же такая строка не будет найдена, мы хотим, чтобы вместо значений столбцов из таблицы `cities` были подставлены «пустые значения». Запросы такого типа называются *внешними соединениями*. (Соединения, которые мы видели до этого, называются *внутренними*.) Эта команда будет выглядеть так:

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON weather.city = cities.name;

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

Этот запрос называется *левым внешним соединением*, потому что из таблицы в левой части оператора будут выбраны все строки, а из таблицы справа только те, которые удалось сопоставить каким-нибудь строкам из левой. При выводе строк левой таблицы, для которых не удалось найти соответствия в правой, вместо столбцов правой таблицы подставляются пустые значения (NULL).

Упражнение: Существуют также правые внешние соединения и полные внешние соединения. Попробуйте выяснить, что они собой представляют.

В соединении мы также можем замкнуть таблицу на себя. Это называется *замкнутым соединением*. Например, представьте, что мы хотим найти все записи погоды, в которых температура лежит в диапазоне температур других записей. Для этого мы должны сравнить столбцы `temp_lo` и `temp_hi` каждой строки таблицы `weather` со столбцами `temp_lo` и `temp_hi` другого набора строк `weather`. Это можно сделать с помощью следующего запроса:

```
SELECT w1.city, w1.temp_lo AS low, w1.temp_hi AS high,
       w2.city, w2.temp_lo AS low, w2.temp_hi AS high
    FROM weather w1 JOIN weather w2
        ON w1.temp_lo < w2.temp_lo AND w1.temp_hi > w2.temp_hi;
```

```
     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Здесь мы ввели новые обозначения таблицы weather: `W1` и `W2`, чтобы можно было различить левую и правую стороны соединения. Вы можете использовать подобные псевдонимы и в других запросах для сокращения:

```
SELECT *
    FROM weather w JOIN cities c ON w.city = c.name;
```

Вы будете встречать сокращения такого рода довольно часто.

2.7. Агрегатные функции

Как большинство других серверов реляционных баз данных, Postgres Pro поддерживает *агрегатные функции*. Агрегатная функция вычисляет единственное значение, обрабатывая множество строк. Например, есть агрегатные функции, вычисляющие: `count` (количество), `sum` (сумму), `avg` (среднее), `max` (максимум) и `min` (минимум) для набора строк.

К примеру, мы можем найти самую высокую из всех минимальных дневных температур:

```
SELECT max(temp_lo) FROM weather;
```

```
 max
-----
  46
(1 row)
```

Если мы хотим узнать, в каком городе (или городах) наблюдалась эта температура, можно попробовать:

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     НЕВЕРНО
```

но это не будет работать, так как агрегатную функцию `max` нельзя использовать в предложении `WHERE`. (Это ограничение объясняется тем, что предложение `WHERE` должно определить, для каких строк вычислять агрегатную функцию, так что оно, очевидно, должно вычисляться до агрегатных функций.) Однако как часто бывает, запрос можно переписать и получить желаемый результат, применив *подзапрос*:

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

```
     city
---------------
 San Francisco
(1 row)
```

Теперь всё в порядке — подзапрос выполняется отдельно и результат агрегатной функции вычисляется вне зависимости от того, что происходит во внешнем запросе.

Агрегатные функции также очень полезны в сочетании с предложением `GROUP BY`. Например, мы можем получить количество замеров и максимум минимальной дневной температуры в разрезе городов:

```
SELECT city, count(*), max(temp_lo)
    FROM weather
    GROUP BY city;
```

```
     city      | count | max
---------------+-------+-----
 Hayward       |     1 |  37
```

```
 San Francisco |     2 |  46
(2 rows)
```

Здесь мы получаем по одной строке для каждого города. Каждый агрегатный результат вычисляется по строкам таблицы, соответствующим отдельному городу. Мы можем отфильтровать сгруппированные строки с помощью предложения `HAVING`:

```
SELECT city, count(*), max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

```
  city   | count | max
---------+-------+-----
 Hayward |     1 |  37
(1 row)
```

Мы получаем те же результаты, но только для тех городов, где все значения `temp_lo` меньше 40. Наконец, если нас интересуют только города, названия которых начинаются с «`S`», мы можем сделать:

```
SELECT city, count(*), max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'            -- 1
    GROUP BY city;
```

1 Оператор `LIKE` (выполняющий сравнение по шаблону) рассматривается в Разделе 9.7.

Важно понимать, как соотносятся агрегатные функции и SQL-предложения `WHERE` и `HAVING`. Основное отличие `WHERE` от `HAVING` заключается в том, что `WHERE` сначала выбирает строки, а затем группирует их и вычисляет агрегатные функции (таким образом, она отбирает строки для вычисления агрегатов), тогда как `HAVING` отбирает строки групп после группировки и вычисления агрегатных функций. Как следствие, предложение `WHERE` не должно содержать агрегатных функций; не имеет смысла использовать агрегатные функции для определения строк для вычисления агрегатных функций. Предложение `HAVING`, напротив, всегда содержит агрегатные функции. (Строго говоря, вы можете написать предложение `HAVING`, не используя агрегаты, но это редко бывает полезно. То же самое условие может работать более эффективно на стадии `WHERE`.)

В предыдущем примере мы смогли применить фильтр по названию города в предложении `WHERE`, так как названия не нужно агрегировать. Такой фильтр эффективнее, чем дополнительное ограничение `HAVING`, потому что с ним не приходится группировать и вычислять агрегаты для всех строк, не удовлетворяющих условию `WHERE`.

Ещё один способ выбрать строки, которые входят в составные вычисления, — это использовать предложение `FILTER`, которое указывается для каждой агрегатной функции:

```
SELECT city, count(*) FILTER (WHERE temp_lo < 45), max(temp_lo)
    FROM weather
    GROUP BY city;
```

```
     city      | count | max
---------------+-------+-----
 Hayward       |     1 |  37
 San Francisco |     1 |  46
(2 rows)
```

Предложение `FILTER` очень похоже на `WHERE`, за исключением того, что отбрасываются входные строки только конкретной агрегатной функции, с которой оно используется. Здесь агрегатная функция `count` подсчитывает только строки с `temp_lo` ниже 45; но агрегатная функция `max` по-прежнему применяется ко всем строкам, поэтому находит значение 46.

2.8. Изменение данных

Данные в существующих строках можно изменять, используя команду `UPDATE`. Например, предположим, что вы обнаружили, что все значения температуры после 28 ноября завышены на два градуса. Вы можете поправить ваши данные следующим образом:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Посмотрите на новое состояние данных:

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. Удаление данных

Строки также можно удалить из таблицы, используя команду `DELETE`. Предположим, что вас больше не интересует погода в Хейуорде. В этом случае вы можете удалить ненужные строки из таблицы:

```
DELETE FROM weather WHERE city = 'Hayward';
```

Записи всех наблюдений, относящиеся к Хейуорду, удалены.

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

Остерегайтесь операторов вида

```
DELETE FROM имя_таблицы;
```

Без указания условия `DELETE` удалит *все* строки данной таблицы, полностью очистит её. При этом система не попросит вас подтвердить операцию!

Глава 3. Расширенные возможности

3.1. Введение

В предыдущей главе мы изучили азы использования SQL для хранения и обработки данных в Postgres Pro. Теперь мы обсудим более сложные возможности SQL, помогающие управлять данными и предотвратить их потерю или порчу. В конце главы мы рассмотрим некоторые расширения Postgres Pro.

В этой главе мы будем время от времени ссылаться на примеры, приведённые в Главе 2 и изменять или развивать их, поэтому будет полезно сначала прочитать предыдущую главу. Некоторые примеры этой главы также можно найти в файле `advanced.sql` в каталоге tutorial. Кроме того, этот файл содержит пример данных для загрузки (здесь она повторно не рассматривается). Если вы не знаете, как использовать этот файл, обратитесь к Разделу 2.1.

3.2. Представления

Вспомните запросы, с которыми мы имели дело в Разделе 2.6. Предположим, что вас интересует составной список из погодных записей и координат городов, но вы не хотите каждый раз вводить весь этот запрос. Вы можете создать *представление* по данному запросу, фактически присвоить имя запросу, а затем обращаться к нему как к обычной таблице:

```
CREATE VIEW myview AS
    SELECT name, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

Активное использование представлений — это ключевой аспект хорошего проектирования баз данных SQL. Представления позволяют вам скрыть внутреннее устройство ваших таблиц, которые могут меняться по мере развития приложения, за надёжными интерфейсами.

Представления можно использовать практически везде, где можно использовать обычные таблицы. И довольно часто представления создаются на базе других представлений.

3.3. Внешние ключи

Вспомните таблицы `weather` и `cities` из Главы 2. Давайте рассмотрим следующую задачу: вы хотите добиться, чтобы никто не мог вставить в таблицу `weather` строки, для которых не находится соответствующая строка в таблице `cities`. Это называется обеспечением *ссылочной целостности* данных. В простых СУБД это пришлось бы реализовать (если это вообще возможно) так: сначала явно проверить, есть ли соответствующие записи в таблице `cities`, а затем отклонить или вставить новые записи в таблицу `weather`. Этот подход очень проблематичен и неудобен, поэтому всё это Postgres Pro может сделать за вас.

Новое объявление таблицы будет выглядеть так:

```
CREATE TABLE cities (
        name     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(name),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
```

```
        date      date
);
```

Теперь попробуйте вставить недопустимую запись:

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');
```

```
ОШИБКА:  INSERT или UPDATE в таблице "weather" нарушает ограничение внешнего ключа
 "weather_city_fkey"
ПОДРОБНОСТИ:  Ключ (city)=(Berkeley) отсутствует в таблице "cities".
```

Поведение внешних ключей можно подстроить согласно требованиям вашего приложения. Мы не будем усложнять этот простой пример в данном введении, но вы можете обратиться за дополнительной информацией к Главе 5. Правильно применяя внешние ключи, вы определённо создадите более качественные приложения, поэтому мы настоятельно рекомендуем изучить их.

3.4. Транзакции

Транзакции — это фундаментальное понятие во всех СУБД. Суть транзакции в том, что она объединяет последовательность действий в одну операцию «всё или ничего». Промежуточные состояния внутри последовательности не видны другим транзакциям, и если что-то помешает успешно завершить транзакцию, ни один из результатов этих действий не сохранится в базе данных.

Например, рассмотрим базу данных банка, в которой содержится информация о счетах клиентов, а также общие суммы по отделениям банка. Предположим, что мы хотим перевести 100 долларов со счёта Алисы на счёт Боба. Простоты ради, соответствующие SQL-команды можно записать так:

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

Точное содержание команд здесь не важно, важно лишь то, что для выполнения этой довольно простой операции потребовалось несколько отдельных действий. При этом с точки зрения банка необходимо, чтобы все эти действия выполнились вместе, либо не выполнились совсем. Если Боб получит 100 долларов, но они не будут списаны со счёта Алисы, объяснить это сбоем системы определённо не удастся. И наоборот, Алиса вряд ли будет довольна, если она переведёт деньги, а до Боба они не дойдут. Нам нужна гарантия, что если что-то помешает выполнить операцию до конца, ни одно из действий не оставит следа в базе данных. И мы получаем эту гарантию, объединяя действия в одну *транзакцию*. Говорят, что транзакция *атомарна*: с точки зрения других транзакций она либо выполняется и фиксируется полностью, либо не фиксируется совсем.

Нам также нужна гарантия, что после завершения и подтверждения транзакции системой баз данных, её результаты в самом деле сохраняются и не будут потеряны, даже если вскоре произойдёт авария. Например, если мы списали сумму и выдали её Бобу, мы должны исключить возможность того, что сумма на его счёте восстановится, как только он выйдет за двери банка. Транзакционная база данных гарантирует, что все изменения записываются в постоянное хранилище (например, на диск) до того, как транзакция будет считаться завершённой.

Другая важная характеристика транзакционных баз данных тесно связана с атомарностью изменений: когда одновременно выполняется множество транзакций, каждая из них не видит незавершённые изменения, произведённые другими. Например, если одна транзакция подсчитывает баланс по отделениям, будет неправильно, если она посчитает расход в отделении Алисы, но не учтёт приход в отделении Боба, или наоборот. Поэтому свойство транзакций «всё или ничего» должно определять не только, как изменения сохраняются в базе данных, но и как они видны в процессе работы. Изменения, производимые открытой транзакцией, невидимы для других транзакций, пока она не будет завершена, а затем они становятся видны все сразу.

В Postgres Pro транзакция определяется набором SQL-команд, окружённым командами `BEGIN` и `COMMIT`. Таким образом, наша банковская транзакция должна была бы выглядеть так:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- ...
COMMIT;
```

Если в процессе выполнения транзакции мы решим, что не хотим фиксировать её изменения (например, потому что оказалось, что баланс Алисы стал отрицательным), мы можем выполнить команду `ROLLBACK` вместо `COMMIT`, и все наши изменения будут отменены.

Postgres Pro на самом деле отрабатывает каждый SQL-оператор как транзакцию. Если вы не вставите команду `BEGIN`, то каждый отдельный оператор будет неявно окружён командами `BEGIN` и `COMMIT` (в случае успешного завершения). Группу операторов, окружённых командами `BEGIN` и `COMMIT` иногда называют *блоком транзакции*.

Примечание

Некоторые клиентские библиотеки добавляют команды `BEGIN` и `COMMIT` автоматически и неявно создают за вас блоки транзакций. Подробнее об этом вы можете узнать в документации интересующего вас интерфейса.

Операторами в транзакции можно также управлять на более детальном уровне, используя *точки сохранения*. Точки сохранения позволяют выборочно отменять некоторые части транзакции и фиксировать все остальные. Определив точку сохранения с помощью `SAVEPOINT`, при необходимости вы можете вернуться к ней с помощью команды `ROLLBACK TO`. Все изменения в базе данных, произошедшие после точки сохранения и до момента отката, отменяются, но изменения, произведённые ранее, сохраняются.

Когда вы возвращаетесь к точке сохранения, она продолжает существовать, так что вы можете откатываться к ней несколько раз. С другой стороны, если вы уверены, что вам не придётся откатываться к определённой точке сохранения, её можно удалить, чтобы система высвободила ресурсы. Помните, что при удалении или откате к точке сохранения все точки сохранения, определённые после неё, автоматически уничтожаются.

Всё это происходит в блоке транзакции, так что в других сеансах работы с базой данных этого не видно. Совершённые действия становятся видны для других сеансов все сразу, только когда вы фиксируете транзакцию, а отменённые действия не видны вообще никогда.

Вернувшись к банковской базе данных, предположим, что мы списываем 100 долларов со счёта Алисы, добавляем их на счёт Боба, и вдруг оказывается, что деньги нужно было перевести Уолли. В данном случае мы можем применить точки сохранения:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- ошибочное действие... забыть его и использовать счёт Уолли
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

Этот пример, конечно, несколько надуман, но он показывает, как можно управлять выполнением команд в блоке транзакций, используя точки сохранения. Более того, `ROLLBACK TO` — это единственный способ вернуть контроль над блоком транзакций, оказавшимся в прерванном состоянии из-за ошибки системы, не считая возможности полностью отменить её и начать снова.

3.5. Оконные функции

Оконная функция выполняет вычисления для набора строк, некоторым образом связанных с текущей строкой. Её действие можно сравнить с вычислением, производимым агрегатной функцией. Однако с оконными функциями строки не группируются в одну выходную строку, что имеет место с обычными, не оконными, агрегатными функциями. Вместо этого, эти строки остаются отдельными сущностями. Внутри же, оконная функция, как и агрегатная, может обращаться не только к текущей строке результата запроса.

Вот пример, показывающий, как сравнить зарплату каждого сотрудника со средней зарплатой его отдела:

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;
```

```
  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

Первые три столбца извлекаются непосредственно из таблицы `empsalary`, при этом для каждой строки таблицы есть строка результата. В четвёртом столбце оказалось среднее значение, вычисленное по всем строкам, имеющим то же значение `depname`, что и текущая строка. (Фактически среднее вычисляет та же обычная, не оконная функция `avg`, но предложение `OVER` превращает её в оконную, так что её действие ограничивается рамками окон.)

Вызов оконной функции всегда содержит предложение `OVER`, следующее за названием и аргументами оконной функции. Это синтаксически отличает её от обычной, не оконной агрегатной функции. Предложение `OVER` определяет, как именно нужно разделить строки запроса для обработки оконной функцией. Предложение `PARTITION BY`, дополняющее `OVER`, разделяет строки по группам, или разделам, объединяя одинаковые значения выражений `PARTITION BY`. Оконная функция вычисляется по строкам, попадающим в один раздел с текущей строкой.

Вы можете также определять порядок, в котором строки будут обрабатываться оконными функциями, используя `ORDER BY` в `OVER`. (Порядок `ORDER BY` для окна может даже не совпадать с порядком, в котором выводятся строки.) Например:

```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;
```

```
  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
```

```
 personnel |      5 |    3500 |     2
 sales     |      1 |    5000 |     1
 sales     |      4 |    4800 |     2
 sales     |      3 |    4800 |     2
(10 rows)
```

Как показано здесь, функция `rank` выдаёт порядковый номер для каждого уникального значения в разделе текущей строки, по которому выполняет сортировку предложение `ORDER BY`. У функции `rank` нет параметров, так как её поведение полностью определяется предложением `OVER`.

Строки, обрабатываемые оконной функцией, представляют собой «виртуальные таблицы», созданные из предложения `FROM` и затем прошедшие через фильтрацию и группировку `WHERE` и `GROUP BY` и, возможно, условие `HAVING`. Например, строка, отфильтрованная из-за нарушения условия `WHERE`, не будет видна для оконных функций. Запрос может содержать несколько оконных функций, разделяющих данные по-разному с применением разных предложений `OVER`, но все они будут обрабатывать один и тот же набор строк этой виртуальной таблицы.

Мы уже видели, что `ORDER BY` можно опустить, если порядок строк не важен. Также возможно опустить `PARTITION BY`, в этом случае образуется один раздел, содержащий все строки.

Есть ещё одно важное понятие, связанное с оконными функциями: для каждой строки существует набор строк в её разделе, называемый *рамкой окна*. Некоторые оконные функции обрабатывают только строки рамки окна, а не всего раздела. По умолчанию с указанием `ORDER BY` рамка состоит из всех строк от начала раздела до текущей строки и строк, равных текущей по значению выражения `ORDER BY`. Без `ORDER BY` рамка по умолчанию состоит из всех строк раздела. [1] Посмотрите на пример использования `sum`:

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

```
 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

Так как в этом примере нет указания `ORDER BY` в предложении `OVER`, рамка окна содержит все строки раздела, а он, в свою очередь, без предложения `PARTITION BY` включает все строки таблицы; другими словами, сумма вычисляется по всей таблице и мы получаем один результат для каждой строки результата. Но если мы добавим `ORDER BY`, мы получим совсем другие результаты:

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

```
 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
```

---

[1] Рамки окна можно определять и другими способами, но в этом введении они не рассматриваются. Узнать о них подробнее вы можете в Подразделе 4.2.8.

```
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

Здесь в сумме накапливаются зарплаты от первой (самой низкой) до текущей, включая повторяющиеся текущие значения (обратите внимание на результат в строках с одинаковой зарплатой).

Оконные функции разрешается использовать в запросе только в списке `SELECT` и предложении `ORDER BY`. Во всех остальных предложениях, включая `GROUP BY`, `HAVING` и `WHERE`, они запрещены. Это объясняется тем, что логически они выполняются после этих предложений, а также после не оконных агрегатных функций, и значит агрегатную функцию можно вызывать в аргументах оконной, но не наоборот.

Если вам нужно отфильтровать или сгруппировать строки после вычисления оконных функций, вы можете использовать вложенный запрос. Например:

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
    rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
   FROM empsalary
  ) AS ss
WHERE pos < 3;
```

Данный запрос покажет только те строки внутреннего запроса, у которых `rank` (порядковый номер) меньше 3.

Когда в запросе вычисляются несколько оконных функций для одинаково определённых окон, конечно можно написать для каждой из них отдельное предложение `OVER`, но при этом оно будет дублироваться, что неизбежно будет провоцировать ошибки. Поэтому лучше определение окна выделить в предложение `WINDOW`, а затем ссылаться на него в `OVER`. Например:

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

Подробнее об оконных функциях можно узнать в Подразделе 4.2.8, Разделе 9.22, Подразделе 7.2.5 и в справке SELECT.

3.6. Наследование

Наследование — это концепция, взятая из объектно-ориентированных баз данных. Она открывает множество интересных возможностей при проектировании баз данных.

Давайте создадим две таблицы: `cities` (города) и `capitals` (столицы штатов). Естественно, столицы штатов также являются городами, поэтому нам нужно явным образом добавлять их в результат, когда мы хотим просмотреть все города. Если вы проявите смекалку, вы можете предложить, например, такое решение:

```
CREATE TABLE capitals (
  name       text,
  population real,
  elevation  int,    -- (высота в футах)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
```

```
  elevation  int     -- (высота в футах)
);

CREATE VIEW cities AS
  SELECT name, population, elevation FROM capitals
    UNION
  SELECT name, population, elevation FROM non_capitals;
```

Оно может устраивать, пока мы извлекаем данные, но если нам потребуется изменить несколько строк, это будет выглядеть некрасиво.

Поэтому есть лучшее решение:

```
CREATE TABLE cities (
  name       text,
  population real,
  elevation  int     -- (высота в футах)
);

CREATE TABLE capitals (
  state      char(2) UNIQUE NOT NULL
) INHERITS (cities);
```

В данном случае строка таблицы `capitals` *наследует* все столбцы (`name`, `population` и `elevation`) от *родительской таблицы* `cities`. Столбец `name` имеет тип `text`, собственный тип Postgres Pro для текстовых строк переменной длины. А в таблицу `capitals` добавлен дополнительный столбец `state`, в котором будет указан буквенный код штата. В Postgres Pro таблица может наследоваться от ноля или нескольких других таблиц.

Например, следующий запрос выведет названия всех городов, включая столицы, находящихся выше 500 футов над уровнем моря:

```
SELECT name, elevation
  FROM cities
  WHERE elevation > 500;
```

Результат его выполнения:

```
   name    | elevation
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

А следующий запрос находит все города, которые не являются столицами штатов, но также находятся выше 500 футов:

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;
```

```
   name    | elevation
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Здесь слово `ONLY` перед названием таблицы `cities` указывает, что запрос следует выполнять только для строк таблицы `cities`, не включая таблицы, унаследованные от `cities`. Многие операторы, которые мы уже обсудили — `SELECT`, `UPDATE` и `DELETE` — поддерживают указание `ONLY`.

Примечание

Хотя наследование часто бывает полезно, оно не интегрировано с ограничениями уникальности и внешними ключами, что ограничивает его применимость. Подробнее это описывается в Разделе 5.11.

3.7. Заключение

Postgres Pro имеет множество возможностей, не затронутых в этом кратком введении, рассчитанном на начинающих пользователей SQL. Эти возможности будут рассмотрены в деталях в продолжении книги.

Если вам необходима дополнительная вводная информация, посетите *сайт PostgreSQL*, там вы найдёте ссылки на другие ресурсы.

Часть II. Язык SQL

В этой части книги описывается использование языка SQL в Postgres Pro. Мы начнём с описания общего синтаксиса SQL, затем расскажем, как создавать таблицы, наполнять базу данных и выполнять запросы к ней. В продолжении обратим внимание на существующие типы данных и функции, применяемые с командами SQL. И наконец, рассмотрим важные аспекты настройки базы данных.

Материал упорядочен так, чтобы новичок мог прочитать его от начала до конца и полностью понять все темы, не забегая вперёд. При этом главы сделаны самодостаточными, так что опытные пользователи могут читать их по отдельности. Информация представлена в повествовательном стиле и разделена по темам. Если же вас интересует формальное и полное описание определённой команды, рекомендуем обратиться к Части VI.

Читатели должны уже знать, как подключаться к базе данных Postgres Pro и выполнять команды SQL. Если вы ещё не знаете этого, рекомендуется сначала прочитать Часть I. Команды SQL обычно вводятся в psql — интерактивном терминальном приложении Postgres Pro, но можно воспользоваться и другими программами с подобными функциями.

Глава 4. Синтаксис SQL

В этой главе описывается синтаксис языка SQL. Тем самым закладывается фундамент для следующих глав, где будет подробно рассмотрено, как с помощью команд SQL описывать и изменять данные.

Мы советуем прочитать эту главу и тем, кто уже знаком SQL, так как в ней описываются несколько правил и концепций, которые реализованы в разных базах данных SQL по-разному или относятся только к Postgres Pro.

4.1. Лексическая структура

SQL-программа состоит из последовательности *команд*. Команда, в свою очередь, представляет собой последовательность *компонентов*, оканчивающуюся точкой с запятой («;»). Конец входного потока также считается концом команды. Какие именно компоненты допустимы для конкретной команды, зависит от её синтаксиса.

Компонентом команды может быть *ключевое слово*, *идентификатор*, *идентификатор в кавычках*, *строка* (или константа) или специальный символ. Компоненты обычно разделяются пробельными символами (пробел, табуляция, перевод строки), но это не требуется, если нет неоднозначности (например, когда спецсимвол оказывается рядом с компонентом другого типа).

Например, следующий текст является правильной (синтаксически) SQL-программой:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

Это последовательность трёх команд, по одной в строке (хотя их можно было разместить и в одну строку или наоборот, разделить команды на несколько строк).

Кроме этого, SQL-программы могут содержать *комментарии*. Они не являются компонентами команд, а по сути равносильны пробельным символам.

Синтаксис SQL не очень строго определяет, какие компоненты идентифицируют команды, а какие — их операнды или параметры. Первые несколько компонентов обычно содержат имя команды, так что в данном примере мы можем говорить о командах «SELECT», «UPDATE» и «INSERT». Но например, команда `UPDATE` требует, чтобы также в определённом положении всегда стоял компонент `SET`, а `INSERT` в приведённом виде требует наличия компонента `VALUES`. Точные синтаксические правила для каждой команды описаны в Части VI.

4.1.1. Идентификаторы и ключевые слова

Показанные выше команды содержали компоненты `SELECT`, `UPDATE` и `VALUES`, которые являются примерами *ключевых слов*, то есть слов, имеющих фиксированное значение в языке SQL. Компоненты `MY_TABLE` и `A` являются примерами *идентификаторов*. Они идентифицируют имена таблиц, столбцов или других объектов баз данных, в зависимости от того, где они используются. Поэтому иногда их называют просто «именами». Ключевые слова и идентификаторы имеют одинаковую лексическую структуру, то есть, не зная языка, нельзя определить, является ли некоторый компонент ключевым словом или идентификатором. Полный список ключевых слов приведён в Приложении C.

Идентификаторы и ключевые слова SQL должны начинаться с буквы (`a`-`z`, хотя допускаются также не латинские буквы и буквы с диакритическими знаками) или подчёркивания (_). Последующими символами в идентификаторе или ключевом слове могут быть буквы, цифры (`0`-`9`), знаки доллара (`$`) или подчёркивания. Заметьте, что строго следуя букве стандарта SQL, знаки доллара нельзя использовать в идентификаторах, так что их использование вредит переносимости приложений. В стандарте SQL гарантированно не будет ключевых слов с цифрами и начинающихся или заканчивающихся подчёркиванием, так что идентификаторы такого вида защищены от возможных конфликтов с будущими расширениями стандарта.

Система выделяет для идентификатора не более NAMEDATALEN-1 байт, а более длинные имена усекаются. По умолчанию `NAMEDATALEN` равно 64, так что максимальная длина идентификатора равна 63 байтам.

Ключевые слова и идентификаторы без кавычек воспринимаются системой без учёта регистра. Таким образом:

```
UPDATE MY_TABLE SET A = 5;
```

равносильно записи:

```
uPDaTE my_TabLE SeT a = 5;
```

Часто используется неформальное соглашение записывать ключевые слова заглавными буквами, а имена строчными, например:

```
UPDATE my_table SET a = 5;
```

Есть и другой тип идентификаторов: *отделённые идентификаторы* или *идентификаторы в кавычках*. Они образуются при заключении обычного набора символов в двойные кавычки ("). Такие идентификаторы всегда будут считаться идентификаторами, но не ключевыми словами. Так `"select"` можно использовать для обозначения столбца или таблицы «select», тогда как `select` без кавычек будет воспринят как ключевое слово и приведёт к ошибке разбора команды в месте, где ожидается имя таблицы или столбца. Тот же пример можно переписать с идентификаторами в кавычках следующим образом:

```
UPDATE "my_table" SET "a" = 5;
```

Идентификаторы в кавычках могут содержать любые символы, за исключением символа с кодом 0. (Чтобы включить в такой идентификатор кавычки, продублируйте их.) Это позволяет создавать таблицы и столбцы с именами, которые иначе были бы невозможны, например, с пробелами или амперсандами. Ограничение длины при этом сохраняется.

Идентификатор, заключённый в кавычки, становится зависимым от регистра, тогда как идентификаторы без кавычек всегда переводятся в нижний регистр. Например, идентификаторы `FOO`, `foo` и `"foo"` считаются одинаковыми в Postgres Pro, но `"Foo"` и `"FOO"` отличны друг от друга и от предыдущих трёх. (Приведение имён без кавычек к нижнему регистру, как это делает Postgres Pro, несовместимо со стандартом SQL, который говорит о том, что имена должны приводиться к верхнему регистру. То есть, согласно стандарту `foo` должно быть эквивалентно `"FOO"`, а не `"foo"`. Поэтому при создании переносимых приложений рекомендуется либо всегда заключать определённое имя в кавычки, либо не заключать никогда.)

Ещё один вариант идентификаторов в кавычках позволяет использовать символы Unicode по их кодам. Такой идентификатор начинается с `U&` (строчная или заглавная U и амперсанд), а затем сразу без пробелов идёт двойная кавычка, например `U&"foo"`. (Заметьте, что при этом возникает неоднозначность с оператором `&`. Чтобы её избежать, окружайте этот оператор пробелами.) Затем в кавычках можно записывать символы Unicode двумя способами: обратная косая черта, а за ней код символа из четырёх шестнадцатеричных цифр, либо обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр. Например, идентификатор `"data"` можно записать так:

```
U&"d\0061t\+000061"
```

В следующем менее тривиальном примере закодировано русское слово «слон», записанное кириллицей:

```
U&"\0441\043B\043E\043D"
```

Если вы хотите использовать не обратную косую черту, а другой спецсимвол, его можно указать, добавив `UESCAPE` после строки, например:

```
U&"d!0061t!+000061" UESCAPE '!'
```

В качестве спецсимвола можно выбрать любой символ, кроме шестнадцатеричной цифры, знака плюс, апострофа, кавычки или пробельного символа. Заметьте, что спецсимвол заключается не в двойные кавычки, а в апострофы, после `UESCAPE`.

Чтобы сделать спецсимволом знак апострофа, напишите его дважды.

Записывать суррогатные пары UTF-16 и таким образом составлять символы с кодами больше чем U+FFFF можно либо в четырёх-, либо в шестизначной форме, хотя наличие шестизначной формы технически делает это ненужным. (Суррогатные пары не сохраняются непосредственно, а объединяются в один символ, который затем кодируется в UTF-8.)

Когда кодировка сервера — не UTF-8, символ с кодом, указанным этой спецпоследовательностью, преобразуется в фактическую кодировку сервера; если такое преобразование невозможно, выдаётся ошибка.

4.1.2. Константы

В Postgres Pro есть три типа констант *подразумеваемых типов*: строки, битовые строки и числа. Константы можно также записывать, указывая типы явно, что позволяет представить их более точно и обработать более эффективно. Эти варианты рассматриваются в следующих подразделах.

4.1.2.1. Строковые константы

Строковая константа в SQL — это обычная последовательность символов, заключённая в апострофы (`'`), например: `'Это строка'`. Чтобы включить апостроф в строку, напишите в ней два апострофа рядом, например: `'Жанна д''Арк'`. Заметьте, это *не* то же самое, что двойная кавычка (`"`).

Две строковые константы, разделённые пробельными символами *и минимум одним переводом строки*, объединяются в одну и обрабатываются, как если бы строка была записана в одной константе. Например:

```
SELECT 'foo'
'bar';
```

эквивалентно:

```
SELECT 'foobar';
```

но эта запись:

```
SELECT 'foo'      'bar';
```

считается синтаксической ошибкой. (Это несколько странное поведение определено в стандарте SQL, Postgres Pro просто следует ему.)

4.1.2.2. Строковые константы со спецпоследовательностями в стиле C

Postgres Pro также принимает «спецпоследовательности», что является расширением стандарта SQL. Строка со спецпоследовательностями начинается с буквы `E` (заглавной или строчной), стоящей непосредственно перед апострофом, например: `E'foo'`. (Когда константа со спецпоследовательностью разбивается на несколько строк, букву `E` нужно поставить только перед первым открывающим апострофом.) Внутри таких строк символ обратной косой черты (`\`) начинает C-подобные *спецпоследовательности*, в которых сочетание обратной косой черты со следующим символом(ами) даёт определённое байтовое значение, как показано в Таблице 4.1.

Таблица 4.1. Спецпоследовательности

Спецпоследовательность	**Интерпретация**
`\b`	символ «забой»
`\f`	подача формы
`\n`	новая строка
`\r`	возврат каретки
`\t`	табуляция
`\o`, `\oo`, `\ooo` (*o* = 0–7)	восьмеричное значение байта

Спецпоследовательность	**Интерпретация**
`\x`*h*, `\x`*hh* (*h* = 0–9, A–F)	шестнадцатеричное значение байта
`\u`*xxxx*, `\U`*xxxxxxxx* (*x* = 0–9, A–F)	16- или 32-битный шестнадцатеричный код символа Unicode

Любой другой символ, идущий после обратной косой черты, воспринимается буквально. Таким образом, чтобы включить в строку обратную косую черту, нужно написать две косых черты (`\\`). Так же можно включить в строку апостроф, написав `\'`, в дополнение к обычному способу `''`.

Вы должны позаботиться, чтобы байтовые последовательности, которые вы создаёте таким образом, особенно в восьмеричной и шестнадцатеричной записи, образовывали допустимые символы в серверной кодировке. Также может быть полезно использовать спецпоследовательности Unicode или альтернативную запись, описанную в Подразделе 4.1.2.3; в этом случае сервер будет проверять, возможно ли преобразовать указанный символ.

Внимание

Если параметр конфигурации standard_conforming_strings имеет значение `off`, Postgres Pro распознаёт обратную косую черту как спецсимвол и в обычных строках, и в строках со спецпоследовательностями. Однако в версии PostgreSQL 9.1 по умолчанию принято значение `on`, и в этом случае обратная косая черта распознаётся только в спецстроках. Это поведение больше соответствует стандарту, хотя может нарушить работу приложений, рассчитанных на предыдущий режим, когда обратная косая черта распознавалась везде. В качестве временного решения вы можете изменить этот параметр на `off`, но лучше уйти от такой практики. Если вам нужно, чтобы обратная косая черта представляла специальный символ, задайте строковую константу с `E`.

В дополнение к `standard_conforming_strings` поведением обратной косой черты в строковых константах управляют параметры escape_string_warning и backslash_quote.

Строковая константа не может включать символ с кодом 0.

4.1.2.3. Строковые константы со спецпоследовательностями Unicode

Postgres Pro также поддерживает ещё один вариант спецпоследовательностей, позволяющий включать в строки символы Unicode по их кодам. Строковая константа со спецпоследовательностями Unicode начинается с `U&` (строчная или заглавная U и амперсанд), а затем сразу без пробелов идёт апостроф, например `U&'foo'`. (Заметьте, что при этом возникает неоднозначность с оператором `&`. Чтобы её избежать, окружайте этот оператор пробелами.) Затем в апострофах можно записывать символы Unicode двумя способами: обратная косая черта, а за ней код символа из четырёх шестнадцатеричных цифр, либо обратная косая черта, знак плюс, а затем код из шести шестнадцатеричных цифр. Например, строку `'data'` можно записать так:

```
U&'d\0061t\+000061'
```

В следующем менее тривиальном примере закодировано русское слово «слон», записанное кириллицей:

```
U&'\0441\043B\043E\043D'
```

Если вы хотите использовать не обратную косую черту, а другой спецсимвол, его можно указать, добавив `UESCAPE`после строки, например:

```
U&'d!0061t!+000061' UESCAPE '!'
```

В качестве спецсимвола можно выбрать любой символ, кроме шестнадцатеричной цифры, знака плюс, апострофа, кавычки или пробельного символа.

Чтобы включить спецсимвол в строку буквально, напишите его дважды.

Записывать суррогатные пары UTF-16 и таким образом составлять символы с кодами больше чем U+FFFF можно либо в четырёх-, либо в шестизначной форме, хотя наличие шестизначной формы технически делает это ненужным. (Суррогатные пары не сохраняются непосредственно, а объединяются в один символ, который затем кодируется в UTF-8.)

Когда кодировка сервера — не UTF-8, символ с кодом, указанным этой спецпоследовательностью, преобразуется в фактическую кодировку сервера; если такое преобразование невозможно, выдаётся ошибка.

Также заметьте, что спецпоследовательности Unicode в строковых константах работают, только когда параметр конфигурации standard_conforming_strings равен `on`. Это объясняется тем, что иначе клиентские программы, проверяющие SQL-операторы, можно будет ввести в заблуждение и эксплуатировать это как уязвимость, например, для SQL-инъекций. Если этот параметр имеет значение `off`, эти спецпоследовательности будут вызывать ошибку.

4.1.2.4. Строковые константы, заключённые в доллары

Хотя стандартный синтаксис для строковых констант обычно достаточно удобен, он может плохо читаться, когда строка содержит много апострофов, так как каждый такой символ приходится дублировать. Чтобы и в таких случаях запросы оставались читаемыми, Postgres Pro предлагает ещё один способ записи строковых констант — «заключение строк в доллары». Строковая константа, заключённая в доллары, начинается со знака доллара (`$`), необязательного «тега» из нескольких символов и ещё одного знака доллара, затем содержит обычную последовательность символов, составляющую строку, и оканчивается знаком доллара, тем же тегом и замыкающим знаком доллара. Например, строку «Жанна д'Арк» можно записать в долларах двумя способами:

```
$$Жанна д'Арк$$
$SomeTag$Жанна д'Арк$SomeTag$
```

Заметьте, что внутри такой строки апострофы не нужно записывать особым образом. На самом деле, в строке, заключённой в доллары, все символы можно записывать в чистом виде: содержимое строки всегда записывается буквально. Ни обратная косая черта, ни даже знак доллара не являются спецсимволами, если только они не образуют последовательность, соответствующую открывающему тегу.

Строковые константы в долларах можно вкладывать друг в друга, выбирая на разных уровнях вложенности разные теги. Чаще всего это используется при написании определений функций. Например:

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Здесь последовательность `$q$[\t\r\n\v\\]$q$` представляет в долларах текстовую строку `[\t\r\n\v\\]`, которая будет обработана, когда Postgres Pro будет выполнять эту функцию. Но так как эта последовательность не соответствует внешнему тегу в долларах (`$function$`), с точки зрения внешней строки это просто обычные символы внутри константы.

Тег строки в долларах, если он присутствует, должен соответствовать правилам, определённым для идентификаторов без кавычек, и к тому же не должен содержать знак доллара. Теги регистрозависимы, так что `$tag$String content$tag$` — правильная строка, а `$TAG$String content$tag$` — нет.

Строка в долларах, следующая за ключевым словом или идентификатором, должна отделяться от него пробельными символами, иначе доллар будет считаться продолжением предыдущего идентификатора.

Заключение строк в доллары не является частью стандарта SQL, но часто это более удобный способ записывать сложные строки, чем стандартный вариант с апострофами. Он особенно полезен, когда нужно представить строковую константу внутри другой строки, что часто требуется в определениях процедурных функций. Ограничившись только апострофами, каждую обратную косую черту в приведённом примере пришлось бы записывать четырьмя такими символами, которые бы затем уменьшились до двух при разборе внешней строки, и наконец до одного при обработке внутренней строки во время выполнения функции.

4.1.2.5. Битовые строковые константы

Битовые строковые константы похожи на обычные с дополнительной буквой `B` (заглавной или строчной), добавленной непосредственно перед открывающим апострофом (без промежуточных пробелов), например: `B'1001'`. В битовых строковых константах допускаются лишь символы `0` и `1`.

Битовые константы могут быть записаны и по-другому, в шестнадцатеричном виде, с начальной буквой `X` (заглавной или строчной), например: `X'1FF'`. Такая запись эквивалентна двоичной, только четыре двоичных цифры заменяются одной шестнадцатеричной.

Обе формы записи допускают перенос строк так же, как и обычные строковые константы. Однако заключать в доллары битовые строки нельзя.

4.1.2.6. Числовые константы

Числовые константы могут быть заданы в следующем общем виде:

цифры
цифры`.`[*цифры*][`e`[`+-`]*цифры*]
[*цифры*]`.`*цифры*[`e`[`+-`]*цифры*]
цифры`e`[`+-`]*цифры*

где *цифры* — это одна или несколько десятичных цифр (0..9). До или после десятичной точки (при её наличии) должна быть минимум одна цифра. Как минимум одна цифра должна следовать за обозначением экспоненты (`e`), если оно присутствует. В числовой константе не может быть пробелов или других символов, за исключением символов подчёркивания, которые могут использоваться для визуальной группировки, как описано ниже. Обратите внимание на то, что любой знак минус или плюс в начале строки не считается частью числа — это оператор, применённый к константе.

Несколько примеров допустимых числовых констант:

42
3.5
4.
.001
5e2
1.925e-3

Кроме того, недесятичные целочисленные константы принимаются в следующих формах:

`0x`*hexdigits*
`0o`*octdigits*
`0b`*bindigits*

где *hexdigits* — одна или более шестнадцатеричных цифр (0-9, A-F), *octdigits* — одна или более восьмеричных цифр (0-7) и *bindigits* — одна или более двоичных цифр (0 или 1). Шестнадцатеричные цифры и префиксы могут указываться в верхнем или нижнем регистре. Обратите внимание на то, что недесятичные формы могут принимать не числа с дробной частью, а только целые числа.

Несколько примеров допустимых недесятичных целочисленных констант:

0b100101
0B10011001
0o273
0O755
0x42f
0XFFFF

Для визуальной группировки между цифрами можно вставлять символы подчёркивания. Они не влияют на значение константы. Например:

```
1_500_000_000
0b10001000_00000000
0o_1_755
0xFFFF_FFFF
1.618_034
```

Запрещается указывать символы подчёркивания в начале или в конце числовой константы или группы цифр (то есть сразу перед или после десятичной точки или обозначения экспоненты), а также более одного символа подчёркивания подряд.

Числовая константа, не содержащая точки и экспоненты, изначально рассматривается как константа типа `integer`, если её значение умещается в 32-битный тип `integer`; затем как константа типа `bigint`, если её значение умещается в 64-битный `bigint`; в противном случае она принимает тип `numeric`. Константы, содержащие десятичные точки и/или экспоненты, всегда считаются константами типа `numeric`.

Изначально назначенный тип данных числовой константы это только отправная точка для алгоритмов определения типа. В большинстве случаев константа будет автоматически приведена к наиболее подходящему типу для данного контекста. При необходимости вы можете принудительно интерпретировать числовое значение как значение определённого типа, приведя его тип к нужному. Например, вы можете сделать, чтобы числовое значение рассматривалось как имеющее тип `real` (`float4`), написав:

```
REAL '1.23'  -- строковый стиль
1.23::REAL   -- стиль Postgres Pro (исторический)
```

На самом деле это только частные случаи синтаксиса приведения типов, который будет рассматриваться далее.

4.1.2.7. Константы других типов

Константу *обычного* типа можно ввести одним из следующих способов:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

Текст строковой константы передаётся процедуре преобразования ввода для типа, обозначенного здесь *`type`*. Результатом становится константа указанного типа. Явное приведение типа можно опустить, если нужный тип константы определяется однозначно (например, когда она присваивается непосредственно столбцу таблицы), так как в этом случае приведение происходит автоматически.

Строковую константу можно записать, используя как обычный синтаксис SQL, так и формат с долларами.

Также можно записать приведение типов, используя синтаксис функций:

```
typename ( 'string' )
```

но это работает не для всех имён типов; подробнее об этом написано в Подразделе 4.2.9.

Конструкцию `::`, `CAST()` и синтаксис вызова функции можно также использовать для преобразования типов обычных выражений во время выполнения, как описано в Подразделе 4.2.9. Во избежание синтаксической неопределённости, запись `тип 'строка'` можно использовать только для указания типа простой текстовой константы. Ещё одно ограничение записи `тип 'строка'`: она не работает для массивов; для таких констант следует использовать `::` или `CAST()`.

Синтаксис `CAST()` соответствует SQL, а запись `type 'string'` является обобщением стандарта: в SQL такой синтаксис поддерживает только некоторые типы данных, но Postgres Pro позволяет использовать его для всех. Синтаксис с `::` имеет исторические корни в Postgres Pro, как и запись в виде вызова функции.

4.1.3. Операторы

Имя оператора образует последовательность не более чем `NAMEDATALEN`-1 (по умолчанию 63) символов из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Однако для имён операторов есть ещё несколько ограничений:

- Сочетания символов `--` и `/*` не могут присутствовать в имени оператора, так как они будут обозначать начало комментария.
- Многосимвольное имя оператора не может заканчиваться знаком `+` или `-`, если только оно не содержит также один из этих символов:

  ~ ! @ # % ^ & | ` ?

  Например, `@-` — допустимое имя оператора, а `*-` — нет. Благодаря этому ограничению, Postgres Pro может разбирать корректные SQL-запросы без пробелов между компонентами.

Записывая нестандартные SQL-операторы, обычно нужно отделять имена соседних операторов пробелами для однозначности. Например, если вы определили префиксный оператор с именем `@`, вы не можете написать `X*@Y`, а должны написать `X* @Y`, чтобы Postgres Pro однозначно прочитал это как два оператора, а не один.

4.1.4. Специальные знаки

Некоторые не алфавитно-цифровые символы имеют специальное значение, но при этом не являются операторами. Подробнее их использование будет рассмотрено при описании соответствующего элемента синтаксиса. Здесь они упоминаются только для сведения и обобщения их предназначения.

- Знак доллара (`$`), предваряющий число, используется для представления позиционного параметра в теле определения функции или подготовленного оператора. В других контекстах знак доллара может быть частью идентификатора или строковой константы, заключённой в доллары.
- Круглые скобки (`()`) имеют обычное значение и применяются для группировки выражений и повышения приоритета операций. В некоторых случаях скобки — это необходимая часть синтаксиса определённых SQL-команд.
- Квадратные скобки (`[]`) применяются для выделения элементов массива. Подробнее массивы рассматриваются в Разделе 8.15.
- Запятые (`,`) используются в некоторых синтаксических конструкциях для разделения элементов списка.
- Точка с запятой (`;`) завершает команду SQL. Она не может находиться нигде внутри команды, за исключением строковых констант или идентификаторов в кавычках.
- Двоеточие (`:`) применяется для выборки «срезов» массивов (см. Раздел 8.15.) В некоторых диалектах SQL (например, в Embedded SQL) двоеточие может быть префиксом в имени переменной.
- Звёздочка (`*`) используется в некоторых контекстах как обозначение всех полей строки или составного значения. Она также имеет специальное значение, когда используется как аргумент некоторых агрегатных функций, а именно функций, которым не нужны явные параметры.
- Точка (`.`) используется в числовых константах, а также для отделения имён схемы, таблицы и столбца.

4.1.5. Комментарии

Комментарий — это последовательность символов, которая начинается с двух минусов и продолжается до конца строки, например:

```
-- Это стандартный комментарий SQL
```

Кроме этого, блочные комментарии можно записывать в стиле C:

```
/* многострочный комментарий
 * с вложенностью: /* вложенный блок комментария */
 */
```

где комментарий начинается с `/*` и продолжается до соответствующего вхождения `*/`. Блочные комментарии можно вкладывать друг в друга, как разрешено по стандарту SQL (но не разрешено в C), так что вы можете комментировать большие блоки кода, которые при этом уже могут содержать блоки комментариев.

Комментарий удаляется из входного потока в начале синтаксического анализа и фактически заменяется пробелом.

4.1.6. Приоритеты операторов

В Таблице 4.2 показаны приоритеты и очерёдность операторов, действующие в Postgres Pro. Большинство операторов имеют одинаковый приоритет и вычисляются слева направо. Приоритет и очерёдность операторов жёстко фиксированы в синтаксическом анализаторе. Если вы хотите, чтобы выражение с несколькими операторами разбиралось не в том порядке, который диктуют эти приоритеты, добавьте скобки.

Таблица 4.2. Приоритет операторов (от большего к меньшему)

Оператор/элемент	Очерёдность	Описание
`.`	слева-направо	разделитель имён таблицы и столбца
`::`	слева-направо	приведение типов в стиле Postgres Pro
`[ ]`	слева-направо	выбор элемента массива
`+ -`	справа-налево	унарный плюс, унарный минус
`COLLATE`	слева-направо	выбор правила сортировки
`AT`	слева-направо	`AT TIME ZONE`, `AT LOCAL`
`^`	слева-направо	возведение в степень
`* / %`	слева-направо	умножение, деление, остаток от деления
`+ -`	слева-направо	сложение, вычитание
(любой другой оператор)	слева-направо	все другие встроенные и пользовательские операторы
`BETWEEN IN LIKE ILIKE SIMILAR`		проверка диапазона, проверка членства, сравнение строк
`< > = <= >= <>`		операторы сравнения
`IS ISNULL NOTNULL`		`IS TRUE`, `IS FALSE`, `IS NULL`, `IS DISTINCT FROM` и т. д.
`NOT`	справа-налево	логическое отрицание
`AND`	слева-направо	логическая конъюнкция
`OR`	слева-направо	логическая дизъюнкция

Заметьте, что правила приоритета операторов также применяются к операторам, определённым пользователем с теми же именами, что и вышеперечисленные встроенные операторы. Например, если вы определите оператор «+» для некоторого нестандартного типа данных, он будет иметь тот же приоритет, что и встроенный оператор «+», независимо от того, что он у вас делает.

Когда в конструкции `OPERATOR` используется имя оператора со схемой, например так:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

тогда `OPERATOR` имеет приоритет по умолчанию, соответствующий в Таблице 4.2 строке «любой другой оператор». Это не зависит от того, какие именно операторы находятся в конструкции `OPERATOR()`.

Примечание

В PostgreSQL до версии 9.5 действовали немного другие правила приоритета операторов. В частности, операторы `<=`, `>=` и `<>` обрабатывались по общему правилу; проверки `IS` имели более высокий приоритет; а `NOT BETWEEN` и связанные конструкции работали несогласованно — в некоторых случаях приоритетнее оказывался оператор `NOT`, а не `BETWEEN`. Эти правила были изменены для лучшего соответствия стандарту SQL и для уменьшения путаницы из-за несогласованной обработки логически равнозначных конструкций. В большинстве случаев эти изменения никак не проявятся, либо могут привести к ошибкам типа «нет такого оператора», которые можно разрешить, добавив скобки. Однако возможны особые случаи, когда запрос будет разобран без ошибки, но его поведение может измениться.

4.2. Выражения значения

Выражения значения применяются в самых разных контекстах, например в списке результатов команды `SELECT`, в значениях столбцов в `INSERT` или `UPDATE` или в условиях поиска во многих командах. Результат такого выражения иногда называют *скаляром*, чтобы отличить его от результата табличного выражения (который представляет собой таблицу). А сами выражения значения часто называют *скалярными* (или просто *выражениями*). Синтаксис таких выражений позволяет вычислять значения из примитивных частей, используя арифметические, логические и другие операции.

Выражениями значения являются:

- Константа или непосредственное значение
- Ссылка на столбец
- Ссылка на позиционный параметр в теле определения функции или подготовленного оператора
- Выражение с индексом
- Выражение выбора поля
- Применение оператора
- Вызов функции
- Агрегатное выражение
- Вызов оконной функции
- Приведение типов
- Применение правил сортировки
- Скалярный подзапрос
- Конструктор массива
- Конструктор табличной строки
- Кроме того, выражением значения являются скобки (предназначенные для группировки подвыражений и переопределения приоритета)

В дополнение к этому списку есть ещё несколько конструкций, которые можно классифицировать как выражения, хотя они не соответствуют общим синтаксическим правилам. Они обычно имеют вид функции или оператора и будут рассмотрены в соответствующем разделе Главы 9. Пример такой конструкции — предложение `IS NULL`.

Мы уже обсудили константы в Подразделе 4.1.2. В следующих разделах рассматриваются остальные варианты.

4.2.1. Ссылки на столбцы

Ссылку на столбец можно записать в форме:

```
отношение.имя_столбца
```

Здесь *`отношение`* — имя таблицы (возможно, полное, с именем схемы) или её псевдоним, определённый в предложении `FROM`. Это имя и разделяющую точку можно опустить, если имя столбца уникально среди всех таблиц, задействованных в текущем запросе. (См. также Главу 7.)

4.2.2. Позиционные параметры

Ссылка на позиционный параметр применяется для обращения к значению, переданному в SQL-оператор извне. Параметры используются в определениях SQL-функций и подготовленных операторов. Некоторые клиентские библиотеки также поддерживают передачу значений данных отдельно от самой SQL-команды, и в этом случае параметры позволяют ссылаться на такие значения. Ссылка на параметр записывается в следующей форме:

```
$число
```

Например, рассмотрим следующее определение функции `dept`:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Здесь `$1` всегда будет ссылаться на значение первого аргумента функции.

4.2.3. Индексы элементов

Если в выражении вы имеете дело с массивом, то можно извлечь определённый его элемент, написав:

```
выражение[индекс]
```

или несколько соседних элементов («срез массива»):

```
выражение[нижний_индекс:верхний_индекс]
```

(Здесь квадратные скобки `[ ]` должны присутствовать буквально.) Каждый *`индекс`* сам по себе является выражением, результат которого округляется к ближайшему целому.

В общем случае *`выражение`* массива должно заключаться в круглые скобки, но их можно опустить, когда выражение с индексом — это просто ссылка на столбец или позиционный параметр. Кроме того, можно соединить несколько индексов, если исходный массив многомерный. Например:

```
моя_таблица.столбец_массив[4]
моя_таблица.столбец_массив_2d[17][34]
$1[10:42]
(функция_массив(a,b))[42]
```

В последней строке круглые скобки необходимы. Подробнее массивы рассматриваются в Разделе 8.15.

4.2.4. Выбор поля

Если результат выражения — значение составного типа (строка таблицы), тогда определённое поле этой строки можно извлечь, написав:

```
выражение.имя_поля
```

В общем случае *выражение* такого типа должно заключаться в круглые скобки, но их можно опустить, когда это ссылка на таблицу или позиционный параметр. Например:

```
моя_таблица.столбец
$1.столбец
(функция_кортеж(a,b)).стол3
```

(Таким образом, полная ссылка на столбец — это просто частный случай выбора поля.) Важный особый случай здесь — извлечение поля из столбца составного типа:

```
(составной_столбец).поле
(моя_таблица.составной_столбец).поле
```

Здесь скобки нужны, чтобы показать, что `составной_столбец` — это имя столбца, а не таблицы, и что `моя_таблица` — имя таблицы, а не схемы.

Вы можете запросить все поля составного значения, написав `.*`:

```
(составной_столбец).*
```

Эта запись действует по-разному в зависимости от контекста; подробнее об этом говорится в Подразделе 8.16.5.

4.2.5. Применение оператора

Существуют два возможных синтаксиса применения операторов:

выражение оператор выражение (бинарный инфиксный оператор)
оператор выражение (унарный префиксный оператор)

где *оператор* соответствует синтаксическим правилам, описанным в Подразделе 4.1.3, либо это одно из ключевых слов `AND`, `OR` и `NOT`, либо полное имя оператора в форме:

```
OPERATOR(схема.имя_оператора)
```

Существование конкретных операторов и их тип (унарный или бинарный) зависит от того, как и какие операторы определены системой и пользователем. Встроенные операторы описаны в Главе 9.

4.2.6. Вызовы функций

Вызов функции записывается просто как имя функции (возможно, дополненное именем схемы) и список аргументов в скобках:

```
имя_функции ([выражение [, выражение ... ]])
```

Например, так вычисляется квадратный корень из 2:

```
sqrt(2)
```

Список встроенных функций приведён в Главе 9. Пользователь также может определить и другие функции.

Выполняя запросы в базе данных, где одни пользователи могут не доверять другим, в записи вызовов функций соблюдайте меры предосторожности, описанные в Разделе 10.3.

Аргументам могут быть присвоены необязательные имена. Подробнее об этом см. Раздел 4.3.

Примечание

Функцию, принимающую один аргумент составного типа, можно также вызывать, используя синтаксис выбора поля, и наоборот, выбор поля можно записать в функциональном стиле. То есть записи `col(table)` и `table.col` равносильны и взаимозаменяемы. Это поведение не оговорено стандартом SQL, но реализовано в Postgres Pro, так как это позволяет использовать функции для эмуляции «вычисляемых полей». Подробнее это описано в Подразделе 8.16.5.

4.2.7. Агрегатные выражения

Агрегатное выражение представляет собой применение агрегатной функции к строкам, выбранным запросом. Агрегатная функция сводит множество входных значений к одному выходному, как например, сумма или среднее. Агрегатное выражение может записываться следующим образом:

```
агрегатная_функция (выражение [ , ... ] [ предложение_order_by ] ) [ FILTER
 ( WHERE условие_фильтра ) ]
агрегатная_функция (ALL выражение [ , ... ] [ предложение_order_by ] ) [ FILTER
 ( WHERE условие_фильтра ) ]
агрегатная_функция (DISTINCT выражение [ , ... ] [ предложение_order_by ] ) [ FILTER
 ( WHERE условие_фильтра ) ]
агрегатная_функция ( * ) [ FILTER ( WHERE условие_фильтра ) ]
агрегатная_функция ( [ выражение [ , ... ] ] ) WITHIN GROUP ( предложение_order_by )
 [ FILTER ( WHERE условие_фильтра ) ]
```

Здесь *`агрегатная_функция`* — имя ранее определённой агрегатной функции (возможно, дополненное именем схемы), *`выражение`* — любое выражение значения, не содержащее в себе агрегатного выражения или вызова оконной функции. Необязательные предложения *`предложение_order_by`* и *`условие_фильтра`* описываются ниже.

В первой форме агрегатного выражения агрегатная функция вызывается для каждой строки. Вторая форма эквивалентна первой, так как указание `ALL` подразумевается по умолчанию. В третьей форме агрегатная функция вызывается для всех различных значений выражения (или набора различных значений, для нескольких выражений), выделенных во входных данных. В четвёртой форме агрегатная функция вызывается для каждой строки, так как никакого конкретного значения не указано (обычно это имеет смысл только для функции `count(*)`). В последней форме используются *сортирующие* агрегатные функции, которые будут описаны ниже.

Большинство агрегатных функций игнорируют значения NULL, так что строки, для которых выражения выдают одно или несколько значений NULL, отбрасываются. Это можно считать истинным для всех встроенных операторов, если явно не говорится об обратном.

Например, `count(*)` подсчитает общее количество строк, а `count(f1)` только количество строк, в которых `f1` не NULL (так как `count` игнорирует NULL), а `count(distinct f1)` подсчитает число различных и отличных от NULL значений столбца `f1`.

Обычно строки данных передаются агрегатной функции в неопределённом порядке и во многих случаях это не имеет значения, например функция `min` выдаёт один и тот же результат независимо от порядка поступающих данных. Однако некоторые агрегатные функции (такие как `array_agg` и `string_agg`) выдают результаты, зависящие от порядка данных. Для таких агрегатных функций можно добавить *`предложение_order_by`* и задать нужный порядок. Это *`предложение_order_by`* имеет тот же синтаксис, что и предложение `ORDER BY` на уровне запроса, как описано в Разделе 7.5, за исключением того, что его выражения должны быть просто выражениями, а не именами результирующих столбцов или числами. Например:

```
WITH vals (v) AS ( VALUES (1),(3),(4),(3),(2) )
SELECT array_agg(v ORDER BY v DESC) FROM vals;
  array_agg
-------------
 {4,3,3,2,1}
```

При работе с `jsonb` порядок ключей имеет значение, потому что при наличии одинаковых ключей в объекте этого типа сохраняется только последний:

```
WITH vals (k, v) AS ( VALUES ('key0','1'), ('key1','3'), ('key1','2') )
SELECT jsonb_object_agg(k, v ORDER BY v) FROM vals;
```

```
      jsonb_object_agg
----------------------------
 {"key0": "1", "key1": "3"}
```

Заметьте, что при использовании агрегатных функций с несколькими аргументами, предложение `ORDER BY` идёт после всех аргументов. Например, надо писать так:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

а не так:

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- неправильно
```

Последний вариант синтаксически допустим, но он представляет собой вызов агрегатной функции одного аргумента с двумя ключами `ORDER BY` (при этом второй не имеет смысла, так как это константа).

Если *`предложение_order_by`* дополнено указанием `DISTINCT`, тогда выражения `ORDER BY` могут ссылаться только на столбцы в списке `DISTINCT`. Например:

```
WITH vals (v) AS ( VALUES (1),(3),(4),(3),(2) )
SELECT array_agg(DISTINCT v ORDER BY v DESC) FROM vals;
 array_agg
-----------
 {4,3,2,1}
```

При добавлении `ORDER BY` в обычный список аргументов агрегатной функции, описанном до этого, выполняется сортировка входных строк для универсальных и статистических агрегатных функций, для которых сортировка необязательна. Но есть подмножество агрегатных функций, *сортирующие агрегатные функции*, для которых *`предложение_order`* является *обязательным*, обычно потому, что вычисление этой функции имеет смысл только при определённой сортировке входных строк. Типичными примерами сортирующих агрегатных функций являются вычисления ранга и процентиля. Для сортирующей агрегатной функции *`предложение_order_by`* записывается внутри `WITHIN GROUP (...)`, что иллюстрирует последний пример, приведённый выше. Выражения в *`предложении_order_by`* вычисляются однократно для каждой входной строки как аргументы обычной агрегатной функции, сортируются в соответствии с требованием *`предложения_order_by`* и поступают в агрегатную функции как входящие аргументы. (Если же *`предложение_order_by`* находится не в `WITHIN GROUP`, оно не передаётся как аргумент(ы) агрегатной функции.) Выражения-аргументы, предшествующие `WITHIN GROUP`, (если они есть), называются *непосредственными аргументами*, а выражения, указанные в *`предложении_order_by`* — *агрегируемыми аргументами*. В отличие от аргументов обычной агрегатной функции, непосредственные аргументы вычисляются однократно для каждого вызова функции, а не для каждой строки. Это значит, что они могут содержать переменные, только если эти переменные сгруппированы в `GROUP BY`; это суть то же ограничение, что действовало бы, будь эти непосредственные аргументы вне агрегатного выражения. Непосредственные аргументы обычно используются, например, для указания значения процентиля, которое имеет смысл, только если это конкретное число для всего расчёта агрегатной функции. Список непосредственных аргументов может быть пуст; в этом случае запишите просто `()`, но не `(*)`. (На самом деле Postgres Pro примет обе записи, но только первая соответствует стандарту SQL.)

Пример вызова сортирующей агрегатной функции:

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
           50489
```

она получает 50-ый процентиль, или медиану, значения столбца `income` из таблицы `households`. В данном случае `0.5` — это непосредственный аргумент; если бы дробь процентиля менялась от строки к строке, это не имело бы смысла.

Если добавлено предложение `FILTER`, агрегатной функции подаются только те входные строки, для которых *`условие_фильтра`* вычисляется как истинное; другие строки отбрасываются. Например:

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

Предопределённые агрегатные функции описаны в Разделе 9.21. Пользователь также может определить другие агрегатные функции.

Агрегатное выражение может фигурировать только в списке результатов или в предложении `HAVING` команды `SELECT`. Во всех остальных предложениях, например `WHERE`, они запрещены, так как эти предложения логически вычисляются до того, как формируются результаты агрегатных функций.

Когда агрегатное выражение используется в подзапросе (см. Подраздел 4.2.11 и Раздел 9.24), оно обычно вычисляется для всех строк подзапроса. Но если в аргументах (или в *`условии_filter`*) агрегатной функции есть только переменные внешнего уровня, агрегатная функция относится к ближайшему внешнему уровню и вычисляется для всех строк соответствующего запроса. Такое агрегатное выражение в целом является внешней ссылкой для своего подзапроса и на каждом вычислении считается константой. При этом допустимое положение агрегатной функции ограничивается списком результатов и предложением `HAVING` на том уровне запросов, где она находится.

4.2.8. Вызовы оконных функций

Вызов оконной функции представляет собой применение функции, подобной агрегатной, к некоторому набору строк, выбранному запросом. В отличие от вызовов не оконных агрегатных функций, при этом не происходит группировка выбранных строк в одну — каждая строка остаётся отдельной в результате запроса. Однако оконная функция имеет доступ ко всем строкам, вошедшим в группу текущей строки согласно указанию группировки (списку `PARTITION BY`) в вызове оконной функции. Вызов оконной функции может иметь следующие формы:

```
имя_функции ([выражение [, выражение ... ]]) [ FILTER ( WHERE предложение_фильтра ) ]
 OVER имя_окна
имя_функции ([выражение [, выражение ... ]]) [ FILTER ( WHERE предложение_фильтра ) ]
 OVER ( определение_окна )
имя_функции ( * ) [ FILTER ( WHERE предложение_фильтра ) ] OVER имя_окна
имя_функции ( * ) [ FILTER ( WHERE предложение_фильтра ) ] OVER ( определение_окна )
```

Здесь *`определение_окна`* записывается в виде

```
[ имя_существующего_окна ]
[ PARTITION BY выражение [, ...] ]
[ ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
[ определение_рамки ]
```

Необязательное *`определение_рамки`* может иметь вид:

```
{ RANGE | ROWS | GROUPS } начало_рамки [ исключение_рамки ]
{ RANGE | ROWS | GROUPS } BETWEEN начало_рамки AND конец_рамки [ исключение_рамки ]
```

Здесь *`начало_рамки`* и *`конец_рамки`* задаются одним из следующих способов:

```
UNBOUNDED PRECEDING
смещение PRECEDING
CURRENT ROW
смещение FOLLOWING
UNBOUNDED FOLLOWING
```

и *`исключение_рамки`* может быть следующим:

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

Здесь *`выражение`* — это любое выражение значения, не содержащее вызовов оконных функций.

`имя_окна` — ссылка на именованное окно, определённое предложением `WINDOW` в данном запросе. Также возможно написать в скобках полное *`определение_окна`*, используя тот же синтаксис определения именованного окна в предложении `WINDOW`; подробнее это описано в справке по SELECT. Стоит отметить, что запись `OVER имя_окна` не полностью равнозначна `OVER (имя_окна ...)`; последний вариант подразумевает копирование и изменение определения окна и не будет допустимым, если определение этого окна включает определение рамки.

Указание `PARTITION BY` группирует строки запроса в *разделы*, которые затем обрабатываются оконной функцией независимо друг от друга. `PARTITION BY` работает подобно предложению `GROUP BY` на уровне запроса, за исключением того, что его аргументы всегда просто выражения, а не имена выходных столбцов или числа. Без `PARTITION BY` все строки, выдаваемые запросом, рассматриваются как один раздел. Указание `ORDER BY` определяет порядок, в котором оконная функция обрабатывает строки раздела. Оно так же подобно предложению `ORDER BY` на уровне запроса и так же не принимает имена выходных столбцов или числа. Без `ORDER BY` строки обрабатываются в неопределённом порядке.

`определение_рамки` задаёт набор строк, образующих *рамку окна*, которая представляет собой подмножество строк текущего раздела и используется для оконных функций, работающих с рамкой, а не со всем разделом. Подмножество строк в рамке может меняться в зависимости от того, какая строка является текущей. Рамку можно задать в режимах `RANGE`, `ROWS` или `GROUPS`; в каждом случае она начинается с положения *`начало_рамки`* и заканчивается положением *`конец_рамки`*. Если *`конец_рамки`* не задаётся явно, подразумевается `CURRENT ROW` (текущая строка).

Если *`начало_рамки`* задано как `UNBOUNDED PRECEDING`, рамка начинается с первой строки раздела, а если *`конец_рамки`* определён как `UNBOUNDED FOLLOWING`, рамка заканчивается последней строкой раздела.

В режиме `RANGE` или `GROUPS` *`начало_рамки`*, заданное как `CURRENT ROW`, определяет в качестве начала первую *родственную* строку (строку, которая при сортировке согласно указанному для окна предложению `ORDER BY` считается равной текущей), тогда как *`конец_рамки`*, заданный как `CURRENT ROW`, определяет концом рамки последнюю родственную строку. В режиме `ROWS` вариант `CURRENT ROW` просто обозначает текущую строку.

В вариантах определения рамки *`смещение`* `PRECEDING` и *`смещение`* `FOLLOWING` в качестве *`смещения`* должно задаваться выражение, не содержащее какие-либо переменные и вызовы агрегатных или оконных функций. Что именно будет означать *`смещение`*, определяется в зависимости от режима рамки:

- В режиме `ROWS` *`смещение`* должно задаваться отличным от NULL неотрицательным целым числом, и это число определяет сдвиг, с которым начало рамки позиционируется перед текущей строкой, а конец — после текущей строки.
- В режиме `GROUPS` *`смещение`* также должно задаваться отличным от NULL неотрицательным целым числом, и это число определяет сдвиг (по количеству групп родственных строк), с которым начало рамки позиционируется перед группой строк, родственных текущей, а конец — после этой группы. Группу родственных строк образуют строки, которые считаются равными согласно `ORDER BY`. (Для использования режима `GROUPS` определение окна должно содержать предложение `ORDER BY`.)
- В режиме `RANGE` для использования этих указаний предложение `ORDER BY` должно содержать ровно один столбец. В этом случае *`смещение`* задаёт максимальную разницу между значением этого столбца в текущей строке и значением его же в предшествующих или последующих строках рамки. Тип данных выражения *`смещение`* зависит от типа данных упорядочиваю-

щего столбца. Для числовых столбцов это обычно тот же числовой тип, а для столбцов с типом дата/время — тип `interval`. Например, если упорядочивающий столбец имеет тип `date` или `timestamp`, возможна такая запись: `RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING`. Значение *смещение* при этом может так же быть отличным от NULL и неотрицательным, хотя что считать «неотрицательным», будет зависит от типа данных.

В любом случае расстояние до конца рамки ограничивается расстоянием до конца раздела, так что для строк, которые находятся у конца раздела, рамка может содержать меньше строк, чем для других.

Заметьте, что в режимах `ROWS` и `GROUPS` указания `0 PRECEDING` и `0 FOLLOWING` равнозначны указанию `CURRENT ROW`. Обычно это справедливо и для режима `RANGE`, в случае подходящего для типа данных определения значения «нуля».

Дополнение *исключение_рамки* позволяет исключить из рамки строки, которые окружают текущую строку, даже если они должны быть включены согласно указаниям, определяющим начало и конец рамки. `EXCLUDE CURRENT ROW` исключает из рамки текущую строку. `EXCLUDE GROUP` исключает из рамки текущую строку и родственные ей согласно порядку сортировки. `EXCLUDE TIES` исключает из рамки все родственные строки для текущей, но не собственно текущую строку. `EXCLUDE NO OTHERS` просто явно выражает поведение по умолчанию — не исключает ни текущую строку, ни родственные ей.

По умолчанию рамка определяется как `RANGE UNBOUNDED PRECEDING`, что равносильно расширенному определению `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. С указанием `ORDER BY` это означает, что рамка будет включать все строки от начала раздела до последней строки, родственной текущей (для `ORDER BY`). Без `ORDER BY` это означает, что в рамку включаются все строки раздела, так как все они считаются родственными текущей.

Действуют также следующие ограничения: в качестве *начала_рамки* нельзя задать `UNBOUNDED FOLLOWING`, в качестве *конца_рамки* не допускается `UNBOUNDED PRECEDING` и *конец_рамки* не может идти в показанном выше списке указаний *начало_рамки* AND *конец_рамки* перед *началом_рамки*. В частности, синтаксис `RANGE BETWEEN CURRENT ROW AND` *смещение* `PRECEDING` не допускается. Но при этом, например, определение `ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING` допустимо, хотя оно и не выберет никакие строки.

Если добавлено предложение `FILTER`, оконной функции подаются только те входные строки, для которых *условие_фильтра* вычисляется как истинное; другие строки отбрасываются. Предложение `FILTER` допускается только для агрегирующих оконных функций.

Встроенные оконные функции описаны в Таблице 9.65, но пользователь может расширить этот набор, создавая собственные функции. Кроме того, в качестве оконных функций можно использовать любые встроенные или пользовательские универсальные, а также статистические агрегатные функции. (Сортирующие и гипотезирующие агрегатные функции в настоящее время использовать в качестве оконных нельзя.)

Запись со звёздочкой (`*`) применяется при вызове не имеющих параметров агрегатных функций в качестве оконных, например `count(*) OVER (PARTITION BY x ORDER BY y)`. Звёздочка (`*`) обычно не применяется для исключительно оконных функций. Такие функции не допускают использования `DISTINCT` и `ORDER BY` в списке аргументов функции.

Вызовы оконных функций разрешены в запросах только в списке `SELECT` и в предложении `ORDER BY`.

Дополнительно об оконных функциях можно узнать в Разделе 3.5, Разделе 9.22 и Подразделе 7.2.5.

4.2.9. Приведения типов

Приведение типа определяет преобразование данных из одного типа в другой. Postgres Pro воспринимает две равносильные записи приведения типов:

```
CAST ( выражение AS тип )
выражение::тип
```

Запись с `CAST` соответствует стандарту SQL, тогда как вариант с `::` — историческое наследие Postgres Pro.

Когда приведению подвергается значение выражения известного типа, происходит преобразование типа во время выполнения. Это приведение будет успешным, только если определён подходящий оператор преобразования типов. Обратите внимание на небольшое отличие от приведения констант, описанного в Подразделе 4.1.2.7. Приведение строки в чистом виде представляет собой начальное присваивание строковой константы и оно будет успешным для любого типа (конечно, если строка содержит значение, приемлемое для данного типа данных).

Явное приведение типа можно опустить, если возможно однозначно определить, какой тип должно иметь выражение (например, когда оно присваивается столбцу таблицы); в таких случаях система автоматически преобразует тип. Однако автоматическое преобразование выполняется только для приведений с пометкой «допускается неявное применение» в системных каталогах. Все остальные приведения должны записываться явно. Это ограничение позволяет избежать сюрпризов с неявным преобразованием.

Также можно записать приведение типа как вызов функции:

```
имя_типа ( выражение )
```

Однако это будет работать только для типов, имена которых являются также допустимыми именами функций. Например, `double precision` так использовать нельзя, а `float8` (альтернативное название того же типа) — можно. Кроме того, имена типов `interval`, `time` и `timestamp` из-за синтаксического конфликта можно использовать в такой записи только в кавычках. Таким образом, запись приведения типа в виде вызова функции провоцирует несоответствия и, возможно, лучше будет её не применять.

Примечание

Приведение типа, представленное в виде вызова функции, на самом деле соответствует внутреннему механизму. Даже при использовании двух стандартных типов записи внутри происходит вызов зарегистрированной функции, выполняющей преобразование. По соглашению именем такой функции преобразования является имя выходного типа, и таким образом запись «в виде вызова функции» есть не что иное, как прямой вызов нижележащей функции преобразования. При создании переносимого приложения на это поведение, конечно, не следует рассчитывать. Подробнее это описано в справке CREATE CAST.

4.2.10. Применение правил сортировки

Предложение `COLLATE` переопределяет правило сортировки выражения. Оно добавляется после выражения:

```
выражение COLLATE правило_сортировки
```

где *правило_сортировки* — идентификатор правила, возможно дополненный именем схемы. Предложение `COLLATE` связывает выражение сильнее, чем операторы, так что при необходимости следует использовать скобки.

Если правило сортировки не определено явно, система либо выбирает его по столбцам, которые используются в выражении, либо, если таких столбцов нет, переключается на установленное для базы данных правило сортировки по умолчанию.

Предложение `COLLATE` имеет два распространённых применения: переопределение порядка сортировки в предложении `ORDER BY`, например:

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

и переопределение правил сортировки при вызове функций или операторов, возвращающих языкозависимые результаты, например:

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

Заметьте, что в последнем случае предложение `COLLATE` добавлено к аргументу оператора, на действие которого мы хотим повлиять. При этом не имеет значения, к какому именно аргументу оператора или функции добавляется `COLLATE`, так как правило сортировки, применяемое к оператору или функции, выбирается при рассмотрении всех аргументов, а явное предложение `COLLATE` переопределяет правила сортировки для всех других аргументов. (Однако добавление разных предложений `COLLATE` к нескольким аргументам будет ошибкой. Подробнее об этом см. Раздел 22.2.) Таким образом, эта команда выдаст тот же результат:

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

Но это будет ошибкой:

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

здесь правило сортировки нельзя применить к результату оператора `>`, который имеет несравниваемый тип данных `boolean`.

4.2.11. Скалярные подзапросы

Скалярный подзапрос — это обычный запрос `SELECT` в скобках, который возвращает ровно одну строку и один столбец. (Написание запросов освещается в Главе 7.) После выполнения запроса `SELECT` его единственный результат используется в окружающем его выражении. В качестве скалярного подзапроса нельзя использовать запросы, возвращающие более одной строки или столбца. (Но если в результате выполнения подзапрос не вернёт строк, скалярный результат считается равным NULL.) В подзапросе можно ссылаться на переменные из окружающего запроса; в процессе одного вычисления подзапроса они будут считаться константами. Другие выражения с подзапросами описаны в Разделе 9.24.

Например, следующий запрос находит число жителей в самом населённом городе в каждом штате:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
    FROM states;
```

4.2.12. Конструкторы массивов

Конструктор массива — это выражение, которое создаёт массив, определяя значения его элементов. Конструктор простого массива состоит из ключевого слова `ARRAY`, открывающей квадратной скобки `[`, списка выражений (разделённых запятыми), задающих значения элементов массива, и закрывающей квадратной скобки `]`. Например:

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

По умолчанию типом элементов массива считается общий тип для всех выражений, определённый по правилам, действующим и для конструкций `UNION` и `CASE` (см. Раздел 10.5). Вы можете переопределить его явно, приведя конструктор массива к требуемому типу, например:

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

Это равносильно тому, что привести к нужному типу каждое выражение по отдельности. Подробнее приведение типов описано в Подразделе 4.2.9.

Многомерные массивы можно образовывать, вкладывая конструкторы массивов. При этом во внутренних конструкторах слово `ARRAY` можно опускать. Например, результат работы этих конструкторов одинаков:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Многомерные массивы должны быть прямоугольными, и поэтому внутренние конструкторы одного уровня должны создавать вложенные массивы одинаковой размерности. Любое приведение типа, применённое к внешнему конструктору `ARRAY`, автоматически распространяется на все внутренние.

Элементы многомерного массива можно создавать не только вложенными конструкторами `ARRAY`, но и другими способами, позволяющими получить массивы нужного типа. Например:

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
------------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

Вы можете создать и пустой массив, но так как массив не может быть не типизированным, вы должны явно привести пустой массив к нужному типу. Например:

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

Также возможно создать массив из результатов подзапроса. В этом случае конструктор массива записывается так же с ключевым словом `ARRAY`, за которым в круглых скобках следует подзапрос. Например:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                              array
-------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
              array
----------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

Такой подзапрос должен возвращать один столбец. Если этот столбец имеет тип, отличный от массива, результирующий одномерный массив будет включать элементы для каждой строки-результата подзапроса и типом элемента будет тип столбца результата. Если же тип столбца — массив, будет создан массив того же типа, но большей размерности; в любом случае во всех строках подзапроса должны выдаваться массивы одинаковой размерности, чтобы можно было получить прямоугольный результат.

Индексы массива, созданного конструктором `ARRAY`, всегда начинаются с одного. Подробнее о массивах вы узнаете в Разделе 8.15.

4.2.13. Конструкторы табличных строк

Конструктор табличной строки — это выражение, создающее строку или кортеж (или составное значение) из значений его аргументов-полей. Конструктор строки состоит из ключевого слова `ROW`, открывающей круглой скобки, нескольких выражений (разделённых запятыми), определяющих значения полей, и закрывающей скобки. При этом выражения могут отсутствовать, то есть конструктор имеет вид ROW(). Например:

```
SELECT ROW(1,2.5,'this is a test');
```

Если в списке более одного выражения, ключевое слово `ROW` можно опустить.

Конструктор строки поддерживает запись *`составное_значение`*`.*`, при этом данное значение будет развёрнуто в список элементов, так же, как в записи `.*` на верхнем уровне списка `SELECT` (см. Подраздел 8.16.5). Например, если таблица `t` содержит столбцы `f1` и `f2`, эти записи равнозначны:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

Примечание

До версии PostgreSQL 8.2 запись `.*` не разворачивалась в конструкторах строк, так что выражение `ROW(t.*, 42)` создавало составное значение из двух полей, в котором первое поле так же было составным. Новое поведение обычно более полезно. Если вам нужно получить прежнее поведение, чтобы одно значение строки было вложено в другое, напишите внутреннее значение без `.*`, например: `ROW(t, 42)`.

По умолчанию значение, созданное выражением `ROW`, имеет тип анонимной записи. Если необходимо, его можно привести к именованному составному типу — либо к типу строки таблицы, либо составному типу, созданному оператором `CREATE TYPE AS`. Явное приведение может потребоваться для достижения однозначности. Например:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Приведение не требуется, так как существует только одна getf1()
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Теперь приведение необходимо для однозначного выбора функции:
SELECT getf1(ROW(1,2.5,'this is a test'));
ОШИБКА:  функция getf1(record) не уникальна

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)
```

```
SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

Используя конструктор строк (кортежей), можно создавать составное значение для сохранения в столбце составного типа или для передачи функции, принимающей составной параметр. Кроме того, можно тестировать строки, используя стандартные операторы сравнения, а также сравнивать строки между собой, как описано в Разделе 9.25, и использовать их в сочетании с подзапросами, как описано в Разделе 9.24.

4.2.14. Правила вычисления выражений

Порядок вычисления подвыражений не определён. В частности, аргументы оператора или функции не обязательно вычисляются слева направо или в любом другом фиксированном порядке.

Более того, если результат выражения можно получить, вычисляя только некоторые его части, тогда другие подвыражения не будут вычисляться вовсе. Например, если написать:

```
SELECT true OR somefunc();
```

тогда функция `somefunc()` не будет вызываться (возможно). То же самое справедливо для записи:

```
SELECT somefunc() OR true;
```

Заметьте, что это отличается от «оптимизации» вычисления логических операторов слева направо, реализованной в некоторых языках программирования.

Как следствие, в сложных выражениях не стоит использовать функции с побочными эффектами. Особенно опасно рассчитывать на порядок вычисления или побочные эффекты в предложениях `WHERE` и `HAVING`, так как эти предложения тщательно оптимизируются при построении плана выполнения. Логические выражения (сочетания `AND`/`OR`/`NOT`) в этих предложениях могут быть видоизменены любым способом, допустимым законами Булевой алгебры.

Когда порядок вычисления важен, его можно зафиксировать с помощью конструкции `CASE` (см. Раздел 9.18). Например, такой способ избежать деления на ноль в предложении `WHERE` ненадёжен:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

Безопасный вариант:

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

Применяемая так конструкция `CASE` защищает выражение от оптимизации, поэтому использовать её нужно только при необходимости. (В данном случае было бы лучше решить проблему, переписав условие как `y > 1.5*x`.)

Однако `CASE` не всегда спасает в подобных случаях. Показанный выше приём плох тем, что не предотвращает раннее вычисление константных подвыражений. Как описано в Разделе 36.7, функции и операторы, помеченные как `IMMUTABLE`, могут вычисляться при планировании, а не выполнении запроса. Поэтому в примере

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

, скорее всего, произойдёт деление на ноль из-за того, что планировщик попытается упростить константное подвыражение, даже если во всех строках в таблице `x > 0`, а значит во время выполнения ветвь `ELSE` никогда не будет выполняться.

Хотя этот конкретный пример может показаться надуманным, похожие ситуации, в которых неявно появляются константы, могут возникать и в запросах внутри функций, так как значения аргументов функции и локальных переменных при планировании могут быть заменены константами. Поэтому, например, в функциях PL/pgSQL гораздо безопаснее для защиты от рискованных вычислений использовать конструкцию `IF`-`THEN`-`ELSE`, чем выражение `CASE`.

Ещё один подобный недостаток этого подхода в том, что `CASE` не может предотвратить вычисление заключённого в нём агрегатного выражения, так как агрегатные выражения вычисляются перед всеми остальными в списке `SELECT` или предложении `HAVING`. Например, в следующем запросе может возникнуть ошибка деления на ноль, несмотря на то, что он вроде бы защищён от неё:

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

Агрегатные функции `min()` и `avg()` вычисляются независимо по всем входным строкам, так что если в какой-то строке поле `employees` окажется равным нулю, деление на ноль произойдёт раньше, чем станет возможным проверить результат функции `min()`. Поэтому, чтобы проблемные входные строки изначально не попали в агрегатную функцию, следует воспользоваться предложениями `WHERE` или `FILTER`.

4.3. Вызов функций

Postgres Pro позволяет вызывать функции с именованными параметрами, используя запись с *позиционной* или *именной* передачей аргументов. Именная передача особенно полезна для функций со множеством параметров, так как она делает связь параметров и аргументов более явной и надёжной. В позиционной записи значения аргументов функции указываются в том же порядке, в каком они описаны в определении функции. При именной передаче аргументы сопоставляются с параметрами функции по именам и указывать их можно в любом порядке. Для каждого варианта вызова также учитывайте влияние типов аргументов функций, описанное в Разделе 10.3.

При записи любым способом параметры, для которых в определении функции заданы значения по умолчанию, можно вовсе не указывать. Но это особенно полезно при именной передаче, так как опустить можно любой набор параметров, тогда как при позиционной параметры можно опускать только последовательно, справа налево.

Postgres Pro также поддерживает *смешанную* передачу, когда параметры передаются и по именам, и по позиции. В этом случае позиционные параметры должны идти перед параметрами, передаваемыми по именам.

Мы рассмотрим все три варианта записи на примере следующей функции:

```
CREATE FUNCTION concat_lower_or_upper(a text, b text,
  uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

Функция `concat_lower_or_upper` имеет два обязательных параметра: `a` и `b`. Кроме того, есть один необязательный параметр `uppercase`, который по умолчанию имеет значение `false`. Аргументы `a` и `b` будут сложены вместе и переведены в верхний или нижний регистр, в зависимости от параметра `uppercase`. Остальные тонкости реализации функции сейчас не важны (подробнее о них рассказано в Главе 36).

4.3.1. Позиционная передача

Позиционная передача — это традиционный механизм передачи аргументов функции в Postgres Pro. Пример такой записи:

```
SELECT concat_lower_or_upper('Hello', 'World', true);
```

```
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

Все аргументы указаны в заданном порядке. Результат возвращён в верхнем регистре, так как параметр `uppercase` имеет значение `true`. Ещё один пример:

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Здесь параметр `uppercase` опущен, и поэтому он принимает значение по умолчанию (`false`), и результат переводится в нижний регистр. В позиционной записи любые аргументы с определённым значением по умолчанию можно опускать справа налево.

4.3.2. Именная передача

При именной передаче для аргумента добавляется имя, которое отделяется от выражения значения знаками `=>`. Например:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Здесь аргумент `uppercase` был так же опущен, так что он неявно получил значение `false`. Преимуществом такой записи является возможность записывать аргументы в любом порядке, например:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

Для обратной совместимости поддерживается и старый синтаксис с «:=»:

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. Смешанная передача

При смешанной передаче параметры передаются и по именам, и по позиции. Однако как уже было сказано, именованные аргументы не могут стоять перед позиционными. Например:

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

В данном запросе аргументы `a` и `b` передаются по позиции, а `uppercase` — по имени. Единственное обоснование такого вызова здесь — он стал чуть более читаемым. Однако для более сложных функций с множеством аргументов, часть из которых имеют значения по умолчанию, именная или смешанная передача позволяют записать вызов эффективнее и уменьшить вероятность ошибок.

Примечание

Именная и смешанная передача в настоящий момент не может использоваться при вызове агрегатной функции (но они допускаются, если агрегатная функция используется в качестве оконной).

Глава 5. Определение данных

Эта глава рассказывает, как создавать структуры базы данных, в которых будут храниться данные. В реляционной базе данных данные хранятся в таблицах, так что большая часть этой главы будет посвящена созданию и изменению таблиц, а также средствам управления данными в них. Затем мы обсудим, как таблицы можно объединять в схемы и как ограничивать доступ к ним. Наконец, мы кратко рассмотрим другие возможности, связанные с хранением данных, в частности наследование, секционирование таблиц, представления, функции и триггеры.

5.1. Основы таблиц

Таблица в реляционной базе данных похожа на таблицу на бумаге: она так же состоит из строк и столбцов. Число и порядок столбцов фиксированы, а каждый столбец имеет имя. Число строк переменно — оно отражает текущее количество находящихся в ней данных. SQL не даёт никаких гарантий относительно порядка строк таблицы. При чтении таблицы строки выводятся в произвольном порядке, если только явно не требуется сортировка. Подробнее это рассматривается в Главе 7. Более того, SQL не назначает строкам уникальные идентификаторы, так что можно иметь в таблице несколько полностью идентичных строк. Это вытекает из математической модели, которую реализует SQL, но обычно такое дублирование нежелательно. Позже в этой главе мы увидим, как его избежать.

Каждому столбцу сопоставлен тип данных. Тип данных ограничивает набор допустимых значений, которые можно присвоить столбцу, и определяет смысловое значение данных для вычислений. Например, в столбец числового типа нельзя записать обычные текстовые строки, но зато его данные можно использовать в математических вычислениях. И наоборот, если столбец имеет тип текстовой строки, для него допустимы практически любые данные, но он непригоден для математических действий (хотя другие операции, например конкатенация строк, возможны).

В Postgres Pro есть внушительный набор встроенных типов данных, удовлетворяющий большинство приложений. Пользователи также могут определять собственные типы данных. Большинство встроенных типов данных имеют понятные имена и семантику, так что мы отложим их подробное рассмотрение до Главы 8. Наиболее часто применяются следующие типы данных: `integer` для целых чисел, `numeric` для чисел, которые могут быть дробными, `text` для текстовых строк, `date` для дат, `time` для времени и `timestamp` для значений, включающих дату и время.

Для создания таблицы используется команда CREATE TABLE. В этой команде вы должны указать как минимум имя новой таблицы и имена и типы данных каждого столбца. Например:

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

Так вы создадите таблицу `my_first_table` с двумя столбцами. Первый столбец называется `first_column` и имеет тип данных `text`; второй столбец называется `second_column` и имеет тип `integer`. Имена таблицы и столбцов соответствуют синтаксису идентификаторов, описанному в Подразделе 4.1.1. Имена типов также являются идентификаторами, хотя есть некоторые исключения. Заметьте, что список столбцов заключается в скобки, а его элементы разделяются запятыми.

Конечно, предыдущий пример ненатурален. Обычно в именах таблиц и столбцов отражается, какие данные они будут содержать. Поэтому давайте взглянем на более реалистичный пример:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

(Тип `numeric` может хранить дробные числа, в которых обычно выражаются денежные суммы.)

Подсказка

Когда вы создаёте много взаимосвязанных таблиц, имеет смысл заранее выбрать единый шаблон именования таблиц и столбцов. Например, решить, будут ли в именах таблиц использоваться существительные во множественном или в единственном числе (есть соображения в пользу каждого варианта).

Число столбцов в таблице не может быть бесконечным. Это число ограничивается максимумом в пределах от 250 до 1600, в зависимости от типов столбцов. Однако создавать таблицы с таким большим числом столбцов обычно не требуется, а если такая потребность возникает, это скорее признак сомнительного дизайна.

Если таблица вам больше не нужна, вы можете удалить её, выполнив команду DROP TABLE. Например:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Попытка удаления несуществующей таблицы считается ошибкой. Тем не менее в SQL-скриптах часто применяют безусловное удаление таблиц перед созданием, игнорируя все сообщения об ошибках, так что они выполняют свою задачу независимо от того, существовали таблицы или нет. (Если вы хотите избежать таких ошибок, можно использовать вариант `DROP TABLE IF EXISTS`, но это не будет соответствовать стандарту SQL.)

Как изменить существующую таблицу, будет рассмотрено в этой главе позже, в Разделе 5.7.

Имея средства, которые мы обсудили, вы уже можете создавать полностью функциональные таблицы. В продолжении этой главы рассматриваются дополнительные возможности, призванные обеспечить целостность данных, безопасность и удобство. Если вам не терпится наполнить свои таблицы данными, вы можете вернуться к этой главе позже, а сейчас перейти к Главе 6.

5.2. Значения по умолчанию

Столбцу можно назначить значение по умолчанию. Когда добавляется новая строка и каким-то её столбцам не присваиваются значения, эти столбцы принимают значения по умолчанию. Также команда управления данными может явно указать, что столбцу должно быть присвоено значение по умолчанию, не зная его. (Подробнее команды управления данными описаны в Главе 6.)

Если значение по умолчанию не объявлено явно, им считается значение NULL. Обычно это имеет смысл, так как можно считать, что NULL представляет неизвестные данные.

В определении таблицы значения по умолчанию указываются после типа данных столбца. Например:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

Значение по умолчанию может быть выражением, которое в этом случае вычисляется в момент присваивания значения по умолчанию (а *не* когда создаётся таблица). Например, столбцу `timestamp` в качестве значения по умолчания часто присваивается `CURRENT_TIMESTAMP`, чтобы в момент добавления строки в нём оказалось текущее время. Ещё один распространённый пример — генерация «последовательных номеров» для всех строк. В Postgres Pro это обычно делается примерно так:

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
```

```
    ...
);
```

здесь функция `nextval()` выбирает очередное значение из *последовательности* (см. Раздел 9.17). Это употребление настолько распространено, что для него есть специальная короткая запись:

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

`SERIAL` обсуждается позже в Подразделе 8.1.4.

5.3. Столбцы идентификации

Столбец идентификации — это специальный столбец, который автоматически генерируется из неявной последовательности. Его можно использовать, чтобы генерировать значения ключей.

Чтобы создать столбец идентификации, воспользуйтесь предложением `GENERATED ... AS IDENTITY` команды `CREATE TABLE`, например:

```
CREATE TABLE people (
    id bigint GENERATED ALWAYS AS IDENTITY,
    ...,
);
```

или

```
CREATE TABLE people (
    id bigint GENERATED BY DEFAULT AS IDENTITY,
    ...,
);
```

За подробностями обратитесь к CREATE TABLE.

Если команда `INSERT` выполняется для таблицы со столбцом идентификации, и значение для этого столбца не указано явно, вставляется значение, сгенерированное неявной последовательностью. Например, с приведёнными выше определениями и при условии наличия дополнительных нужных столбцов

```
INSERT INTO people (name, address) VALUES ('A', 'foo');
INSERT INTO people (name, address) VALUES ('B', 'bar');
```

сгенерирует значения для столбца `id`, начиная с 1, и в таблице появятся следующие данные:

```
 id | name | address
----+------+---------
  1 | A    | foo
  2 | B    | bar
```

Кроме того, можно явно запросить значение, сгенерированное последовательностью, указав вместо значения ключевое слово `DEFAULT`, например:

```
INSERT INTO people (id, name, address) VALUES (DEFAULT, 'C', 'baz');
```

Ключевое слово `DEFAULT` можно использовать аналогичным образом в командах `UPDATE`.

Таким образом, во многом столбец идентификации ведёт себя как столбец со значением по умолчанию.

Предложения `ALWAYS` и `BY DEFAULT` в определении столбца устанавливают, как явно указанные пользователем значения обрабатываются командами `INSERT` и `UPDATE`. В команде `INSERT`, в случае выбора `ALWAYS`, пользовательское значение принимается, только если в этой команде указано `OVERRIDING SYSTEM VALUE`. С предложением `BY DEFAULT` пользовательскому значению отдаётся предпочтение. Таким образом, поведение с `BY DEFAULT` похоже на поведение с использованием значений по умолчанию, когда такие значения могут быть переопределены пользовательскими, тогда как `ALWAYS` обеспечивает дополнительную защиту от случайного изменения значения.

Тип данных столбца идентификации должен поддерживаться последовательностями (см. CREATE SEQUENCE). Свойства связанной последовательности могут быть указаны при создании столбца идентификации (см. CREATE TABLE) или изменены позднее (см. ALTER TABLE).

Столбец идентификации автоматически помечается как `NOT NULL`, однако уникальность значений не гарантируется. (Обычно последовательность возвращает уникальные значения, но она может быть сброшена или значения могут быть вставлены в столбец идентификации вручную, как описано выше.) Уникальность можно обеспечить с помощью ограничения `PRIMARY KEY` или `UNIQUE`.

В иерархии наследования таблиц столбцы идентификации и их свойства в дочерней таблице не зависят от столбцов и свойств в родительской таблице. Дочерняя таблица не наследует столбцы идентификации или их свойства автоматически от родительской. Во время выполнения команд `INSERT` или `UPDATE` столбец рассматривается как столбец идентификации, если он является столбцом идентификации в таблице, указанной в операторе, и применяются соответствующие свойства идентификации.

Секции наследуют столбцы идентификации из секционированной таблицы. У них не может быть собственных столбцов идентификации. Свойства заданного столбца идентификации одинаковы во всех секциях в иерархии секций.

5.4. Генерируемые столбцы

Генерируемый столбец является столбцом особого рода, который всегда вычисляется из других. Таким образом, для столбцов он является тем же, чем представление для таблицы. Есть два типа генерируемых столбцов: сохраняемые и виртуальные. Сохраняемый генерируемый столбец вычисляется при записи (добавлении или изменении) и занимает место в таблице так же, как и обычный столбец. Виртуальный генерируемый столбец не занимает места и вычисляется при чтении. Поэтому можно сказать, что виртуальный генерируемый столбец похож на представление, а сохраняемый — на материализованное представление (за исключением того, что он всегда обновляется автоматически). В настоящее время в Postgres Pro реализованы только сохраняемые генерируемые столбцы.

Чтобы создать генерируемый столбец, воспользуйтесь предложением `GENERATED ALWAYS AS` команды `CREATE TABLE`, например:

```
CREATE TABLE people (
    ...,
    height_cm numeric,
    height_in numeric GENERATED ALWAYS AS (height_cm / 2.54) STORED
);
```

Ключевое слово `STORED`, определяющее тип хранения генерируемого столбца, является обязательным. За подробностями обратитесь к CREATE TABLE.

Произвести запись непосредственно в генерируемый столбец нельзя. Поэтому в командах `INSERT` или `UPDATE` нельзя задать значение для таких столбцов, хотя ключевое слово `DEFAULT` указать можно.

Примите к сведению следующие отличия генерируемых столбцов от столбцов со значением по умолчанию. Значение столбца по умолчанию вычисляется один раз, когда в таблицу впервые вставляется строка и никакое другое значение не задано; значение же генерируемого столбца может меняться при изменении строки и не может быть переопределено. Выражение значения по умолчанию не может обращаться к другим столбцам таблицы, а генерирующее выражение, напротив, обычно обращается к ним. В выражении значения по умолчанию могут вызываться изменчивые функции, например, `random()` или функции, зависящие от времени, а для генерируемых столбцов это не допускается.

С определением генерируемых столбцов и их содержащими таблицами связан ряд ограничений и особенностей:

- В генерирующем выражении могут использоваться только постоянные функции и не могут фигурировать подзапросы или ссылки на какие-либо значения, не относящиеся к данной строке.
- Генерирующее выражение не может обращаться к другому генерируемому столбцу.
- Генерирующее выражение не может обращаться к системным столбцам, за исключением `tableoid`.
- Для генерируемого столбца нельзя задать значение по умолчанию или свойство идентификации.
- Генерируемый столбец не может быть частью ключа секционирования.
- Генерируемые столбцы могут содержаться в сторонних таблицах. За подробностями обратитесь к CREATE FOREIGN TABLE.
- Применительно к наследованию и секционированию:
  - Если родительский столбец является генерируемым, его дочерний столбец также должен быть генерируемым; однако дочерний столбец может генерироваться другим выражением. Генерирующее выражение, которое фактически применяется во время вставки или изменения строки, связано с таблицей, в которой физически находится строка. (Это отличается от поведения столбцов со значением по умолчанию: для них применяется значение по умолчанию, которое связано с таблицей, указанной в запросе.)
  - Если родительский столбец не является генерируемым, его дочерний столбец также должен быть негенерируемым.
  - Для унаследованных таблиц, если в определении дочернего столбца нет предложения `GENERATED` в `CREATE TABLE ... INHERITS`, то предложение `GENERATED` будет автоматически копироваться из родительского столбца. Для команды `ALTER TABLE ... INHERIT` требуется, чтобы родительский и дочерний столбцы уже совпадали в отношении состояния генерации, но не требуется совпадения их генерирующих выражений.
  - Аналогично для секционированных таблиц, если в определении дочернего столбца нет предложения `GENERATED` в `CREATE TABLE ... INHERITS`, то его предложение `GENERATED` будет автоматически копироваться у родительского столбца. Для команды `ALTER TABLE ... ATTACH PARTITION` требуется, чтобы родительский и дочерний столбцы уже совпадали в отношении состояния генерации, но не требует совпадения их генерирующих выражений.
  - В случае множественного наследования, если один родительский столбец является генерируемым, все соответствующие ему столбцы в иерархии наследования также должны быть генерируемыми. Если генерирующее выражение не одинаковое для всех столбцов, то желаемое выражение для дочернего столбца должно быть указано явно.

Дополнительные замечания касаются использования генерируемых столбцов.

- Права доступа к генерируемым столбцам существуют отдельно от прав для нижележащих базовых столбцов. Поэтому их можно организовать так, чтобы определённый пользователь мог прочитать генерируемый столбец, но не нижележащие базовые столбцы.
- Генерируемые столбцы, в соответствии с концепцией, пересчитываются после выполнения триггеров `BEFORE`. Вследствие этого, в генерируемых столбцах будут отражаться изменения, производимые триггером `BEFORE` в базовых столбцах, а обращаться в коде такого триггера к генерируемым столбцам, напротив, нельзя.
- Генерируемые столбцы не участвуют в логической репликации и не могут быть указаны в списке столбцов `CREATE PUBLICATION`.

5.5. Ограничения

Типы данных сами по себе ограничивают множество данных, которые можно сохранить в таблице. Однако для многих приложений такие ограничения слишком грубые. Например, столбец, содержащий цену продукта, должен, вероятно, принимать только положительные значения. Но такого стандартного типа данных нет. Возможно, вы также захотите ограничить данные столбца по отношению к другим столбцам или строкам. Например, в таблице с информацией о товаре должна быть только одна строка с определённым кодом товара.

Для решения подобных задач SQL позволяет вам определять ограничения для столбцов и таблиц. Ограничения дают вам возможность управлять данными в таблицах так, как вы захотите. Если пользователь попытается сохранить в столбце значение, нарушающее ограничения, возникнет ошибка. Ограничения будут действовать, даже если это значение по умолчанию.

5.5.1. Ограничения-проверки

Ограничение-проверка — наиболее общий тип ограничений. В его определении вы можете указать, что значение данного столбца должно удовлетворять логическому выражению (проверке истинности). Например, цену товара можно ограничить положительными значениями так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

Как вы видите, ограничение определяется после типа данных, как и значение по умолчанию. Значения по умолчанию и ограничения могут указываться в любом порядке. Ограничение-проверка состоит из ключевого слова `CHECK`, за которым идёт выражение в скобках. Это выражение должно включать столбец, для которого задаётся ограничение, иначе оно не имеет большого смысла.

Вы можете также присвоить ограничению отдельное имя. Это улучшит сообщения об ошибках и позволит вам ссылаться на это ограничение, когда вам понадобится изменить его. Сделать это можно так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

То есть, чтобы создать именованное ограничение, напишите ключевое слово `CONSTRAINT`, а за ним идентификатор и собственно определение ограничения. (Если вы не определите имя ограничения таким образом, система выберет для него имя за вас.)

Ограничение-проверка может также ссылаться на несколько столбцов. Например, если вы храните обычную цену и цену со скидкой, так вы можете гарантировать, что цена со скидкой будет всегда меньше обычной:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

Первые два ограничения определяются похожим образом, но для третьего используется новый синтаксис. Оно не связано с определённым столбцом, а представлено отдельным элементом в списке. Определения столбцов и такие определения ограничений можно переставлять в произвольном порядке.

Про первые два ограничения можно сказать, что это ограничения столбцов, тогда как третье является ограничением таблицы, так как оно написано отдельно от определений столбцов. Ограничения столбцов также можно записать в виде ограничений таблицы, тогда как обратное не всегда возможно, так как подразумевается, что ограничение столбца ссылается только на связанный столбец. (Хотя Postgres Pro этого не требует, но для совместимости с другими СУБД лучше следовать этому правилу.) Ранее приведённый пример можно переписать и так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

Или даже так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

Это дело вкуса.

Ограничениям таблицы можно присваивать имена так же, как и ограничениям столбцов:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

Следует заметить, что ограничение-проверка удовлетворяется, если выражение принимает значение true или NULL. Так как результатом многих выражений с операндами NULL будет значение NULL, такие ограничения не будут препятствовать записи NULL в ограничиваемые столбцы. Чтобы гарантировать, что столбец не содержит значения NULL, можно использовать ограничение NOT NULL, описанное в следующем разделе.

Примечание

Postgres Pro не поддерживает ограничения `CHECK`, которые обращаются к данным, не относящимся к новой или изменённой строке. Хотя ограничение `CHECK`, нарушающее это правило, может работать в простых случаях, в общем случае нельзя гарантировать, что база данных не придёт в состояние, когда условие ограничения окажется ложным (вследствие последующих изменений других участвующих в его вычислении строк). В результате восстановление выгруженных данных может оказаться невозможным. Во время восстановления возможен сбой, даже если полное состояние базы данных согласуется с условием ограничения, по причине того, что строки загружаются не в том порядке, в котором это условие будет соблюдаться. Поэтому для определения ограничений, затрагивающих другие строки и другие таблицы, используйте ограничения `UNIQUE`, `EXCLUDE` или `FOREIGN KEY`, если это возможно.

Если вам не нужна постоянно поддерживаемая гарантия целостности, а достаточно разовой проверки добавляемой строки по отношению к другим строкам, вы можете реализовать эту проверку в собственном триггере. (Этот подход исключает вышеописанные проблемы при восстановлении, так как в выгрузке pg_dump триггеры воссоздаются после восстановления данных, и поэтому эта проверка не будет действовать в процессе выгрузки/восстановления.)

Примечание

В Postgres Pro предполагается, что условия ограничений `CHECK` являются постоянными, то есть при одинаковых данных в строке они всегда выдают одинаковый результат. Именно этим предположением оправдывается то, что ограничения `CHECK` проверяются только при добавлении или изменении строк, а не при каждом обращении к ним. (Приведённое выше предупреждение о недопустимости обращений к другим таблицам является частным следствием этого предположения.)

Однако это предположение может нарушаться, как часто бывает, когда в выражении `CHECK` используется пользовательская функция, поведение которой впоследствии меняется. Postgres Pro не запрещает этого, и если строки в таблице перестанут удовлетворять ограничению `CHECK`, это останется незамеченным. В итоге при попытке загрузить выгруженные позже данные могут возникнуть проблемы. Поэтому подобные изменения рекомендуется осуществлять следующим образом: удалить ограничение (используя `ALTER TABLE`), изменить определение функции, а затем пересоздать ограничение той же командой, которая при этом перепроверит все строки таблицы.

5.5.2. Ограничения NOT NULL

Ограничение NOT NULL просто указывает, что столбцу нельзя присваивать значение NULL. Пример синтаксиса:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

Ограничение NOT NULL всегда записывается как ограничение столбца и функционально эквивалентно ограничению `CHECK (имя_столбца IS NOT NULL)`, но в Postgres Pro явное ограничение NOT NULL работает более эффективно. Хотя у такой записи есть недостаток — назначить имя таким ограничениям нельзя.

Естественно, для столбца можно определить больше одного ограничения. Для этого их нужно просто указать одно за другим:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

Порядок здесь не имеет значения, он не обязательно соответствует порядку проверки ограничений.

Для ограничения `NOT NULL` есть и обратное: ограничение `NULL`. Оно не означает, что столбец должен иметь только значение NULL, что конечно было бы бессмысленно. Суть же его в простом указании, что столбец может иметь значение NULL (это поведение по умолчанию). Ограничение `NULL` отсутствует в стандарте SQL и использовать его в переносимых приложениях не следует. (Оно было добавлено в Postgres Pro только для совместимости с некоторыми другими СУБД.) Однако некоторые пользователи любят его использовать, так как оно позволяет легко переключать ограничения в скрипте. Например, вы можете начать с:

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

и затем вставить ключевое слово `NOT`, где потребуется.

Подсказка

При проектировании баз данных чаще всего большинство столбцов должны быть помечены как NOT NULL.

5.5.3. Ограничения уникальности

Ограничения уникальности гарантируют, что данные в определённом столбце или группе столбцов уникальны среди всех строк таблицы. Ограничение записывается так:

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

в виде ограничения столбца и так:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

в виде ограничения таблицы.

Чтобы определить ограничение уникальности для группы столбцов, запишите его в виде ограничения таблицы, перечислив имена столбцов через запятую:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

Такое ограничение указывает, что сочетание значений перечисленных столбцов должно быть уникально во всей таблице, тогда как значения каждого столбца по отдельности не должны быть (и обычно не будут) уникальными.

Вы можете назначить уникальному ограничению имя обычным образом:

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

При добавлении ограничения уникальности будет автоматически создан уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в ограничении. Условие уникальности, распространяющееся только на некоторые строки, нельзя записать в виде ограничения уникальности, однако такое условие можно установить, создав уникальный частичный индекс.

Как правило, ограничение уникальности нарушается, если в таблице оказывается несколько строк, у которых совпадают значения всех столбцов, включённых в ограничение. По умолчанию два значения NULL при таком сравнении не считаются равными. Это означает, что даже при наличии ограничения уникальности в таблице можно сохранить строки с дублирующимися значениями, если они содержат NULL в одном или нескольких ограничиваемых столбцах. Это поведение можно изменить, добавив предложение `NULLS NOT DISTINCT`, например

```
CREATE TABLE products (
    product_no integer UNIQUE NULLS NOT DISTINCT,
    name text,
    price numeric
);
```

или

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE NULLS NOT DISTINCT (product_no)
);
```

Поведение по умолчанию можно выбрать явно, добавив `NULLS DISTINCT`. Согласно стандарту SQL вариант обработки значений NULL по умолчанию определяется реализацией, и в других СУБД встречается другое поведение. Это следует учитывать при разработке переносимых приложений.

5.5.4. Первичные ключи

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов может быть уникальным идентификатором строк в таблице. Для этого требуется, чтобы значения были одновременно уникальными и отличными от NULL. Таким образом, таблицы со следующими двумя определениями будут принимать одинаковые данные:

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Первичные ключи могут включать несколько столбцов; синтаксис похож на запись ограничений уникальности:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

При добавлении первичного ключа автоматически создаётся уникальный индекс-B-дерево для столбца или группы столбцов, перечисленных в первичном ключе, и данные столбцы помечаются как `NOT NULL`.

Таблица может иметь максимум один первичный ключ. (Ограничений уникальности и ограничений NOT NULL, которые функционально почти равнозначны первичным ключам, может быть сколько угодно, но назначить ограничением первичного ключа можно только одно.) Теория реляционных баз данных говорит, что первичный ключ должен быть в каждой таблице. В Postgres Pro такого жёсткого требования нет, но обычно лучше ему следовать.

Первичные ключи полезны и для документирования, и для клиентских приложений. Например, графическому приложению с возможностями редактирования содержимого таблицы, вероятно, потребуется знать первичный ключ таблицы, чтобы однозначно идентифицировать её строки. Первичные ключи находят и другое применение в СУБД; в частности, первичный ключ в таблице определяет целевые столбцы по умолчанию для сторонних ключей, ссылающихся на эту таблицу.

5.5.5. Внешние ключи

Ограничение внешнего ключа указывает, что значения столбца (или группы столбцов) должны соответствовать значениям в некоторой строке другой таблицы. Это называется *ссылочной целостностью* двух связанных таблиц.

Пусть у вас уже есть таблица продуктов, которую мы неоднократно использовали ранее:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Давайте предположим, что у вас есть таблица с заказами этих продуктов. Мы хотим, чтобы в таблице заказов содержались только заказы действительно существующих продуктов. Поэтому мы определим в ней ограничение внешнего ключа, ссылающееся на таблицу продуктов:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

С таким ограничением создать заказ со значением `product_no`, отсутствующим в таблице products (и не равным NULL), будет невозможно.

В такой схеме таблицу orders называют *подчинённой* таблицей, а products — *главной*. Соответственно, столбцы называют так же подчинённым и главным (или ссылающимся и целевым).

Предыдущую команду можно сократить так:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

то есть, если опустить список столбцов, внешний ключ будет неявно связан с первичным ключом главной таблицы.

Ограничению внешнего ключа можно назначить имя стандартным способом.

Внешний ключ также может ссылаться на группу столбцов. В этом случае его нужно записать в виде обычного ограничения таблицы. Например:

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Естественно, число и типы столбцов в ограничении должны соответствовать числу и типам целевых столбцов.

Иногда имеет смысл задать в ограничении внешнего ключа в качестве «другой таблицы» ту же таблицу; такой внешний ключ называется *ссылающимся на себя*. Например, если вы хотите, чтобы строки таблицы представляли узлы древовидной структуры, вы можете написать

```
CREATE TABLE tree (
    node_id integer PRIMARY KEY,
    parent_id integer REFERENCES tree,
    name text,
    ...
);
```

Для узла верхнего уровня `parent_id` будет равен NULL, пока записи с отличным от NULL `parent_id` будут ссылаться только на существующие строки таблицы.

Таблица может содержать несколько ограничений внешнего ключа. Это полезно для связи таблиц в отношении многие-ко-многим. Скажем, у вас есть таблицы продуктов и заказов, но вы хотите, чтобы один заказ мог содержать несколько продуктов (что невозможно в предыдущей схеме). Для этого вы можете использовать такую схему:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Заметьте, что в последней таблице первичный ключ покрывает внешние ключи.

Мы знаем, что внешние ключи запрещают создание заказов, не относящихся ни к одному продукту. Но что делать, если после создания заказов с определённым продуктом мы захотим удалить его? SQL справится с этой ситуацией. Интуиция подсказывает следующие варианты поведения:

- Запретить удаление продукта
- Удалить также связанные заказы
- Что-то ещё?

Для иллюстрации давайте реализуем следующее поведение в вышеприведённом примере: при попытке удаления продукта, на который ссылаются заказы (через таблицу `order_items`), мы запрещаем эту операцию. Если же кто-то попытается удалить заказ, то удалится и его содержимое:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
```

```
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Ограничивающие и каскадные удаления — два наиболее распространённых варианта. `RESTRICT` предотвращает удаление связанной строки. `NO ACTION` означает, что если зависимые строки продолжают существовать при проверке ограничения, возникает ошибка (это поведение по умолчанию). (Главным отличием этих двух вариантов является то, что `NO ACTION` позволяет отложить проверку в процессе транзакции, а `RESTRICT` — нет.) `CASCADE` указывает, что при удалении связанных строк зависимые от них будут так же автоматически удалены. Есть ещё два варианта: `SET NULL` и `SET DEFAULT`. При удалении связанных строк они назначают зависимым столбцам в подчинённой таблице значения NULL или значения по умолчанию, соответственно. Заметьте, что это не будет основанием для нарушения ограничений. Например, если в качестве действия задано `SET DEFAULT`, но значение по умолчанию не удовлетворяет ограничению внешнего ключа, операция закончится ошибкой.

Какой вариант действия `ON DELETE` выбрать — зависит от того, какие типы объектов представляются в связанных таблицах. Когда в подчинённой таблице представляется объект, который является составной частью сущности, представленной в главной таблице, и не может существовать независимо, уместно выбрать действие `CASCADE`. Если две таблицы представляют независимые объекты, более подходящим действием будет `RESTRICT` или `NO ACTION`; тогда приложению, которому действительно нужно удалить оба объекта, потребуется сделать это явно и выполнить две команды удаления. В приведённом выше примере позиции заказа являются частью заказа, и будет удобно, если они удалятся автоматически при удалении заказа. Но продукты и заказы — разные вещи, поэтому автоматическое удаление некоторых позиций заказов при удалении продуктов может быть неприемлемым. Если же отношение внешнего ключа представляет необязательную информацию, подходящим может быть действие `SET NULL` или `SET DEFAULT`. Например, если в таблице продуктов содержится ссылка на менеджера продукта, и запись менеджера удаляется, может быть полезным установить в поле менеджера продукта значение NULL или значение по умолчанию.

Действия `SET NULL` и `SET DEFAULT` могут принимать список столбцов, которые получат соответствующие значения. Обычно значения присваиваются всем столбцам ограничения внешнего ключа; присвоение для подмножества столбцов полезно только в некоторых особых случаях. Рассмотрите следующий пример:

```
CREATE TABLE tenants (
    tenant_id integer PRIMARY KEY
);

CREATE TABLE users (
    tenant_id integer REFERENCES tenants ON DELETE CASCADE,
    user_id integer NOT NULL,
    PRIMARY KEY (tenant_id, user_id)
);

CREATE TABLE posts (
    tenant_id integer REFERENCES tenants ON DELETE CASCADE,
    post_id integer NOT NULL,
    author_id integer,
    PRIMARY KEY (tenant_id, post_id),
    FOREIGN KEY (tenant_id, author_id) REFERENCES users ON DELETE SET NULL (author_id)
);
```

Без указания столбца внешний ключ также установил бы значение NULL для столбца `tenant_id`, но этот столбец является частью первичного ключа.

Аналогично указанию `ON DELETE` существует `ON UPDATE`, которое срабатывает при изменении заданного столбца. При этом возможные действия те же, за исключением того, что для `SET NULL` и `SET DEFAULT` нельзя задать списки столбцов. `CASCADE` в данном случае означает, что изменённые значения связанных столбцов будут скопированы в зависимые строки.

Обычно зависимая строка не должна удовлетворять ограничению внешнего ключа, если один из связанных столбцов содержит NULL. Если в объявление внешнего ключа добавлено `MATCH FULL`, строка будет удовлетворять ограничению, только если все связанные столбцы равны NULL (то есть при разных значениях (NULL и не NULL) гарантируется невыполнение ограничения `MATCH FULL`). Если вы хотите, чтобы зависимые строки не могли избежать и этого ограничения, объявите связанные столбцы как `NOT NULL`.

Внешний ключ должен ссылаться на столбцы, либо являющиеся первичным ключом, либо образующие ограничение уникальности, либо являющиеся столбцами из нечастичного уникального индекса. Таким образом, для связанных столбцов всегда будет существовать индекс, а значит проверки наличия соответствия для связанной строки будут выполняться эффективно. Так как команды `DELETE` для строк главной таблицы или `UPDATE` для зависимых столбцов потребуют просканировать подчинённую таблицу и найти строки, ссылающиеся на старые значения, полезно будет иметь индекс и для подчинённых столбцов. Но это нужно не всегда, и создать соответствующий индекс можно по-разному, поэтому объявление внешнего ключа не создаёт автоматически индекс по связанным столбцам.

Подробнее об изменении и удалении данных рассказывается в Главе 6. Вы также можете подробнее узнать о синтаксисе ограничений внешнего ключа в справке CREATE TABLE.

5.5.6. Ограничения-исключения

Ограничения-исключения гарантируют, что при сравнении любых двух строк по указанным столбцам или выражениям с помощью заданных операторов, минимум одно из этих сравнений возвратит false или NULL. Записывается это так:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Подробнее об этом см. `CREATE TABLE ... CONSTRAINT ... EXCLUDE`.

При добавлении ограничения-исключения будет автоматически создан индекс того типа, который указан в объявлении ограничения.

5.6. Системные столбцы

В каждой таблице есть несколько *системных столбцов*, неявно определённых системой. Как следствие, их имена нельзя использовать в качестве имён пользовательских столбцов. (Заметьте, что это не зависит от того, является ли имя ключевым словом или нет; заключение имени в кавычки не поможет избежать этого ограничения.) Эти столбцы не должны вас беспокоить, вам лишь достаточно знать об их существовании.

`tableoid`

Идентификатор объекта для таблицы, содержащей строку. Этот столбец особенно полезен для запросов, имеющих дело с секционированными таблицами (см. Раздел 5.12) или иерархией наследования (см. Раздел 5.11), так как без него сложно определить, из какой таблицы выбрана строка. Связав `tableoid` со столбцом `oid` в таблице `pg_class`, можно будет получить имя таблицы.

`xmin`

Идентификатор (код) транзакции, добавившей строку этой версии. (Версия строки — это её индивидуальное состояние; при каждом изменении создаётся новая версия одной и той же логической строки.)

`cmin`

Номер команды (начиная с нуля) внутри транзакции, добавившей строку.

`xmax`

Идентификатор транзакции, удалившей строку, или 0 для неудалённой версии строки. Значение этого столбца может быть ненулевым и для видимой версии строки. Это обычно означает, что удаляющая транзакция ещё не была зафиксирована, или удаление было отменено.

`cmax`

Номер команды в удаляющей транзакции или ноль.

`ctid`

Физическое расположение данной версии строки в таблице. Заметьте, что хотя по `ctid` можно очень быстро найти версию строки, значение `ctid` изменится при выполнении `VACUUM FULL`. Таким образом, `ctid` нельзя применять в качестве долгосрочного идентификатора строки. Для идентификации логических строк следует использовать первичный ключ.

Идентификаторы транзакций также являются 32-битными. В долгоживущей базе данных они могут пойти по кругу. Это не критично при правильном обслуживании БД; подробнее об этом см. Главу 23. Однако полагаться на уникальность кодов транзакций в течение длительного времени (при более чем миллиарде транзакций) не следует.

Идентификаторы команд также 32-битные. Это создаёт жёсткий лимит на 2^{32} (4 миллиарда) команд SQL в одной транзакции. На практике это не проблема — заметьте, что это лимит числа команд SQL, а не количества обрабатываемых строк. Кроме того, идентификатор получают только те команды, которые фактически изменяют содержимое базы данных.

5.7. Изменение таблиц

Если вы создали таблицы, а затем поняли, что допустили ошибку, или изменились требования вашего приложения, вы можете удалить её и создать заново. Но это будет неудобно, если таблица уже заполнена данными, или если на неё ссылаются другие объекты базы данных (например, по внешнему ключу). Поэтому Postgres Pro предоставляет набор команд для модификации таблиц. Заметьте, что это по сути отличается от изменения данных, содержащихся в таблице: здесь мы обсуждаем модификацию определения, или структуры, таблицы.

Вы можете:

- Добавлять столбцы
- Удалять столбцы
- Добавлять ограничения
- Удалять ограничения
- Изменять значения по умолчанию
- Изменять типы столбцов
- Переименовывать столбцы
- Переименовывать таблицы

Все эти действия выполняются с помощью команды ALTER TABLE; подробнее о ней вы можете узнать в её справке.

5.7.1. Добавление столбца

Добавить столбец вы можете так:

```
ALTER TABLE products ADD COLUMN description text;
```

Новый столбец заполняется заданным для него значением по умолчанию (или значением NULL, если вы не добавите указание `DEFAULT`).

> ### Подсказка
>
> Начиная с Postgres Pro 11, добавление столбца с постоянным значением по умолчанию более не означает, что при выполнении команды `ALTER TABLE` будут изменены все строки таблицы. Вместо этого установленное значение по умолчанию будет просто выдаваться при следующем обращении к строкам, а сохранится в строках при перезаписи таблицы. Благодаря этому операция `ALTER TABLE` и с большими таблицами выполняется очень быстро.
>
> Однако если значение по умолчанию изменчивое (например, это `clock_timestamp()`), в каждую строку нужно будет внести значение, вычисленное в момент выполнения `ALTER TABLE`. Чтобы избежать потенциально длительной операции изменения всех строк, если вы планируете заполнить столбец в основном не значениями по умолчанию, лучше будет добавить столбец без значения по умолчанию, затем вставить требуемые значения с помощью `UPDATE`, а потом определить значение по умолчанию, как описано ниже.

При этом вы можете сразу определить ограничения столбца, используя обычный синтаксис:

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

На самом деле здесь можно использовать все конструкции, допустимые в определении столбца в команде `CREATE TABLE`. Помните однако, что значение по умолчанию должно удовлетворять данным ограничениям, чтобы операция `ADD` выполнилась успешно. Вы также можете сначала заполнить столбец правильно, а затем добавить ограничения (см. ниже).

5.7.2. Удаление столбца

Удалить столбец можно так:

```
ALTER TABLE products DROP COLUMN description;
```

Данные, которые были в этом столбце, исчезают. Вместе со столбцом удаляются и включающие его ограничения таблицы. Однако если на столбец ссылается ограничение внешнего ключа другой таблицы, Postgres Pro не удалит это ограничение неявно. Разрешить удаление всех зависящих от этого столбца объектов можно, добавив указание `CASCADE`:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

Общий механизм, стоящий за этим, описывается в Разделе 5.15.

5.7.3. Добавление ограничения

Для добавления ограничения используется синтаксис ограничения таблицы. Например:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

Чтобы добавить ограничение NOT NULL, которое нельзя записать в виде ограничения таблицы, используйте такой синтаксис:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

Ограничение проходит проверку автоматически и будет добавлено, только если ему удовлетворяют данные таблицы.

5.7.4. Удаление ограничения

Для удаления ограничения вы должны знать его имя. Если вы не присваивали ему имя, это неявно сделала система, и вы должны выяснить его. Здесь может быть полезна команда psql `\d` *`имя_таблицы`* (или другие программы, показывающие подробную информацию о таблицах). Зная имя, вы можете использовать команду:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

Как и при удалении столбца, если вы хотите удалить ограничение с зависимыми объектами, добавьте указание `CASCADE`. Примером такой зависимости может быть ограничение внешнего ключа, связанное со столбцами ограничения первичного ключа.

Так можно удалить ограничения любых типов, кроме NOT NULL. Чтобы удалить ограничение NOT NULL, используйте команду:

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(Вспомните, что у ограничений NOT NULL нет имён.)

5.7.5. Изменение значения по умолчанию

Назначить столбцу новое значение по умолчанию можно так:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Заметьте, что это никак не влияет на существующие строки таблицы, а просто задаёт значение по умолчанию для последующих команд `INSERT`.

Чтобы удалить значение по умолчанию, выполните:

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

При этом по сути значению по умолчанию просто присваивается NULL. Как следствие, ошибки не будет, если вы попытаетесь удалить значение по умолчанию, не определённое явно, так как неявно оно существует и равно NULL.

5.7.6. Изменение типа данных столбца

Чтобы преобразовать столбец в другой тип данных, используйте команду:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

Она будет успешна, только если все существующие значения в столбце могут быть неявно приведены к новому типу. Если требуется более сложное преобразование, вы можете добавить указание `USING`, определяющее, как получить новые значения из старых.

Postgres Pro попытается также преобразовать к новому типу значение столбца по умолчанию (если оно определено) и все связанные с этим столбцом ограничения. Но преобразование может оказаться неправильным, и тогда вы получите неожиданные результаты. Поэтому обычно лучше удалить все ограничения столбца, перед тем как менять его тип, а затем воссоздать модифицированные должным образом ограничения.

5.7.7. Переименование столбца

Чтобы переименовать столбец, выполните:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.7.8. Переименование таблицы

Таблицу можно переименовать так:

```
ALTER TABLE products RENAME TO items;
```

5.8. Права

Когда в базе данных создаётся объект, ему назначается владелец. Владельцем обычно становится роль, с которой был выполнен оператор создания. Для большинства типов объектов в исходном состоянии только владелец (или суперпользователь) может делать с объектом всё, что угодно. Чтобы разрешить использовать его другим ролям, нужно дать им *права*.

Существует несколько типов прав: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, `USAGE`, `SET`, `ALTER SYSTEM` и `MAINTAIN`. Набор прав, применимых к определённому объекту, зависит от типа объекта (таблица, функция и т. д.). Более подробно назначение этих прав описывается ниже. Как применяются эти права, вы также увидите в следующих разделах и главах.

Право изменять или удалять объект является неотъемлемым правом владельца объекта, его нельзя лишиться или передать другому. (Однако как и все другие права, это право наследуют члены роли-владельца, см. Раздел 20.3.)

Объекту можно назначить нового владельца с помощью команды `ALTER` для соответствующего типа объекта, например:

```
ALTER TABLE имя_таблицы OWNER TO новый_владелец;
```

Суперпользователь может делать это без ограничений, а обычный пользователь — только если он является одновременно текущим владельцем объекта (или наследует права роли владельца) и может выполнять `SET ROLE` для новой роли владельца.

Для назначения прав применяется команда GRANT. Например, если в базе данных есть роль `joe` и таблица `accounts`, право на изменение таблицы можно дать этой роли так:

```
GRANT UPDATE ON accounts TO joe;
```

Если вместо конкретного права написать `ALL`, роль получит все права, применимые для объекта этого типа.

Для назначения права всем ролям в системе можно использовать специальное имя «роли»: `PUBLIC`. Также для упрощения управления ролями, когда в базе данных есть множество пользователей, можно настроить «групповые» роли; подробнее об этом см. Главу 20.

Чтобы лишить пользователей ранее данных им прав, используйте команду REVOKE:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Обычно распоряжаться правами может только владелец объекта (или суперпользователь). Однако возможно дать право доступа к объекту «с правом передачи», что позволит получившему такое право назначать его другим. Если такое право передачи впоследствии будет отозвано, то все, кто получил данное право доступа (непосредственно или по цепочке передачи), потеряют его. Подробнее об этом см. справку GRANT и REVOKE.

Владелец объекта может лишить себя обычных прав, например запретить не только всем остальным, но и себе, вносить изменения в таблицу. Однако владельцы всегда имеют возможность управлять правами, так что они могут в любом случае вернуть себе права, которых лишились.

Все существующие права перечислены ниже:

`SELECT`

Позволяет выполнять `SELECT` для любого столбца или перечисленных столбцов в заданной таблице, представлении, матпредставлении или другом объекте табличного вида. Также позволяет выполнять `COPY TO`. Помимо этого, данное право требуется для обращения к существующим значениям столбцов в `UPDATE`, `DELETE` или `MERGE`. Для последовательностей это право позволяет пользоваться функцией `currval`. Для больших объектов оно позволяет читать содержимое объекта.

`INSERT`

Позволяет вставлять с помощью `INSERT` строки в заданную таблицу, представление и т. п. Может назначаться для отдельных столбцов; в этом случае только этим столбцам можно присваивать значения в команде `INSERT` (другие столбцы получат значения по умолчанию). Также позволяет выполнять `COPY FROM`.

`UPDATE`

Позволяет изменять с помощью `UPDATE` данные во всех, либо только перечисленных, столбцах в заданной таблице, представлении и т. п. (На практике для любой нетривиальной команды

`UPDATE` потребуется и право `SELECT`, так как она должна обратиться к столбцам таблицы, чтобы определить, какие строки подлежат изменению, и/или вычислить новые значения столбцов.) Для `SELECT ... FOR UPDATE` и `SELECT ... FOR SHARE` также требуется иметь это право как минимум для одного столбца, помимо права `SELECT`. Для последовательностей это право позволяет пользоваться функциями `nextval` и `setval`. Для больших объектов это право позволяет записывать данные в объект или обрезать его.

`DELETE`

Позволяет удалять с помощью команды `DELETE` строки из таблицы, представления и т. п. (На практике для любой нетривиальной команды `DELETE` потребуется также право `SELECT`, так как она должна обратиться к колонкам таблицы, чтобы определить, какие строки подлежат удалению.)

`TRUNCATE`

Позволяет опустошать таблицу с помощью `TRUNCATE`.

`REFERENCES`

Позволяет создавать ограничение внешнего ключа, обращающееся к таблице или определённым столбцам таблицы.

`TRIGGER`

Позволяет создавать триггер для таблицы, представления и т. п.

`CREATE`

Для баз данных это право позволяет создавать схемы и публикации, а также устанавливать доверенные расширения в конкретной базе.

Для схем это право позволяет создавать новые объекты в заданной схеме. Чтобы переименовать существующий объект, необходимо быть его владельцем *и* иметь это право для схемы, содержащей его.

Для табличных пространств это право позволяет создавать таблицы, индексы и временные файлы в определённом табличном пространстве, а также создавать базы данных, для которых это пространство будет основным.

Заметьте, что факт лишения пользователя этого права не влияет на существование или размещение существующих объектов.

`CONNECT`

Позволяет подключаться к базе данных. Это право проверяется при установлении соединения (в дополнение к условиям, определённым в конфигурации `pg_hba.conf`).

`TEMPORARY`

Позволяет создавать временные таблицы в определённой базе данных.

`EXECUTE`

Позволяет вызывать функцию или процедуру, в том числе использовать любые операторы, реализованные данной функцией. Это единственный тип прав, применимый к функциям и процедурам.

`USAGE`

Для процедурных языков это право позволяет создавать функции на определённом языке. Это единственный тип прав, применимый к процедурным языкам.

Для схем это право даёт доступ к содержащимся в них объектам (предполагается, что при этом имеются права, необходимые для доступа к самим объектам). По сути это право позволяет субъекту «просматривать» объекты внутри схемы. Без этого разрешения имена объектов всё же можно будет узнать, например, обратившись к системным каталогам. Кроме того, если отозвать это право, в существующих сеансах могут оказаться операторы, для которых просмотр имён объектов был выполнен ранее, так что это право не позволяет абсолютно надёжно перекрыть доступ к объектам.

Для последовательностей это право позволяет использовать функции `currval` и `nextval`.

Для типов и доменов это право позволяет использовать заданный тип или домен при создании таблиц, функций или других объектов схемы. (Заметьте, что это право не ограничивает общее «использование» типа, например обращение к значениям типа в запросах. Без этого права нельзя только создавать объекты, зависящие от заданного типа. Основное предназначение этого права в том, чтобы ограничить круг пользователей, способных создавать зависимости от типа, которые могут впоследствии помешать владельцу типа изменить его.)

Для обёрток сторонних данных это право позволяет создавать использующие их определения сторонних серверов.

Для сторонних серверов это право позволяет создавать использующие их сторонние таблицы. Наделённые этим правом могут также создавать, модифицировать или удалять собственные сопоставления пользователей, связанные с определённым сервером.

`SET`

Позволяет задать для параметра конфигурации сервера новое значение в текущем сеансе. (Хотя это право может быть предоставлено для любого параметра, оно имеет смысл только для тех параметров, для которых обычно требуются права суперпользователя.)

`ALTER SYSTEM`

Позволяет присваивать значение параметру конфигурации сервера, используя команду ALTER SYSTEM.

`MAINTAIN`

Позволяет выполнять `VACUUM`, `ANALYZE`, `CLUSTER`, `REFRESH MATERIALIZED VIEW`, `REINDEX` и `LOCK TABLE` с отношением.

Права, требующиеся для других команд, указаны на страницах справки этих команд.

Postgres Pro по умолчанию назначает роли `PUBLIC` права для некоторых типов объектов, когда эти объекты создаются. Для таблиц, столбцов, последовательностей, обёрток сторонних данных, сторонних серверов, больших объектов, схем, табличных пространств или параметров конфигурации `PUBLIC` по умолчанию никаких прав не получает. Для других типов объектов `PUBLIC` получает следующие права по умолчанию: `CONNECT` и `TEMPORARY` (создание временных таблиц) для баз данных; `EXECUTE` — для функций и процедур; `USAGE` — для языков и типов данных (включая домены). Владелец объекта, конечно же, может отозвать (посредством `REVOKE`) как явно назначенные права, так и права по умолчанию. (Для максимальной безопасности команду `REVOKE` нужно выполнять в транзакции, создающей объект; тогда не образуется окно, в котором другой пользователь сможет обратиться к объекту.) Кроме того, эти изначально назначаемые права по умолчанию можно переопределить, воспользовавшись командой ALTER DEFAULT PRIVILEGES.

В Таблице 5.1 показаны однобуквенные сокращения, которыми обозначаются эти права в списках *ACL* (Access Control List, Список контроля доступа). Вы увидите эти сокращения в выводе перечисленных ниже команд psql или в столбцах ACL в системных каталогах.

Таблица 5.1. Сокращённые обозначения прав в ACL

Право	Сокращение	Применимые типы объектов
`SELECT`	`r` («read», чтение)	`LARGE OBJECT`, `SEQUENCE`, `TABLE` (и объекты, подобным таблицам), столбец таблицы

Право	**Сокращение**	**Применимые типы объектов**
`INSERT`	`a` («append», добавление)	`TABLE`, столбец таблицы
`UPDATE`	`w` («write», запись)	`LARGE OBJECT`, `SEQUENCE`, `TABLE`, столбец таблицы
`DELETE`	`d`	`TABLE`
`TRUNCATE`	`D`	`TABLE`
`REFERENCES`	`x`	`TABLE`, столбец таблицы
`TRIGGER`	`t`	`TABLE`
`CREATE`	`C`	`DATABASE`, `SCHEMA`, `TABLESPACE`
`CONNECT`	`c`	`DATABASE`
`TEMPORARY`	`T`	`DATABASE`
`EXECUTE`	`X`	`FUNCTION`, `PROCEDURE`
`USAGE`	`U`	`DOMAIN`, `FOREIGN DATA WRAPPER`, `FOREIGN SERVER`, `LANGUAGE`, `SCHEMA`, `SEQUENCE`, `TYPE`
`SET`	`s`	`PARAMETER`
`ALTER SYSTEM`	`A`	`PARAMETER`
`MAINTAIN`	`m`	`TABLE`

В Таблица 5.2 для каждого типа SQL-объекта показаны относящиеся к нему права, с использованием приведённых выше сокращений. Также в ней для каждого типа приведена команда psql, которая позволяет узнать, какие права назначены для объекта этого типа.

Таблица 5.2. Сводка прав доступа

Тип объекта	**Все права**	**Права `PUBLIC` по умолчанию**	**Команда psql**
`DATABASE`	`CTc`	`Tc`	`\l`
`DOMAIN`	`U`	`U`	`\dD+`
`FUNCTION` или `PROCEDURE`	`X`	`X`	`\df+`
`FOREIGN DATA WRAPPER`	`U`	нет	`\dew+`
`FOREIGN SERVER`	`U`	нет	`\des+`
`LANGUAGE`	`U`	`U`	`\dL+`
`LARGE OBJECT`	`rw`	нет	`\dl+`
`PARAMETER`	`sA`	нет	`\dconfig+`
`SCHEMA`	`UC`	нет	`\dn+`
`SEQUENCE`	`rwU`	нет	`\dp`
`TABLE` (и объекты, подобные таблицам)	`arwdDxtm`	нет	`\dp`
Столбец таблицы	`arwx`	нет	`\dp`
`TABLESPACE`	`C`	нет	`\db+`
`TYPE`	`U`	`U`	`\dT+`

Права, назначенные для определённого объекта, выводятся в виде списка элементов `aclitem`, где каждый `aclitem` имеет такой формат:

```
правообладатель=аббревиатура-права[*].../праводатель
```

Каждый `aclitem` содержит разрешения, которые были предоставлены субъекту определённым праводателем. Конкретные права представлены однобуквенными аббревиатурами из таблицы Таблица 5.1 с добавлением `*`, если право было выдано с параметром передачи. Например, запись `calvin=r*w/hobbes` означает, что роль `calvin` имеет право `SELECT` (`r`) с возможностью передачи (`*`), а также непередаваемое право `UPDATE` (`w`), и оба эти права даны ему ролью `hobbes`. Если `calvin` имеет для этого объекта и другие права, предоставленные ему другим праводателем, они выводятся в отдельном элементе `aclitem`. Пустое поле правообладателя в `aclitem` соответствует роли `PUBLIC`.

Например, предположим, что пользователь `miriam` создаёт таблицы `mytable` и выполняет:

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

Тогда команда `\dp` в psql покажет:

```
=> \dp mytable
                                  Access privileges
 Schema |  Name   | Type  |   Access privileges    |   Column privileges   | Policies
--------+---------+-------+------------------------+-----------------------+----------
 public | mytable | table | miriam=arwdDxtm/miriam+| col1:                +|
        |         |       | =r/miriam             +|   miriam_rw=rw/miriam |
        |         |       | admin=arw/miriam       |                       |
(1 row)
```

Если столбец «Права доступа» (Access privileges) для данного объекта пуст, это значит, что для объекта действуют стандартные права (то есть запись прав в соответствующем каталоге содержит NULL). Права по умолчанию всегда включают все права для владельца и могут также включать некоторые права для `PUBLIC` в зависимости от типа объекта, как разъяснялось выше. Первая команда `GRANT` или `REVOKE` для объекта приводит к созданию записи прав по умолчанию (например, `{miriam=arwdDxt/miriam}`), а затем изменяет эту запись в соответствии с заданным запросом. Подобным образом, строки, показанные в столбце «Права доступа к столбцам» (Column privileges), выводятся только для столбцов с нестандартными правами доступа. (Заметьте, что в данном контексте под «стандартными правами» всегда подразумевается встроенный набор прав, предопределённый для типа объекта. Если с объектом связан набор прав по умолчанию, полученный после изменения в результате `ALTER DEFAULT PRIVILEGES`, изменённые права будут всегда выводиться явно, показывая эффект команды `ALTER`.)

Заметьте, что право распоряжения правами, которое имеет владелец, не отмечается в выводимой сводке. Знаком `*` отмечаются только те права с правом передачи, которые были явно назначены кому-либо.

Столбец «Право доступа» показывает `(none)`, когда запись прав объекта существует, но пуста. Это означает, что никакие права не предоставлены вообще, даже владельцу объекта, хотя подобные ситуации возникают редко. (В таких случаях у владельца всё ещё есть неявные возможности выдать права, и поэтому он может снова выдать их себе, но в данный момент у него их нет.)

5.9. Политики защиты строк

В дополнение к стандартной системе прав SQL, управляемой командой GRANT, на уровне таблиц можно определить *политики защиты строк*, ограничивающие для пользователей наборы строк, которые могут быть возвращены обычными запросами или добавлены, изменены и удалены командами, изменяющими данные. Это называется также *защитой на уровне строк* (RLS, Row-Level Security). По умолчанию таблицы не имеют политик, так что если система прав SQL разрешает пользователю доступ к таблице, все строки в ней одинаково доступны для чтения или изменения.

Когда для таблицы включается защита строк (с помощью команды ALTER TABLE ... ENABLE ROW LEVEL SECURITY), все обычные запросы к таблице на выборку или модификацию строк должны разрешаться политикой защиты строк. (Однако на владельца таблицы такие политики обычно не действуют.) Если политика для таблицы не определена, применяется политика запрета по умолчанию, так что никакие строки в этой таблице нельзя увидеть или модифицировать. На операции с таблицей в целом, такие как `TRUNCATE` и `REFERENCES`, защита строк не распространяется.

Политики защиты строк могут применяться к определённым командам и/или ролям. Политику можно определить как применяемую к командам `ALL` (всем), либо `SELECT`, `INSERT`, `UPDATE` и `DELETE`. Кроме того, политику можно связать с несколькими ролями, при этом действуют обычные правила членства и наследования.

Чтобы определить, какие строки будут видимыми или могут изменяться в таблице, для политики задаётся выражение, возвращающее логический результат. Это выражение будет вычисляться для каждой строки перед другими условиями или функциями, поступающими из запроса пользователя. (Единственным исключением из этого правила являются `герметичные` функции, которые гарантированно не допускают утечки информации; оптимизатор может решить выполнить эти функции до проверок защиты строк.) Строки, для которых это выражение возвращает не `true`, обрабатываться не будут. Чтобы независимо управлять набором строк, которые можно видеть, и набором строк, которые можно модифицировать, в политике можно задать отдельные выражения. Выражения политик обрабатываются в составе запроса с правами исполняющего его пользователя, но для обращения к данным, недоступным этому пользователю, в этих выражениях могут применяться функции, определяющие контекст безопасности.

Суперпользователи и роли с атрибутом `BYPASSRLS` всегда обращаются к таблице, минуя систему защиты строк. На владельца таблицы защита строк тоже не действует, хотя он может включить её для себя принудительно, выполнив команду ALTER TABLE ... FORCE ROW LEVEL SECURITY.

Неотъемлемое право включать или отключать защиту строк, а также определять политики для таблицы, имеет только её владелец.

Для создания политик предназначена команда CREATE POLICY, для изменения — ALTER POLICY, а для удаления — DROP POLICY. Чтобы включить или отключить защиту строк для определённой таблицы, воспользуйтесь командой ALTER TABLE.

Каждой политике назначается имя, при этом для одной таблицы можно определить несколько политик. Так как политики привязаны к таблицам, каждая политика для таблицы должна иметь уникальное имя. В разных таблицах политики могут иметь одинаковые имена.

Когда к определённому запросу применяются несколько политик, они объединяются либо логическим сложением (если политики разрешительные (по умолчанию)), либо умножением (если политики ограничительные). Это подобно тому, как некоторая роль получает права всех ролей, в которые она включена. Разрешительные и ограничительные политики рассматриваются ниже.

В качестве простого примера, создать политику для отношения `account`, позволяющую только членам роли `managers` обращаться к строкам отношения и при этом только к своим, можно так:

```
CREATE TABLE accounts (manager text, company text, contact_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account_managers ON accounts TO managers
    USING (manager = current_user);
```

Эта политика неявно подразумевает и предложение `WITH CHECK`, идентичное предложению `USING`, поэтому указанное ограничение применяется и к строкам, выбираемым командой (так что один менеджер не может выполнить `SELECT`, `UPDATE` или `DELETE` для существующих строк, принадлежащих другому), и к строкам, изменяемым командой (так что командами `INSERT` и `UPDATE` нельзя создать строки, принадлежащие другому менеджеру).

Если роль не задана, либо задано специальное имя пользователя `PUBLIC`, политика применяется ко всем пользователям в данной системе. Чтобы все пользователи могли обратиться только к собственной строке в таблице `users`, можно применить простую политику:

```
CREATE POLICY user_policy ON users
    USING (user_name = current_user);
```

Это работает подобно предыдущему примеру.

Чтобы определить для строк, добавляемых в таблицу, отдельную политику, отличную от политики, ограничивающей видимые строки, можно скомбинировать несколько политик. Следующая пара политик позволит всем пользователям видеть все строки в таблице `users`, но изменять только свою собственную:

```
CREATE POLICY user_sel_policy ON users
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
    USING (user_name = current_user);
```

Для команды `SELECT` эти две политики объединяются операцией `OR`, так что в итоге это позволяет выбирать все строки. Для команд других типов применяется только вторая политика, и эффект тот же, что и раньше.

Защиту строк можно отключить так же командой `ALTER TABLE`. При отключении защиты, политики, определённые для таблицы, не удаляются, а просто игнорируются. В результате в таблице будут видны и могут модифицироваться все строки, с учётом ограничений стандартной системы прав SQL.

Ниже показан развёрнутый пример того, как этот механизм защиты можно применять в производственной среде. Таблица `passwd` имитирует файл паролей в Unix:

```
-- Простой пример на базе файла passwd
CREATE TABLE passwd (
  user_name              text UNIQUE NOT NULL,
  pwhash                text,
  uid                   int  PRIMARY KEY,
  gid                   int  NOT NULL,
  real_name             text NOT NULL,
  home_phone            text,
  extra_info            text,
  home_dir              text NOT NULL,
  shell                 text NOT NULL
);

CREATE ROLE admin;  -- Администратор
CREATE ROLE bob;    -- Обычный пользователь
CREATE ROLE alice;  -- Обычный пользователь

-- Наполнение таблицы
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh');

-- Необходимо включить для этой таблицы защиту на уровне строк
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- Создание политик
-- Администратор может видеть и добавлять любые строки
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);
-- Обычные пользователи могут видеть все строки
```

```
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- Обычные пользователи могут изменять собственные данные, но
-- не могут задать произвольную оболочку входа
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh')
  );

-- Администраторы получают все обычные права
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- Пользователям разрешается чтение только общедоступных столбцов
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info, home_dir, shell)
  ON passwd TO public;
-- Пользователям разрешается изменение определённых столбцов
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

Как и любые средства защиты, важно проверить политики, и убедиться в том, что они работают ожидаемым образом. Применительно к предыдущему примеру, эти команды показывают, что система разрешений работает корректно.

```
-- Администратор может видеть все строки и поля
postgres=> set role admin;
SET
postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  | extra_info | home_dir    |
   shell
-----------+--------+-----+-----+-----------+--------------+------------+-------------
+-----------
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |            | /root
 | /bin/dash
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |            | /home/bob
 | /bin/zsh
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |            | /home/alice
 | /bin/zsh
(3 rows)

-- Проверим, что может делать Алиса
postgres=> set role alice;
SET
postgres=> table passwd;
ERROR:  permission denied for table passwd
postgres=> select user_name,real_name,home_phone,extra_info,home_dir,shell from passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+-----------+--------------+------------+-------------+-----------
 admin     | Admin     | 111-222-3333 |            | /root       | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)

postgres=> update passwd set user_name = 'joe';
ERROR:  permission denied for table passwd
-- Алиса может изменить своё имя (поле real_name), но не имя кого-либо другого
postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
```

```
postgres=> update passwd set real_name = 'John Doe' where user_name = 'admin';
UPDATE 0
postgres=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
postgres=> delete from passwd;
ERROR:  permission denied for table passwd
postgres=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for table passwd
-- Алиса может изменить собственный пароль; попытки поменять другие пароли RLS просто
 игнорирует
postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

Все политики, создаваемые до этого, были разрешительными, что значит, что при применении нескольких политик они объединялись логическим оператором «ИЛИ». Хотя можно создать такие разрешительные политики, которые будут только разрешать доступ к строкам при определённых условиях, может быть проще скомбинировать разрешительные политики с ограничительными (которым должны удовлетворять записи и которые объединяются логическим оператором «И»). В развитие предыдущего примера мы добавим ограничительную политику, разрешающую администратору, подключённому через локальный сокет Unix, обращаться к записям таблицы `passwd`:

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (pg_catalog.inet_client_addr() IS NULL);
```

Затем мы можем убедиться, что администратор, подключённый по сети, не увидит никаких записей, благодаря этой ограничительной политике:

```
=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> select inet_client_addr();
 inet_client_addr
------------------
 127.0.0.1
(1 row)

=> TABLE passwd;
 user_name | pwhash | uid | gid | real_name | home_phone | extra_info | home_dir |
 shell
-----------+--------+-----+-----+-----------+------------+------------+----------
+-------
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

На проверки ссылочной целостности, например, на ограничения уникальности и внешние ключи, защита строк никогда не распространяется, чтобы не нарушалась целостность данных. Поэтому организацию и политики защиты на уровне строк необходимо тщательно прорабатывать, чтобы не возникли «скрытые каналы» утечки информации через эти проверки.

В некоторых случаях важно, чтобы защита на уровне строк, наоборот, не действовала. Например, резервное копирование может оказаться провальным, если механизм защиты на уровне строк молча не даст скопировать какие-либо строки. В таком случае вы можете установить для параметра конфигурации row_security значение `off`. Это само по себе не отключит защиту строк; при этом просто будет выдана ошибка, если результаты запроса отфильтруются политикой, с тем чтобы можно было изучить причину ошибки и устранить её.

В приведённых выше примерах выражения политики учитывали только текущие значения в запрашиваемой или изменяемой строке. Это самый простой и наиболее эффективный по скорости вариант; по возможности реализацию защиты строк следует проектировать именно так. Если же для принятия решения о доступе необходимо обращаться к другим строкам или другим таблицам, это можно осуществить, применяя в выражениях политик вложенные `SELECT` или функции, содержащие `SELECT`. Однако учтите, что при такой реализации возможны условия гонки, что чревато утечкой информации, если не принять меры предосторожности. Например, рассмотрим следующую конструкцию таблиц:

```
-- определение групп привилегий
CREATE TABLE groups (group_id int PRIMARY KEY,
                     group_name text NOT NULL);

INSERT INTO groups VALUES
  (1, 'low'),
  (2, 'medium'),
  (5, 'high');

GRANT ALL ON groups TO alice;  -- alice является администратором
GRANT SELECT ON groups TO public;

-- определение уровней привилегий для пользователей
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;

-- таблица, содержащая защищаемую информацию
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
  ('barely secret', 1),
  ('slightly secret', 2),
  ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- строка должна быть доступна для чтения/изменения пользователям с group_id,
-- большим или равным group_id данной строки
CREATE POLICY fp_s ON information FOR SELECT
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));

-- мы защищаем таблицу с информацией, полагаясь только на RLS
GRANT ALL ON information TO public;
```

Теперь предположим, что Алиса (роль `alice`) желает записать «слегка секретную» информацию, но при этом не хочет давать `mallory` доступ к ней. Она делает следующее:

```
BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
```

```
UPDATE information SET info = 'secret from mallory' WHERE group_id = 2;
COMMIT;
```

На первый взгляд всё нормально; `mallory` ни при каких условиях не должна увидеть строку «secret from mallory». Однако здесь возможно условие гонки. Если Мэллори (роль `mallory`) параллельно выполняет, скажем:

```
SELECT * FROM information WHERE group_id = 2 FOR UPDATE;
```

и её транзакция в режиме `READ COMMITTED`, она сможет увидеть «secret from mallory». Это произойдёт, если её транзакция дойдёт до строки `information` сразу после того, как эту строку изменит Алиса (роль `alice`). Она заблокируется, ожидая фиксирования транзакции Алисы, а затем прочитает изменённое содержимое строки благодаря предложению `FOR UPDATE`. Однако при этом изменённое содержимое `users` *не* будет прочитано неявным запросом `SELECT`, так как этот вложенный `SELECT` выполняется без указания `FOR UPDATE`; вместо этого строка `users` читается из снимка, полученного в начале запроса. Таким образом, выражение политики проверяет старое значение уровня привилегий пользователя `mallory` и позволяет ей видеть изменённую строку.

Обойти эту проблему можно несколькими способами. Первое простое решение заключается в использовании `SELECT ... FOR SHARE` во вложенных запросах `SELECT` в политиках защиты строк. Однако для этого потребуется давать затронутым пользователям право `UPDATE` в целевой таблице (здесь `users`), что может быть нежелательно. (Хотя можно применить ещё одну политику защиты строк, чтобы они не могли практически воспользоваться этим правилом; либо поместить вложенный `SELECT` в функцию, определяющую контекст безопасности.) Кроме этого, активное использование блокировок строк в целевой таблице может повлечь проблемы с производительностью, особенно при частых изменениях. Другое решение, практичное, если целевая таблица изменяется нечасто, заключается в блокировке целевой таблицы в режиме `ACCESS EXCLUSIVE` при изменении, чтобы никакие параллельные транзакции не видели старые значения строк. Либо можно просто дождаться завершения всех параллельных транзакций после изменения в целевой таблице, прежде чем вносить изменения, рассчитанные на новые условия безопасности.

За дополнительными подробностями обратитесь к CREATE POLICY и ALTER TABLE.

5.10. Схемы

Кластер баз данных Postgres Pro содержит один или несколько именованных экземпляров баз. На уровне кластера создаются роли и некоторые другие объекты. При этом в рамках одного подключения к серверу можно обращаться к данным только одной базы — той, что была выбрана при установлении соединения.

Примечание

Пользователи кластера не обязательно будут иметь доступ ко всем базам данных этого кластера. Тот факт, что роли создаются на уровне кластера, означает только то, что в кластере не может быть двух ролей `joe` в разных базах данных, хотя система позволяет ограничить доступ `joe` только некоторыми базами данных.

База данных содержит одну или несколько именованных *схем*, которые в свою очередь содержат таблицы. Схемы также содержат именованные объекты других видов, включая типы данных, функции и операторы. В одной схеме два объекта одного типа не могут иметь одинаковые имена. Более того, таблицы, последовательности, индексы, представления, материализованные представления и внешние таблицы существуют в одном пространстве имён, так что, например, имена индекса и таблицы должны отличаться, если они находятся в одной схеме. Одно и то же имя объекта можно свободно использовать в разных схемах, например и `schema1`, и `myschema` могут содержать таблицы с именем `mytable`. В отличие от баз данных, схемы не ограничивают доступ к данным: пользователи могут обращаться к объектам в любой схеме текущей базы данных, если им назначены соответствующие права.

Есть несколько возможных объяснений, для чего стоит применять схемы:

- Чтобы одну базу данных могли использовать несколько пользователей, независимо друг от друга.
- Чтобы объединить объекты базы данных в логические группы для облегчения управления ими.
- Чтобы в одной базе сосуществовали разные приложения, и при этом не возникало конфликтов имён.

Схемы в некотором смысле подобны каталогам в операционной системе, но они не могут быть вложенными.

5.10.1. Создание схемы

Для создания схемы используется команда CREATE SCHEMA. При этом вы определяете имя схемы по своему выбору, например так:

```
CREATE SCHEMA myschema;
```

Чтобы создать объекты в схеме или обратиться к ним, указывайте *полное имя*, состоящее из имён схемы и объекта, разделённых точкой:

```
схема.таблица
```

Этот синтаксис работает везде, где ожидается имя таблицы, включая команды модификации таблицы и команды обработки данных, обсуждаемые в следующих главах. (Для краткости мы будем говорить только о таблицах, но всё это распространяется и на другие типы именованных объектов, например, типы и функции.)

Есть ещё более общий синтаксис

```
база_данных.схема.таблица
```

но в настоящее время он поддерживается только для формального соответствия стандарту SQL. Если вы указываете базу данных, это может быть только база данных, к которой вы подключены.

Таким образом, создать таблицу в новой схеме можно так:

```
CREATE TABLE myschema.mytable (
 ...
);
```

Чтобы удалить пустую схему (не содержащую объектов), выполните:

```
DROP SCHEMA myschema;
```

Удалить схему со всеми содержащимися в ней объектами можно так:

```
DROP SCHEMA myschema CASCADE;
```

Стоящий за этим общий механизм описан в Разделе 5.15.

Часто бывает нужно создать схему, владельцем которой будет другой пользователь (это один из способов ограничения пользователей пространствами имён). Сделать это можно так:

```
CREATE SCHEMA имя_схемы AUTHORIZATION имя_пользователя;
```

Вы даже можете опустить имя схемы, в этом случае именем схемы станет имя пользователя. Как это можно применять, описано в Подразделе 5.10.6.

Схемы с именами, начинающимися с `pg_`, являются системными; пользователям не разрешено использовать такие имена.

5.10.2. Схема public

До этого мы создавали таблицы, не указывая никакие имена схем. По умолчанию такие таблицы (и другие объекты) автоматически помещаются в схему «public». Она содержится во всех создаваемых базах данных. Таким образом, команда:

```
CREATE TABLE products ( ... );
```

эквивалентна:

```
CREATE TABLE public.products ( ... );
```

5.10.3. Путь поиска схемы

Везде писать полные имена утомительно, и часто всё равно лучше не привязывать приложения к конкретной схеме. Поэтому к таблицам обычно обращаются по *неполному имени*, состоящему просто из имени таблицы. Система определяет, какая именно таблица подразумевается, используя *путь поиска*, который представляет собой список просматриваемых схем. Подразумеваемой таблицей считается первая подходящая таблица, найденная в схемах пути. Если подходящая таблица не найдена, возникает ошибка, даже если таблица с таким именем есть в других схемах базы данных.

Возможность создавать одноимённые объекты в разных схемах усложняет написание запросов, которые должны всегда обращаться к конкретным объектам. Это также потенциально позволяет пользователям влиять на поведение запросов других пользователей, злонамеренно или случайно. Ввиду преобладания неполных имён в запросах и их использования внутри Postgres Pro, добавить схему в `search_path` — по сути значит доверять всем пользователям, имеющим право `CREATE` в этой схеме. Когда вы выполняете обычный запрос, злонамеренный пользователь может создать объекты в схеме, включённой в ваш путь поиска, и таким образом перехватывать управление и выполнять произвольные функции SQL как если бы их выполняли вы.

Первая схема в пути поиска называется текущей. Эта схема будет использоваться не только при поиске, но и при создании объектов — она будет включать таблицы, созданные командой `CREATE TABLE` без указания схемы.

Чтобы узнать текущий тип поиска, выполните следующую команду:

```
SHOW search_path;
```

В конфигурации по умолчанию она возвращает:

```
 search_path
--------------
 "$user", public
```

Первый элемент ссылается на схему с именем текущего пользователя. Если такой схемы не существует, ссылка на неё игнорируется. Второй элемент ссылается на схему public, которую мы уже видели.

Первая существующая схема в пути поиска также считается схемой по умолчанию для новых объектов. Именно поэтому по умолчанию объекты создаются в схеме public. При указании неполной ссылки на объект в любом контексте (при модификации таблиц, изменении данных или в запросах) система просматривает путь поиска, пока не найдёт соответствующий объект. Таким образом, в конфигурации по умолчанию неполные имена могут относиться только к объектам в схеме public.

Чтобы добавить в путь нашу новую схему, мы выполняем:

```
SET search_path TO myschema,public;
```

(Мы опускаем компонент `$user`, так как здесь в нём нет необходимости.) Теперь мы можем обращаться к таблице без указания схемы:

```
DROP TABLE mytable;
```

И так как `myschema` — первый элемент в пути, новые объекты будут по умолчанию создаваться в этой схеме.

Мы можем также написать:

```
SET search_path TO myschema;
```

Тогда мы больше не сможем обращаться к схеме public, не написав полное имя объекта. Единственное, что отличает схему public от других, это то, что она существует по умолчанию, хотя её так же можно удалить.

В Разделе 9.27 вы узнаете, как ещё можно манипулировать путём поиска схем.

Как и для имён таблиц, путь поиска аналогично работает для имён типов данных, имён функций и имён операторов. Имена типов данных и функций можно записать в полном виде так же, как и имена таблиц. Если же вам нужно использовать в выражении полное имя оператора, для этого есть специальный способ — вы должны написать:

```
OPERATOR(схема.оператор)
```

Такая запись необходима для избежания синтаксической неоднозначности. Пример такого выражения:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

На практике пользователи часто полагаются на путь поиска, чтобы не приходилось писать такие замысловатые конструкции.

5.10.4. Схемы и права

По умолчанию пользователь не может обращаться к объектам в чужих схемах. Чтобы изменить это, владелец схемы должен дать пользователю право `USAGE` для данной схемы. По умолчанию все пользователи имеют это право для схемы `public`. Чтобы пользователи могли использовать объекты схемы, может понадобиться назначить дополнительные права на уровне объектов.

Пользователю также можно разрешить создавать объекты в схеме, не принадлежащей ему. Для этого ему нужно дать право `CREATE` в требуемой схеме. В базах данных, обновлённых с PostgreSQL 14 или более ранней версии, все имеют это право в схеме `public`. Некоторые шаблоны использования требуют запретить это:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(Первое слово «public» обозначает схему, а второе означает «каждый пользователь». В первом случае это идентификатор, а во втором — ключевое слово, поэтому они написаны в разном регистре; вспомните указания из Подраздела 4.1.1.)

5.10.5. Схема системного каталога

В дополнение к схеме `public` и схемам, создаваемым пользователями, любая база данных содержит схему `pg_catalog`, в которой находятся системные таблицы и все встроенные типы данных, функции и операторы. `pg_catalog` фактически всегда является частью пути поиска. Если даже эта схема не добавлена в путь явно, она неявно просматривается *до* всех схем, указанных в пути. Так обеспечивается доступность встроенных имён при любых условиях. Однако вы можете явным образом поместить `pg_catalog` в конец пути поиска, если вам нужно, чтобы пользовательские имена переопределяли встроенные.

Так как имена системных таблиц начинаются с `pg_`, такие имена лучше не использовать во избежание конфликта имён, возможного при появлении в будущем системной таблицы с тем же именем, что и ваша. (С путём поиска по умолчанию неполная ссылка будет воспринята как обращение к системной таблице.) Системные таблицы будут и дальше содержать в имени приставку `pg_`, так что они не будут конфликтовать с неполными именами пользовательских таблиц, если пользователи со своей стороны не будут использовать приставку `pg_`.

5.10.6. Шаблоны использования

Схемам можно найти множество применений. Для защиты от влияния недоверенных пользователей на поведение запросов других пользователей предлагается *шаблон безопасного использования схем*, но если этот шаблон не применяется в базе данных, пользователи, желающие безопасно выполнять в ней запросы, должны будут принимать защитные меры в начале каждого сеанса. В частности, они должны начинать каждый сеанс с присваивания пустого значения переменной `search_path` или каким-либо другим образом удалять из `search_path` схемы, доступные для записи обычным пользователям. С конфигурацией по умолчанию легко реализуются следующие шаблоны использования:

- Ограничить обычных пользователей личными схемами. Для реализации этого подхода сначала убедитесь, что ни у одной схемы нет права `CREATE`. Затем для каждого пользователя, который будет создавать не временные объекты, создайте схему с его именем, например `CREATE SCHEMA alice AUTHORIZATION alice`. (Как вы знаете, путь поиска по умолчанию начинается с имени `$user`, вместо которого подставляется имя пользователя. Таким образом, если у всех пользователей будет отдельная схема, они по умолчанию будут обращаться к собственным схемам.) Этот шаблон позволяет безопасно использовать схемы, только если никакой недоверенный пользователь не является владельцем базы данных и не получал право `ADMIN OPTION` для соответствующей роли. В противном случае безопасное использование схем невозможно.

  В PostgreSQL 15 и выше этот подход использования поддерживается конфигурацией по умолчанию. В предыдущих версиях или при использовании базы данных, обновлённой с предыдущей версии, необходимо удалить право `CREATE` из схемы `public` (выполнить `REVOKE CREATE ON SCHEMA public FROM PUBLIC`). Затем проверьте, нет ли в схеме `public` объектов с такими же именами, как у объектов в схеме `pg_catalog`.
- Удалить схему public из пути поиска по умолчанию, изменив `postgresql.conf` или выполнив команду `ALTER ROLE ALL SET search_path = "$user"`. Затем следует предоставить права на создание объектов в схеме public. Выбираться объекты в этой схеме будут только по полному имени. Тогда как обращаться к таблицам по полному имени вполне допустимо, обращения к функциям в общей схеме всё же будут небезопасными или ненадёжными. Поэтому если вы создаёте функции или расширения в схеме public, применяйте первый шаблон. Если же нет, этот шаблон, как и первый, безопасен при условии, что никакой недоверенный пользователь не является владельцем базы данных и не получал право `ADMIN OPTION` для соответствующей роли.
- Сохранить путь поиска по умолчанию и предоставить права создания объектов в схеме public. Все пользователи неявно обращаются к схеме public. Тем самым имитируется ситуация с полным отсутствием схем, что позволяет осуществить плавный переход из среды без схем. Однако данный шаблон ни в коем случае нельзя считать безопасным. Он подходит, только если в базе данных имеется всего один либо несколько доверяющих друг другу пользователей. В базах данных, обновлённых с версии PostgreSQL 14 или более ранней, этот шаблон применяется по умолчанию.

При любом подходе, устанавливая совместно используемые приложения (таблицы, которые нужны всем, дополнительные функции сторонних разработчиков и т. д.), помещайте их в отдельные схемы. Не забудьте дать другим пользователям права для доступа к этим схемам. Тогда пользователи смогут обращаться к этим дополнительным объектам по полному имени или при желании добавят эти схемы в свои пути поиска.

5.10.7. Переносимость

Стандарт SQL не поддерживает обращение в одной схеме к разным объектам, принадлежащим разным пользователям. Более того, в ряде реализаций СУБД нельзя создавать схемы с именем, отличным от имени владельца. На практике, в СУБД, реализующих только базовую поддержку схем согласно стандарту, концепции пользователя и схемы очень близки. Таким образом, многие пользователи полагают, что полное имя на самом деле образуется как *`имя_пользователя.имя_таблицы`*. И именно так будет вести себя Postgres Pro, если вы создадите схемы для каждого пользователя.

В стандарте SQL нет и понятия схемы `public`. Для максимального соответствия стандарту использовать схему `public` не следует.

Конечно, есть СУБД, в которых вообще не реализованы схемы или пространства имён поддерживают (возможно, с ограничениями) обращения к другим базам данных. Если вам потребуется работать с этими системами, максимальной переносимости вы достигнете, вообще не используя схемы.

5.11. Наследование

Postgres Pro реализует наследование таблиц, что может быть полезно для проектировщиков баз данных. (Стандарт SQL:1999 и более поздние версии определяют возможность наследования типов, но это во многом отличается от того, что описано здесь.)

Давайте начнём со следующего примера: предположим, что мы создаём модель данных для городов. В каждом штате есть множество городов, но лишь одна столица. Мы хотим иметь возможность быстро получать город-столицу для любого штата. Это можно сделать, создав две таблицы: одну для столиц штатов, а другую для городов, не являющихся столицами. Однако что делать, если нам нужно получить информацию о любом городе, будь то столица штата или нет? В решении этой проблемы может помочь наследование. Мы определим таблицу `capitals` как наследника `cities`:

```
CREATE TABLE cities (
    name            text,
    population      float,
    elevation       int     -- в футах
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

В этом случае таблица `capitals` *наследует* все столбцы своей родительской таблицы, `cities`. Столицы штатов также имеют дополнительный столбец `state`, в котором будет указан штат.

В Postgres Pro таблица может наследоваться от нуля или нескольких других таблиц, а запросы могут выбирать все строки родительской таблицы или все строки родительской и всех дочерних таблиц. По умолчанию принят последний вариант. Например, следующий запрос найдёт названия всех городов, включая столицы штатов, расположенных выше 500 футов:

```
SELECT name, elevation
    FROM cities
    WHERE elevation > 500;
```

Для данных из введения (см. Раздел 2.1) он выдаст:

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
```

А следующий запрос находит все города, которые не являются столицами штатов, но также находятся на высоте выше 500 футов:

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;

   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
```

Здесь ключевое слово `ONLY` указывает, что запрос должен применяться только к таблице `cities`, но не к таблицам, расположенным ниже `cities` в иерархии наследования. Многие операторы, которые мы уже обсудили, — `SELECT`, `UPDATE` и `DELETE` — поддерживают ключевое слово `ONLY`.

Вы также можете добавить после имени таблицы `*`, чтобы явно указать, что должны включаться и дочерние таблицы:

```
SELECT name, elevation
    FROM cities*
    WHERE elevation > 500;
```

Указывать `*` не обязательно, так как теперь это поведение всегда подразумевается по умолчанию. Однако такая запись всё ещё поддерживается для совместимости со старыми версиями, где поведение по умолчанию могло быть изменено.

В некоторых ситуациях бывает необходимо узнать, из какой таблицы выбрана конкретная строка. Для этого вы можете воспользоваться системным столбцом `tableoid`, присутствующим в каждой таблице:

```
SELECT c.tableoid, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

этот запрос выдаст:

```
 tableoid |   name    | elevation
----------+-----------+-----------
   139793 | Las Vegas |      2174
   139793 | Mariposa  |      1953
   139798 | Madison   |       845
```

(Если вы попытаетесь выполнить его у себя, скорее всего вы получите другие значения OID.) Собственно имена таблиц вы можете получить, обратившись к `pg_class`:

```
SELECT p.relname, c.name, c.elevation
FROM cities c, pg_class p
WHERE c.elevation > 500 AND c.tableoid = p.oid;
```

в результате вы получите:

```
 relname  |   name    | elevation
----------+-----------+-----------
 cities   | Las Vegas |      2174
 cities   | Mariposa  |      1953
 capitals | Madison   |       845
```

Тот же эффект можно получить другим способом, используя альтернативный тип `regclass`; при этом OID таблицы выводится в символьном виде:

```
SELECT c.tableoid::regclass, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

Механизм наследования не способен автоматически распределять данные команд `INSERT` или `COPY` по таблицам в иерархии наследования. Поэтому в нашем примере этот оператор `INSERT` не выполнится:

```
INSERT INTO cities (name, population, elevation, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

Мы могли надеяться на то, что данные каким-то образом попадут в таблицу `capitals`, но этого не происходит: `INSERT` всегда вставляет данные непосредственно в указанную таблицу. В некоторых случаях добавляемые данные можно перенаправлять, используя правила (см. Главу 39). Однако в нашем случае это не поможет, так как таблица `cities` не содержит столбца `state` и команда будет отвергнута до применения правила.

Дочерние таблицы автоматически наследуют от родительской таблицы ограничения-проверки и ограничения NOT NULL (если только для них не задано явно `NO INHERIT`). Все остальные ограничения (уникальности, первичный ключ и внешние ключи) не наследуются.

Таблица может наследоваться от нескольких родительских таблиц, в этом случае она будет объединять в себе все столбцы этих таблиц, а также столбцы, описанные непосредственно в её определении. Если в определениях родительских и дочерней таблиц встретятся столбцы с одним именем, эти столбцы будут «объединены», так что в дочерней таблице окажется только один столбец. Чтобы такое объединение было возможно, столбцы должны иметь одинаковый тип данных, в противном случае произойдёт ошибка. Наследуемые ограничения-проверки и ограничения NOT NULL объединяются подобным образом. Так, например, объединяемый столбец получит свойство NOT NULL, если какое-либо из порождающих его определений имеет свойство NOT NULL. Ограничения-проверки объединяются, если они имеют одинаковые имена; но если их условия различаются, происходит ошибка.

Отношение наследования между таблицами обычно устанавливается при создании дочерней таблицы с использованием предложения `INHERITS` оператора `CREATE TABLE`. Другой способ добавить такое отношение для таблицы, определённой подходящим образом, — использовать `INHERIT` с оператором `ALTER TABLE`. Для этого будущая дочерняя таблица должна уже включать те же столбцы (с совпадающими именами и типами), что и родительская таблица. Также она должна включать аналогичные ограничения-проверки (с теми же именами и выражениями). Удалить отношение наследования можно с помощью указания `NO INHERIT` оператора `ALTER TABLE`. Динамическое добавление и удаление отношений наследования может быть полезно при реализации секционирования таблиц (см. Раздел 5.12).

Для создания таблицы, которая затем может стать наследником другой, удобно воспользоваться предложением `LIKE` оператора `CREATE TABLE`. Такая команда создаст новую таблицу с теми же столбцами, что имеются в исходной. Если в исходной таблицы определены ограничения `CHECK`, для создания полностью совместимой таблицы их тоже нужно скопировать, и это можно сделать, добавив к предложению `LIKE` параметр `INCLUDING CONSTRAINTS`.

Родительскую таблицу нельзя удалить, пока существуют унаследованные от неё. При этом в дочерних таблицах нельзя удалять или модифицировать столбцы или ограничения-проверки, унаследованные от родительских таблиц. Если вы хотите удалить таблицу вместе со всеми её потомками, это легко сделать, добавив в команду удаления родительской таблицы параметр `CASCADE` (см. Раздел 5.15).

При изменениях определений и ограничений столбцов команда `ALTER TABLE` распространяет эти изменения вниз в иерархии наследования. Однако удалить столбцы, унаследованные дочерними таблицами, можно только с помощью параметра `CASCADE`. При создании отношений наследования команда `ALTER TABLE` следует тем же правилам объединения дублирующихся столбцов, что и `CREATE TABLE`.

В запросах с наследуемыми таблицами проверка прав доступа выполняется только в родительской таблице. Так, например, наличие разрешения `UPDATE` для таблицы `cities` подразумевает право на изменение строк также в таблице `capitals`, когда к ним происходит обращение через таблицу `cities`. Это сохраняет видимость того, что эти данные (также) находятся в родительской таблице. Но изменить таблицу `capitals` напрямую без дополнительного разрешения нельзя. Подобным образом, политики защиты на уровне строк (см. Раздел 5.9) для родительской таблицы применяются к строкам, получаемым из дочерних таблиц при выполнении запроса с наследованием. Политики же дочерних таблиц, если они определены, действуют только когда такие таблицы явно задействуются в запросе; в этом случае все политики, связанные с родительскими таблицами, игнорируются.

Сторонние таблицы (см. Раздел 5.13) могут также входить в иерархию наследования как родительские или дочерние таблицы, так же, как и обычные. Если в иерархию наследования входит сторонняя таблица, все операции, не поддерживаемые ей, не будут поддерживаться иерархией в целом.

5.11.1. Ограничения

Заметьте, что не все SQL-команды могут работать с иерархиями наследования. Команды, выполняющие выборку данных, изменение данных или модификацию схемы (например `SELECT`, `UPDATE`, `DELETE`, большинство вариантов `ALTER TABLE`, но не `INSERT` и `ALTER TABLE ... RENAME`), обычно по умолчанию обрабатывают данные дочерних таблиц и могут исключать их, если поддерживают указание `ONLY`. Команды для обслуживания и настройки базы данных (например `REINDEX` и `VACUUM`) обычно работают только с отдельными физическими таблицами и не поддерживают рекурсивную обработку отношений наследования. Соответствующее поведение каждой команды описано в её справке (Команды SQL).

Возможности наследования серьёзно ограничены тем, что индексы (включая ограничения уникальности) и ограничения внешних ключей относятся только к отдельным таблицам, но не к их потомкам. Это касается обеих сторон ограничений внешних ключей. Таким образом, применительно к нашему примеру:

- Если мы объявим `cities.name` с ограничением `UNIQUE` или `PRIMARY KEY`, это не помешает добавить в таблицу `capitals` строки с названиями городов, уже существующими в таблице `cities`. И эти дублирующиеся строки по умолчанию будут выводиться в результате запросов к `cities`. На деле таблица `capitals` по умолчанию вообще не будет содержать ограничение уникальности, так что в ней могут оказаться несколько строк с одним названием. Хотя вы можете добавить в `capitals` соответствующее ограничение, но это не предотвратит дублирование при объединении с `cities`.
- Подобным образом, если мы укажем, что `cities.name` ссылается (`REFERENCES`) на какую-то другую таблицу, это ограничение не будет автоматически распространено на `capitals`. В этом случае решением может стать явное добавление такого же ограничения `REFERENCES` в таблицу `capitals`.
- Если вы сделаете, чтобы столбец другой таблицы ссылался на `cities(name)`, в этом столбце можно будет указывать только названия городов, но не столиц. В этом случае хорошего решения нет.

Некоторая функциональность, не реализованная для иерархий наследования, реализована для декларативного секционирования. Поэтому обязательно взвесьте все за и против, прежде чем применять в своих приложениях секционирование с использованием устаревшего наследования.

5.12. Секционирование таблиц

Postgres Pro поддерживает простое секционирование таблиц. В этом разделе описывается, как и почему бывает полезно применять секционирование при проектировании баз данных.

5.12.1. Обзор

Секционированием называется разбиение данных, логически являющихся одной большой таблицей, на более мелкие физические части (секции). Секционирование может принести следующую пользу:

- В определённых ситуациях оно кардинально увеличивает быстродействие, особенно когда большой процент часто запрашиваемых строк таблицы относится к одной или лишь нескольким секциям. Секционирование по сути заменяет верхние уровни деревьев индексов, что увеличивает вероятность нахождения наиболее востребованных частей индексов в памяти.
- Когда в выборке или изменении данных задействована большая часть одной секции, производительность может возрасти, если будет выполняться последовательное сканирование этой секции, а не поиск по индексу, сопровождаемый произвольным чтением данных, разбросанных по всей таблице.
- Массовую загрузку и удаление данных можно осуществлять, добавляя и удаляя секции, если такой вариант использования был предусмотрен при проектировании секций. Удаление отдельной секции командой `DROP TABLE` и действие `ALTER TABLE DETACH PARTITION` выполняются гораздо быстрее, чем аналогичная массовая операция. Эти команды полностью исключают накладные расходы, связанные с выполнением `VACUUM` после массовой операции `DELETE`.
- Редко используемые данные можно перенести на более дешёвые и медленные носители.

Всё это обычно полезно только для очень больших таблиц. Какие именно таблицы выиграют от секционирования, зависит от конкретного приложения, хотя, как правило, это следует применять для таблиц, размер которых превышает объём ОЗУ сервера.

Postgres Pro предлагает поддержку следующих видов секционирования:

Секционирование по диапазонам

Таблица секционируется по «диапазонам», определённым по ключевому столбцу или набору столбцов, и не пересекающимся друг с другом. Например, можно секционировать данные по диапазонам дат или по диапазонам идентификаторов определённых бизнес-объектов. Границы каждого диапазона считаются включающими нижнее значение и исключающими верхнее. Например, если для первой секции задан диапазон значений от `1` до `10`, а для второй — от `10` до `20`, значение `10` относится ко второй секции, а не к первой.

Секционирование по списку

Таблица секционируется с помощью списка, явно указывающего, какие значения ключа должны относиться к каждой секции.

Секционирование по хешу

Таблица секционируется по определённым модулям и остаткам, которые указываются для каждой секции. Каждая секция содержит строки, для которых хеш-значение ключа секционирования, делённое на модуль, равняется заданному остатку.

Если вашему приложению требуются другие формы секционирования, можно также прибегнуть к альтернативным реализациям, с использованием наследования и представлений с `UNION ALL`. Такие подходы дают гибкость, но не дают такого выигрыша в производительности, как встроенное декларативное секционирование.

5.12.2. Декларативное секционирование

Postgres Pro позволяет декларировать, что некоторая таблица разделяется на секции. Разделённая на секции таблица называется *секционированной таблицей*. Декларация секционирования состоит из описанного выше определения *метода секционирования* и списка столбцов или выражений, образующих *ключ секционирования*.

Сама секционированная таблица является «виртуальной» и как таковая не хранится. Хранилище используется её *секциями*, которые являются обычными таблицами, связанными с секционированной. В каждой секции хранится подмножество данных таблицы, определяемое её *границами секции*. Все строки, вставляемые в секционированную таблицу, перенаправляются в соответствующие секции в зависимости от значений столбцов ключа секционирования. Если при изменении значений ключа секционирования в строке она перестаёт удовлетворять ограничениям исходной секции, эта строка перемещается в другую секцию.

Сами секции могут представлять собой секционированные таблицы, таким образом реализуется *вложенное секционирование*. Хотя все секции должны иметь те же столбцы, что и секционированная родительская таблица, в каждой секции независимо от других могут быть определены свои индексы, ограничения и значения по умолчанию. Подробнее о создании секционированных таблиц и секций рассказывается в описании CREATE TABLE.

Преобразовать обычную таблицу в секционированную и наоборот нельзя. Однако в секционированную таблицу можно добавить в качестве секции существующую обычную или секционированную таблицу, а также можно удалить секцию из секционированной таблицы и превратить её в отдельную таблицу; это может ускорить многие процессы обслуживания. Обратитесь к описанию ALTER TABLE, чтобы узнать больше о подкомандах `ATTACH PARTITION` и `DETACH PARTITION`.

Секции также могут быть сторонними таблицами, но учтите, что именно пользователь ответственен за соответствие содержимого сторонней таблицы правилу секционирования, это не контролируется автоматически. Также существуют и некоторые другие ограничения. За подробностями обратитесь к описанию CREATE FOREIGN TABLE.

5.12.2.1. Пример

Предположим, что мы создаём базу данных для большой компании, торгующей мороженым. Компания учитывает максимальную температуру и продажи мороженого каждый день в разрезе регионов. По сути нам нужна следующая таблица:

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
```

```
);
```

Мы знаем, что большинство запросов будут работать только с данными за последнюю неделю, месяц или квартал, так как в основном эта таблица нужна для формирования текущих отчётов для руководства. Чтобы сократить объём хранящихся старых данных, мы решили оставлять данные только за 3 последних года. Ненужные данные мы будем удалять в начале каждого месяца. В этой ситуации мы можем использовать секционирование для удовлетворения всех наших требований к таблице показателей.

Чтобы использовать декларативное секционирование в этом случае, выполните следующее:

1. Создайте таблицу `measurement` как секционированную таблицу с предложением `PARTITION BY`, указав метод секционирования (в нашем случае `RANGE`) и список столбцов, которые будут образовывать ключ секционирования.

   ```
   CREATE TABLE measurement (
       city_id         int not null,
       logdate         date not null,
       peaktemp        int,
       unitsales       int
   ) PARTITION BY RANGE (logdate);
   ```
2. Создайте секции. В определении каждой секции должны задаваться границы, соответствующие методу и ключу секционирования родительской таблицы. Заметьте, что указание границ, при котором множество значений новой секции пересекается со множеством значений в одной или нескольких существующих секциях, будет ошибочным.

   Секции, создаваемые таким образом, во всех отношениях являются обычными таблицами Postgres Pro (или, возможно, сторонними таблицами). В частности, для каждой секции можно независимо задать табличное пространство и параметры хранения.

   В нашем примере каждая секция должна содержать данные за один месяц, чтобы данные можно было удалять по месяцам согласно требованиям. Таким образом, нужные команды будут выглядеть так:

   ```
   CREATE TABLE measurement_y2006m02 PARTITION OF measurement
       FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

   CREATE TABLE measurement_y2006m03 PARTITION OF measurement
       FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

   ...
   CREATE TABLE measurement_y2007m11 PARTITION OF measurement
       FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

   CREATE TABLE measurement_y2007m12 PARTITION OF measurement
       FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
       TABLESPACE fasttablespace;

   CREATE TABLE measurement_y2008m01 PARTITION OF measurement
       FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
       WITH (parallel_workers = 4)
       TABLESPACE fasttablespace;
   ```

   (Как говорилось ранее, границы соседних секций могут определяться одинаковыми значениями, так как верхние границы не включаются в диапазон.)

   Если вы хотите реализовать вложенное секционирование, дополнительно укажите предложение `PARTITION BY` в командах, создающих отдельные секции, например:

   ```
   CREATE TABLE measurement_y2006m02 PARTITION OF measurement
       FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
       PARTITION BY RANGE (peaktemp);
   ```

Когда будут созданы секции `measurement_y2006m02`, данные, добавляемые в `measurement` и попадающие в `measurement_y2006m02` (или данные, которые могут добавляться непосредственно в `measurement_y2006m02` при условии соблюдения ограничения данной секции) будут затем перенаправлены в одну из вложенных секций в зависимости от значения столбца `peaktemp`. Указанный ключ секционирования может пересекаться с ключом секционирования родителя, хотя определять границы вложенной секции нужно осмотрительно, чтобы множество данных, которое она принимает, входило во множество, допускаемое собственными границами секции; система не пытается контролировать это сама.

При добавлении в родительскую таблицу данных, которые не соответствуют ни одной из существующих секций, произойдёт ошибка; подходящую секцию нужно создавать вручную.

Создавать в таблицах ограничения с условиями, задающими границы секций, вручную не требуется. Такие ограничения будут созданы автоматически.

3. Создайте в секционируемой таблице индекс по ключевому столбцу (или столбцам), а также любые другие индексы, которые могут понадобиться. (Индекс по ключу, строго говоря, создавать не обязательно, но в большинстве случаев он будет полезен.) При этом автоматически будет создан соответствующий индекс в каждой секции и все секции, которые вы будете создавать или присоединять позднее, тоже будут содержать такой индекс. Индексы или ограничения уникальности, созданные в секционированной таблице, являются «виртуальными», как и сама секционированная таблица: фактически данные находятся в дочерних индексах отдельных таблиц-секций.

   ```
   CREATE INDEX ON measurement (logdate);
   ```
4. Убедитесь в том, что параметр конфигурации enable_partition_pruning не выключен в `postgresql.conf`. Иначе запросы не будут оптимизироваться должным образом.

В данном примере нам потребуется создавать секцию каждый месяц, так что было бы разумно написать скрипт, который бы формировал требуемый код DDL автоматически.

5.12.2.2. Обслуживание секций

Обычно набор секций, образованный изначально при создании таблиц, не предполагается сохранять неизменным. Чаще наоборот, планируется удалять секции со старыми данными и периодически добавлять секции с новыми. Одно из наиболее важных преимуществ секционирования состоит именно в том, что оно позволяет практически моментально выполнять трудоёмкие операции, изменяя структуру секций, а не физически перемещая большие объёмы данных.

Самый лёгкий способ удалить старые данные — просто удалить секцию, ставшую ненужной:

```
DROP TABLE measurement_y2006m02;
```

Так можно удалить миллионы записей гораздо быстрее, чем удалять их по одной. Заметьте, однако, что приведённая выше команда требует установления блокировки `ACCESS EXCLUSIVE`.

Ещё один часто более предпочтительный вариант — убрать секцию из главной таблицы, но сохранить возможность обращаться к ней как к самостоятельной таблице:

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02 CONCURRENTLY;
```

При этом можно будет продолжать работать с данными, пока таблица не будет удалена. Например, в этом состоянии очень кстати будет сделать резервную копию данных, используя `COPY`, pg_dump или подобные средства. Возможно, эти данные также можно будет агрегировать, перевести в компактный формат, выполнить другую обработку или построить отчёты. Первая форма команды требует блокировки `ACCESS EXCLUSIVE` родительской таблицы. Благодаря указанию `CONCURRENTLY`, как во второй форме, для операции отсоединения будет требоваться только блокировка `SHARE UPDATE EXCLUSIVE` родительской таблицы, но при этом действуют ограничения, описанные в `ALTER TABLE ... DETACH PARTITION`.

Аналогичным образом можно добавлять новую секцию с данными. Мы можем создать пустую секцию в главной таблице так же, как мы создавали секции в исходном состоянии до этого:

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
    TABLESPACE fasttablespace;
```

А иногда удобнее создать новую таблицу вне структуры секций и присоединить её в виде секции позже. При таком подходе новые данные можно будет загрузить, проверить и преобразовать до того, как они появятся в секционированной таблице. Кроме того, операция `ATTACH PARTITION` требует для секционированной таблицы только блокировки `SHARE UPDATE EXCLUSIVE`, в отличие от операции `CREATE TABLE ... PARTITION OF`, требующей блокировки `ACCESS EXCLUSIVE`, поэтому её удобнее использовать для выполнения параллельных операций с секционированной таблицей. За подробностями обратитесь к описанию `ALTER TABLE ... ATTACH PARTITION`. Обойтись без кропотливого воспроизведения определения родительской таблицы можно, воспользовавшись функциональностью `CREATE TABLE ... LIKE`:

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

Обратите внимание, что при запуске команды `ATTACH PARTITION` таблица будет сканироваться для проверки ограничения секции с блокировкой `ACCESS EXCLUSIVE` этой секции. Как показано выше, рекомендуется избегать такого сканирования, создавая в присоединяемой таблице ограничение `CHECK`, соответствующее предполагаемому ограничению секции, до присоединения. После выполнения команды `ATTACH PARTITION` рекомендуется удалить ограничение `CHECK`, поскольку оно больше не нужно. Если присоединяемая таблица также является секционированной таблицей, то каждая из её секций будет рекурсивно блокироваться и сканироваться до тех пор, пока не встретится подходящее ограничение `CHECK` или не будут достигнуты конечные разделы.

Точно так же, если в секционированной таблице есть секция `DEFAULT`, рекомендуется создать ограничение `CHECK`, которое исключает ограничение секции, подлежащей присоединению. Если этого не сделать, то секция `DEFAULT` будет просканирована, чтобы убедиться, что она не содержит записей, которые должны быть расположены в присоединяемой секции. Эта операция будет выполняться при удержании блокировки `ACCESS EXCLUSIVE` на секции `DEFAULT`. Если секция `DEFAULT` сама является секционированной таблицей, то каждая из её секций будет рекурсивно проверяться таким же образом, как и присоединяемая таблица, как упоминалось выше.

Как упоминалось выше, в секционированных таблицах можно создавать индексы так, чтобы они применялись автоматически ко всей иерархии. Это очень удобно, так как индексироваться будут не только все существующие секции, но и любые секции, создаваемые в будущем. Однако есть одно ограничение: такой секционированный индекс нельзя создать в неблокирующем режиме (с указанием `CONCURRENTLY`), что может приводить к длительным блокировкам. Чтобы их избежать, для создания индекса в самой секционированной таблице можно использовать команду `CREATE INDEX ON ONLY`, тогда созданный индекс сразу будет помечен как нерабочий и не будет автоматически применён к существующим секциям. Индексы собственно в секциях можно создать в индивидуальном порядке с указанием `CONCURRENTLY`, а затем *присоединить* их к индексу-родителю, используя команду `ALTER INDEX .. ATTACH PARTITION`. После того как индексы всех секций будут присоединены к родительскому, последний автоматически перейдёт в рабочее состояние. Например:

```
CREATE INDEX measurement_usls_idx ON ONLY measurement (unitsales);

CREATE INDEX CONCURRENTLY measurement_usls_200602_idx
    ON measurement_y2006m02 (unitsales);
```

```
ALTER INDEX measurement_usls_idx
    ATTACH PARTITION measurement_usls_200602_idx;
...
```

Этот приём можно применять и с ограничениями `UNIQUE` и `PRIMARY KEY`, для них индексы создаются неявно при создании ограничения. Например:

```
ALTER TABLE ONLY measurement ADD UNIQUE (city_id, logdate);

ALTER TABLE measurement_y2006m02 ADD UNIQUE (city_id, logdate);
ALTER INDEX measurement_city_id_logdate_key
    ATTACH PARTITION measurement_y2006m02_city_id_logdate_key;
...
```

Также есть возможность объединить несколько секций таблиц в одну с помощью команды `ALTER TABLE ... MERGE PARTITIONS`. Она упрощает управление секционированными таблицами, позволяя пользователям объединять секции, которые больше не нужны по отдельности. Важно отметить, что эта операция не поддерживается для таблиц, секционированных по хешу, и запрашивает блокировку `ACCESS EXCLUSIVE`, что может ухудшать производительность высоконагруженных систем. Например, можно объединить три ежемесячных секции в одну квартальную:

```
ALTER TABLE measurement
    MERGE PARTITIONS (measurement_y2006m01,
                      measurement_y2006m02,
                      measurement_y2006m03) INTO measurement_y2006q1;
```

Кроме того, есть возможность разбить одну секцию на несколько с помощью команды `ALTER TABLE ... SPLIT PARTITION`. Она пригодится в случаях, когда одна секция становится слишком большой и её необходимо разбить на несколько. Важно отметить, что эта операция не поддерживается для таблиц, секционированных по хешу, и запрашивает блокировку `ACCESS EXCLUSIVE`, что может ухудшать производительность высоконагруженных систем. Например, можно разбить квартальную секцию обратно на ежемесячные:

```
ALTER TABLE measurement SPLIT PARTITION measurement_y2006q1 INTO
   (PARTITION measurement_y2006m01 FOR VALUES FROM ('2006-01-01') TO ('2006-02-01'),
    PARTITION measurement_y2006m02 FOR VALUES FROM ('2006-02-01') TO ('2006-03-01'),
    PARTITION measurement_y2006m03 FOR VALUES FROM ('2006-03-01') TO ('2006-04-01'));
```

5.12.2.3. Ограничения

С секционированными таблицами связаны следующие ограничения:

- Создать ограничение уникальности или первичного ключа для секционированной таблицы можно, только если ключи секций не включают никакие выражения или вызовы функций и в ограничение входят все столбцы ключа секционирования. Это условие объясняется тем, что отдельные индексы, образующие ограничение, могут непосредственно обеспечивать уникальность только в своих секциях. Поэтому сама структура секционирования должна гарантировать отсутствие дубликатов в разных секциях.
- Подобным образом ограничение-исключение должно включать все столбцы ключа секционирования. Кроме того, ограничение должно сравнивать эти столбцы на предмет равенства (не `&&`, например). Это обязательное условие возникает из-за того, что нельзя принудительно применять ограничения между секциями. Ограничение может включать дополнительные столбцы, которые не являются частью ключа секционирования, и сравнивать их с любыми операторами.
- Триггеры `BEFORE ROW` для `INSERT` не могут менять секцию, в которую в итоге попадёт новая строка.
- Смешивание временных и постоянных отношений в одном дереве секционирования не допускается. Таким образом, если секционированная таблица постоянная, такими же должны быть её секции; с временными таблицами аналогично. В случае с временными отношениями все таблицы дерева секционирования должны быть из одного сеанса.

На уровне реализации отдельные секции связываются с секционированной таблицей средствами наследования. Однако с декларативно секционированными таблицами или их секциями нельзя использовать некоторую общую функциональность наследования, как описано ниже. А именно, секция не может иметь никаких других родителей, кроме секционированной таблицы, к которой она относится, равно как и любая таблица не может наследоваться и от секционированной, и от обычной таблицы. Это означает, что секционированные таблицы и их секции не совмещаются в иерархии наследования с обычными таблицами.

Так как иерархия секционирования, образованная секционированной таблицей и её секциями, является одновременно и иерархией наследования, она содержит `tableoid` и на неё распространяются все обычные правила наследования, описанные в Разделе 5.11, с некоторыми исключениями:

- В секциях не может быть столбцов, отсутствующих в родительской таблице. Такие столбцы невозможно определить ни при создании секций командой `CREATE TABLE`, ни путём последующего добавления в секции командой `ALTER TABLE`. Таблицы могут быть подключены в качестве секций командой `ALTER TABLE ... ATTACH PARTITION`, только если их столбцы в точности соответствуют родительской таблице.
- Ограничения `CHECK` вместе с `NOT NULL`, определённые в секционированной таблице, всегда наследуются всеми её секциями. Ограничения `CHECK` с характеристикой `NO INHERIT` в секционированных таблицах создавать нельзя. Также нельзя удалить ограничение `NOT NULL`, заданное для столбца секции, если такое же ограничение существует в родительской таблице.
- Использование указания `ONLY` при добавлении или удалении ограничения только в секционированной таблице поддерживается, лишь когда в ней нет секций. При наличии секций попытка использования `ONLY` вызовет ошибку для любых ограничений, кроме `UNIQUE` и `PRIMARY KEY`. С другой стороны, ограничения можно добавлять или удалять непосредственно в секциях (если они отсутствуют в родительской таблице).
- Так как секционированная таблица сама по себе не содержит данные, использование `TRUNCATE ONLY` для секционированной таблицы всегда будет считаться ошибкой.

5.12.3. Секционирование с использованием наследования

Хотя встроенное декларативное секционирование полезно во многих часто возникающих ситуациях, бывают обстоятельства, требующие более гибкого подхода. В этом случае секционирование можно реализовать, применив механизм наследования таблиц, что даст ряд возможностей, неподдерживаемых при декларативном секционировании, например:

- При декларативном секционировании все секции должны иметь в точности тот же набор столбцов, что и секционируемая таблица, тогда как обычное наследование таблиц допускает наличие в дочерних таблицах дополнительных столбцов, отсутствующих в родителе.
- Механизм наследования таблиц поддерживает множественное наследование.
- С декларативным секционированием поддерживается только секционирование по спискам, по диапазонам и по хешу, тогда как с наследованием таблиц данные можно разделять по любому критерию, выбранному пользователем. (Однако заметьте, что если исключение по ограничению не позволяет эффективно устранять дочерние таблицы из планов запросов, производительность запросов будет очень низкой.)

5.12.3.1. Пример

В этом примере будет создана структура секционирования, равнозначная структуре из примера с декларативным секционированием. Выполните следующие действия:

1. Создайте «главную» таблицу, от которой будут наследоваться все «дочерние» таблицы. Главная таблица не будет содержать данные. Не определяйте в ней никакие ограничения-проверки, если только вы не намерены применить их во всех дочерних таблицах. Также не имеет смысла определять в ней какие-либо индексы или ограничения уникальности. В нашем примере главной таблицей будет `measurement` со своим изначальным определением:

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
```

```
    peaktemp        int,
    unitsales       int
);
```

2. Создайте несколько «дочерних» таблиц, унаследовав их все от главной. Обычно в таких таблицах не будет никаких дополнительных столбцов, кроме унаследованных. Как и с декларативным секционированием, эти таблицы во всех отношениях будут обычными таблицами Postgres Pro (или сторонними таблицами).

   ```
   CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
   CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
   ...
   CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
   CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
   CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
   ```

3. Добавьте в дочерние таблицы неперекрывающиеся ограничения, определяющие допустимые значения ключей для каждой из них.

   Типичные примеры таких ограничений:

   ```
   CHECK ( x = 1 )
   CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
   CHECK ( outletID >= 100 AND outletID < 200 )
   ```

   Убедитесь в том, что ограничения не пересекаются, то есть никакие значения ключа не относятся сразу к нескольким дочерним таблицам. Например, часто допускают такую ошибку в определении диапазонов:

   ```
   CHECK ( outletID BETWEEN 100 AND 200 )
   CHECK ( outletID BETWEEN 200 AND 300 )
   ```

   Это не будет работать, так как неясно, к какой дочерней таблице должно относиться значение 200. Поэтому диапазоны должны определяться следующим образом:

   ```
   CREATE TABLE measurement_y2006m02 (
       CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
   ) INHERITS (measurement);

   CREATE TABLE measurement_y2006m03 (
       CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
   ) INHERITS (measurement);


   ...
   CREATE TABLE measurement_y2007m11 (
       CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
   ) INHERITS (measurement);

   CREATE TABLE measurement_y2007m12 (
       CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
   ) INHERITS (measurement);

   CREATE TABLE measurement_y2008m01 (
       CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
   ) INHERITS (measurement);
   ```

4. Для каждой дочерней таблицы создайте индекс по ключевому столбцу (или столбцам), а также любые другие индексы по своему усмотрению.

   ```
   CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
   CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
   CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
   CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
   CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
   ```

5. Мы хотим, чтобы наше приложение могло сказать `INSERT INTO measurement ...` и данные оказались в соответствующей дочерней таблице. Мы можем добиться этого, добавив подходящую триггерную функцию в главную таблицу. Если данные всегда будут добавляться только в последнюю дочернюю таблицу, нам будет достаточно очень простой функции:

   ```
   CREATE OR REPLACE FUNCTION measurement_insert_trigger()
   RETURNS TRIGGER AS $$
   BEGIN
       INSERT INTO measurement_y2008m01 VALUES (NEW.*);
       RETURN NULL;
   END;
   $$
   LANGUAGE plpgsql;
   ```

   После функции мы создадим вызывающий её триггер:

   ```
   CREATE TRIGGER insert_measurement_trigger
       BEFORE INSERT ON measurement
       FOR EACH ROW EXECUTE FUNCTION measurement_insert_trigger();
   ```

   Мы должны менять определение триггерной функции каждый месяц, чтобы она всегда вставляла данные в текущую дочернюю таблицу. Определение самого триггера, однако, менять не требуется.

   Но мы можем также сделать, чтобы сервер автоматически находил дочернюю таблицу, в которую нужно направить добавляемую строку. Для этого нам потребуется более сложная триггерная функция:

   ```
   CREATE OR REPLACE FUNCTION measurement_insert_trigger()
   RETURNS TRIGGER AS $$
   BEGIN
       IF ( NEW.logdate >= DATE '2006-02-01' AND
            NEW.logdate < DATE '2006-03-01' ) THEN
           INSERT INTO measurement_y2006m02 VALUES (NEW.*);
       ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
               NEW.logdate < DATE '2006-04-01' ) THEN
           INSERT INTO measurement_y2006m03 VALUES (NEW.*);
       ...
       ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
               NEW.logdate < DATE '2008-02-01' ) THEN
           INSERT INTO measurement_y2008m01 VALUES (NEW.*);
       ELSE
           RAISE EXCEPTION
     'Date out of range.  Fix the measurement_insert_trigger() function!';
       END IF;
       RETURN NULL;
   END;
   $$
   LANGUAGE plpgsql;
   ```

   Определение триггера остаётся прежним. Заметьте, что все условия `IF` должны в точности отражать ограничения `CHECK` соответствующих дочерних таблиц.

   Хотя эта функция сложнее, чем вариант с одним текущим месяцем, её не придётся так часто модифицировать, так как ветви условий можно добавить заранее.

Примечание

На практике будет лучше сначала проверять условие для последней дочерней таблицы, если строки добавляются в неё чаще всего, но для простоты мы расположили проверки триггера в том же порядке, что и в других фрагментах кода для этого примера.

Другой способ перенаправления добавляемых строк в соответствующую дочернюю таблицу можно реализовать, определив для главной таблицы не триггер, а правила. Например:

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

С правилами связано гораздо больше накладных расходов, чем с триггером, но они относятся к запросу в целом, а не к каждой строке. Поэтому этот способ может быть более выигрышным при массовом добавлении данных. Однако в большинстве случаев триггеры будут работать быстрее.

Учтите, что команда `COPY` игнорирует правила. Если вы хотите вставить данные с помощью `COPY`, вам придётся копировать их сразу в нужную дочернюю таблицу, а не в главную таблицу. С другой стороны, `COPY` не отменяет триггеры, так что с триггерами вы сможете использовать её обычным образом.

Ещё один недостаток подхода с правилами связан с невозможностью выдать ошибку, если добавляемая строка не подпадает ни под одно из правил; в этом случае данные просто попадут в главную таблицу.

6. Убедитесь в том, что параметр конфигурации constraint_exclusion не выключен в `postgresql.conf`. В противном случае дочерние таблицы могут сканироваться, когда это не требуется.

Как уже можно понять, для реализации сложной иерархии таблиц может потребоваться DDL-код значительного объёма. В данном примере нам потребуется создавать дочернюю таблицу каждый месяц, так что было бы разумно написать скрипт, формирующий требуемый код DDL автоматически.

5.12.3.2. Обслуживание таблиц, секционированных через наследование

Чтобы быстро удалить старые данные, просто удалите ставшую ненужной дочернюю таблицу:

```
DROP TABLE measurement_y2006m02;
```

Чтобы удалить дочернюю таблицу из иерархии наследования, но сохранить к ней доступ как к самостоятельной таблице:

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

Чтобы добавить новую дочернюю таблицу для новых данных, создайте пустую дочернюю таблицу так же, как до этого создавали начальные:

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

Можно также создать новую таблицу и наполнить её данными до добавления в иерархию таблиц. Это позволит загрузить, проверить и при необходимости преобразовать данные до того, как запросы к главной таблице смогут их увидеть.

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- возможна дополнительная подготовка данных
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.12.3.3. Ограничения

С реализацией секционирования через наследование связаны следующие ограничения:

- Система не может проверить автоматически, являются ли все ограничения `CHECK` взаимно исключающими. Поэтому безопаснее будет написать и отладить код для формирования дочерних таблиц и создания и/или изменения связанных объектов, чем делать это вручную.
- Индексы и внешние ключи относятся к определённой таблице, но не к её иерархии наследования, поэтому с ними связаны дополнительные ограничения.
- Показанные здесь схемы подразумевают, что ключевой столбец (или столбцы) в строке никогда не меняется, или меняется не настолько, чтобы строку потребовалось перенести в другую секцию. Если же попытаться выполнить такой оператор `UPDATE`, произойдёт ошибка из-за нарушения ограничения `CHECK`. Если вам нужно обработать и такие случаи, вы можете установить подходящие триггеры на обновление в дочерних таблицах, но это ещё больше усложнит управление всей конструкцией.
- Если вы выполняете команды `VACUUM` или `ANALYZE` вручную, не забывайте, что их нужно запускать для каждой дочерней таблицы в отдельности. Команда

  ```
  ANALYZE measurement;
  ```

  обработает только главную таблицу.
- Операторы `INSERT` с предложениями `ON CONFLICT` скорее всего не будут работать ожидаемым образом, так как действие `ON CONFLICT` предпринимается только в случае нарушений уникальности в указанном целевом отношении, а не его дочерних отношениях.
- Для направления строк в нужные дочерние таблицы потребуются триггеры или правила, если только приложение не знает непосредственно о схеме секционирования. Разработать триггеры может быть довольно сложно, и они будут работать гораздо медленнее, чем внутреннее распределение кортежей при декларативном секционировании.

5.12.4. Отсечение секций

Отсечение секций — это приём оптимизации запросов, который ускоряет работу с декларативно секционированными таблицами. Например:

```
SET enable_partition_pruning = on;                 -- по умолчанию
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

Без отсечения секций показанный запрос должен будет просканировать все секции таблицы `measurement`. Когда отсечение секций включено, планировщик рассматривает определение каждой секции и может заключить, что какую-либо секцию сканировать не нужно, так как в ней не может быть строк, удовлетворяющих предложению `WHERE` в запросе. Когда планировщик может сделать такой вывод, он исключает (*отсекает*) секцию из плана запроса.

Используя команду EXPLAIN и параметр конфигурации enable_partition_pruning, можно увидеть отличие плана, в котором были отсечены секции, от плана без отсечения. Типичный неоптимизированный план для такой конфигурации таблицы будет выглядеть так:

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

```
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=188.76..188.77 rows=1 width=8)
   ->  Append  (cost=0.00..181.05 rows=3085 width=0)
         ->  Seq Scan on measurement_y2006m02  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m11  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2007m12  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

В некоторых или всех секциях может применяться не полное последовательное сканирование, а сканирование по индексу, но основная идея примера в том, что для удовлетворения запроса не нужно сканировать старые секции. И когда мы включаем отсечение секций, мы получаем значительно более эффективный план, дающий тот же результат:

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Aggregate  (cost=37.75..37.76 rows=1 width=8)
   ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
         Filter: (logdate >= '2008-01-01'::date)
```

Заметьте, что механизм отсечения секций учитывает только ограничения, определённые неявно ключами секционирования, но не наличие индексов. Поэтому определять индексы для столбцов ключа не обязательно. Нужно ли создавать индекс для определённой секции, зависит от того, какую часть секции (меньшую или большую), по вашим представлениям, будут сканировать запросы, обращающиеся к этой секции. Индекс будет полезен в первом случае, но не во втором.

Отсечение секций может производиться не только при планировании конкретного запроса, но и в процессе его выполнения. Благодаря этому может быть отсечено больше секций, когда условные выражения содержат значения, неизвестные во время планирования, например параметры, определённые оператором `PREPARE`, значения, получаемые из подзапросов, или параметризованные значения во внутренней стороне соединения с вложенным циклом. Отсечение секций в процессе выполнения запроса возможно в следующие моменты времени:

- Во время подготовки плана запроса. В этот момент можно отсечь секции, учитывая значения параметров, известные при подготовке выполнения запроса. Секции, отсечённые на этом этапе, не будут видны в выводе `EXPLAIN` или `EXPLAIN ANALYZE`. Их общее количество можно определить по свойству «Subplans Removed» в выводе `EXPLAIN`. Важно отметить, что секции, отсечённые на этом этапе, остаются заблокированными на момент начала выполнения запроса.
- В процессе собственно выполнения плана запроса. Отсечение секций также может выполняться на этом этапе и позволяет отфильтровать секции, используя значения, которые становятся известны, когда запрос выполняется фактически. В частности это могут быть значения из подзапросов и значения параметров времени выполнения, например из параметризованных соединений с вложенными циклами. Так как значения параметров могут меняться многократно при выполнении запроса, отсечение секций выполняется при изменении любого из параметров, анализируемых механизмом отсечения. Чтобы определить, были ли секции отсечены на данном этапе, нужно внимательно изучить свойство `loops` в выводе `EXPLAIN ANALYZE`. Подпланы, соответствующие разным секциям, могут иметь разные значения, в зависимости от того, сколько раз они отсекались во время выполнения. Некоторые из них могут даже иметь значение `(never executed)` (никогда не выполнялись), если они отсекались всегда.

Отсечение секций можно отключить, воспользовавшись параметром enable_partition_pruning.

5.12.5. Секционирование и исключение по ограничению

Исключение по ограничению — приём оптимизации запросов, подобный отсечению секций. Прежде всего он применяется, когда секционирование осуществляется с использованием старого метода наследования, но он может быть полезен и для других целей, включая декларативное секционирование.

Исключение по ограничению работает во многом так же, как и отсечение секций; отличие состоит в том, что оно использует ограничения `CHECK` всех таблиц (поэтому оно так и называется), тогда как для отсечения секций используются границы секции, которые существуют только в случае декларативного секционирования. Ещё одно различие состоит в том, что исключение по ограничению применяется только во время планирования; во время выполнения секции из плана удаляться не будут.

То, что исключение по ограничению использует ограничения `CHECK` (вследствие чего оно работает медленнее отсечения секций), иногда может быть и преимуществом. Ограничения могут быть определены даже для декларативно секционированных таблиц, в дополнение к внутренним границам секций, и тогда исключение по ограничению сможет дополнительно убрать некоторые секции из плана запроса.

По умолчанию параметр constraint_exclusion имеет значение не `on` и не `off`, а промежуточное (и рекомендуемое) значение `partition`, при котором этот приём будет применяться только к запросам, где предположительно будут задействованы таблицы, секционированные с использованием наследования. Значение `on` обязывает планировщик просматривать ограничения `CHECK` во всех запросах, даже в самых простых, где выигрыш от исключения по ограничению маловероятен.

Применяя исключения по ограничению, необходимо учитывать следующее:

- Исключение по ограничению применяется только при планировании запроса, в отличие от отсечения секций, которое может осуществляться и при выполнении запроса.
- Исключение по ограничению работает только когда предложение `WHERE` в запросе содержит константы (или получаемые извне параметры). Например, сравнение с непостоянной функцией, такой как `CURRENT_TIMESTAMP`, нельзя оптимизировать, так как планировщик не может знать, в какую дочернюю таблицу попадёт значение функции во время выполнения.
- Ограничения секций должны быть простыми, иначе планировщик не сможет вычислить, какие дочерние таблицы не нужно обрабатывать. Для секционирования по спискам используйте простые условия на равенства, а для секционирования по диапазонам — простые проверки диапазонов, подобные показанным в примерах. Рекомендуется создавать ограничения секций, содержащие только такие сравнения секционирующих столбцов с константами, в которых используются операторы, поддерживающие B-деревья. Это объясняется тем, что в ключе секционирования допускаются только такие столбцы, которые могут быть проиндексированы в B-дереве.
- При анализе для исключения по ограничению исследуются все ограничения всех дочерних таблиц, относящихся к главной, так что при большом их количестве время планирования запросов может значительно увеличиться. Поэтому устаревший вариант секционирования, основанный на наследовании, будет работать хорошо, пока количество дочерних таблиц не превышает примерно ста; не пытайтесь применять его с тысячами дочерних таблиц.

5.12.6. Рекомендации по декларативному секционированию

К секционированию таблицы следует подходить продуманно, так как неудачное решение может отрицательно повлиять на скорость планирования и выполнения запросов.

Одним из самых важных факторов является выбор столбца или столбцов, по которым будут секционироваться ваши данные. Часто оптимальным будет секционирование по столбцу или набору столбцов, которые практически всегда присутствуют в предложении `WHERE` в запросах, обращающихся к секционируемой таблице. Предложения `WHERE`, совместимые с ограничениями границ секции, могут применяться для устранения ненужных для выполнения запроса секций. Однако наличие ограничений `PRIMARY KEY` или `UNIQUE` может подтолкнуть к выбору и других столбцов в качестве секционирующих. Также при планировании секционирования следует продумать, как будут удаляться данные. Секцию целиком можно отсоединить очень быстро, поэтому может иметь смысл разработать стратегию секционирования так, чтобы массово удаляемые данные оказывались в одной секции.

Также важно правильно выбрать число секций, на которые будет разбиваться таблица. Если их будет недостаточно много, индексы останутся большими, не улучшится и локальность данных, вследствие чего процент попаданий в кеш окажется низким. Однако и при слишком большом количестве секций возможны проблемы. С большим количеством секций увеличивается время планирования и потребление памяти как при планировании, так и при выполнении запросов, о чём рассказывается далее. Выбирая стратегию секционирования таблицы, также важно учитывать, какие изменения могут произойти в будущем. Например если вы решите создавать отдельные секции для каждого клиента, и в данный момент у вас всего несколько больших клиентов, подумайте, что будет, если через несколько лет у вас будет много мелких клиентов. В этом случае может быть лучше произвести секционирование по хешу (`HASH`) и выбрать разумное количество секций, но не создавать секции по списку (`LIST`) в надежде, что количество клиентов не увеличится до такой степени, что секционирование данных окажется непрактичным.

Вложенное секционирование позволяет дополнительно разделить те секции, которые предположительно окажутся больше других. Также можно использовать секционирование по диапазонам с ключом, состоящим из нескольких столбцов. Но при любом из подходов легко может получиться слишком много секций, так что рекомендуется использовать их осмотрительно.

Важно учитывать издержки секционирования, которые отражаются на планировании и выполнении запросов. Планировщик запросов обычно довольно неплохо справляется с иерархиями, включающими до нескольких тысяч секций, если при выполнении типичных запросов ему удаётся отсечь почти все секции. Однако когда после отсечения остаётся большое количество секций, возрастает и время планирования запросов, и объём потребляемой памяти. Наличие большого количества секций нежелательно ещё и потому, что потребление памяти сервером может значительно возрастать со временем, особенно когда множество сеансов обращаются ко множеству секций. Это объясняется тем, что в локальную память каждого сеанса, который обращается к секциям, должны быть загружены метаданные всех этих секций.

С нагрузкой, присущей информационным хранилищам, может иметь смысл создавать больше секций, чем с нагрузкой OLTP. Как правило, в информационных хранилищах время планирования запроса второстепенно, так как гораздо больше времени тратится на выполнение запроса. Однако при любом типе нагрузки важно принимать правильные решения на ранней стадии реализации, так как процесс пересекционирования таблиц большого объёма может оказаться чрезвычайно длительным. Для оптимизации стратегии секционирования часто бывает полезно предварительно эмулировать ожидаемую нагрузку. Но ни в коем случае нельзя полагать, что чем больше секций, тем лучше, равно как и наоборот.

5.13. Сторонние данные

Postgres Pro частично реализует спецификацию SQL/MED, позволяя вам обращаться к данным, находящимся снаружи, используя обычные SQL-запросы. Такие данные называются *сторонними*.

Сторонние данные доступны в PostgreSQL через *обёртку сторонних данных*. Обёртка сторонних данных — это библиотека, взаимодействующая с внешним источником данных и скрывающая в себе внутренние особенности подключения и получения данных. Несколько готовых обёрток предоставляются в виде модулей `contrib` ; см. Приложение F. Также вы можете найти другие обёртки, выпускаемые как дополнительные продукты. Если ни одна из существующих обёрток вас не устраивает, вы можете написать свою собственную (см. Главу 55).

Чтобы обратиться к сторонним данным, вы должны создать объект *сторонний сервер*, в котором настраивается подключение к внешнему источнику данных, определяются параметры соответствующей обёртки сторонних данных. Затем вы должны создать одну или несколько *сторонних* *таблиц*, определив тем самым структуру внешних данных. Сторонние таблицы можно использовать в запросах так же, как и обычные, но их данные не хранятся на сервере Postgres Pro. При каждом запросе Postgres Pro обращается к обёртке сторонних данных, которая, в свою очередь, получает данные из внешнего источника или передаёт их ему (в случае команд INSERT или UPDATE).

При обращении к внешним данным удалённый источник может потребовать аутентификации клиента. Соответствующие учётные данные можно предоставить с помощью *сопоставлений пользователей*, позволяющих определить в частности имена и пароли, в зависимости от текущей роли пользователя Postgres Pro.

Дополнительную информацию вы найдёте в CREATE FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE и IMPORT FOREIGN SCHEMA.

5.14. Другие объекты баз данных

Таблицы — центральные объекты в структуре реляционной базы данных, так как они содержат ваши данные. Но это не единственные объекты, которые могут в ней существовать. Помимо них вы можете создавать и использовать объекты и других типов, призванные сделать управление данными эффективнее и удобнее. Они не обсуждаются в этой главе, но мы просто перечислим некоторые из них, чтобы вы знали об их существовании:

- Представления
- Функции, процедуры и операторы
- Типы данных и домены
- Триггеры и правила перезаписи

Подробнее соответствующие темы освещаются в Части V.

5.15. Отслеживание зависимостей

Когда вы создаёте сложные структуры баз данных, включающие множество таблиц с внешними ключами, представлениями, триггерами, функциями и т. п., вы неявно создаёте сеть зависимостей между объектами. Например, таблица с ограничением внешнего ключа зависит от таблицы, на которую она ссылается.

Для сохранения целостности структуры всей базы данных, Postgres Pro не позволяет удалять объекты, от которых зависят другие. Например, попытка удалить таблицу products (мы рассматривали её в Подразделе 5.5.5), от которой зависит таблица orders, приведёт к ошибке примерно такого содержания:

```
DROP TABLE products;

ОШИБКА:  удалить объект "таблица products" нельзя, так как от него зависят другие
ПОДРОБНОСТИ:  ограничение orders_product_no_fkey в отношении "таблица orders" зависит
 от объекта "таблица products"
ПОДСКАЗКА:  Для удаления зависимых объектов используйте DROP ... CASCADE.
```

Сообщение об ошибке даёт полезную подсказку: если вы не хотите заниматься ликвидацией зависимостей по отдельности, можно выполнить:

```
DROP TABLE products CASCADE;
```

и все зависимые объекты, а также объекты, зависящие от них, будут удалены рекурсивно. В этом случае таблица orders останется, а удалено будет только её ограничение внешнего ключа. Удаление не распространится на другие объекты, так как ни один объект не зависит от этого ограничения. (Если вы хотите проверить, что произойдёт при выполнении `DROP ... CASCADE`, запустите `DROP` без `CASCADE` и прочитайте `ПОДРОБНОСТИ` (DETAIL).)

Почти все команды `DROP` в Postgres Pro поддерживают указание `CASCADE`. Конечно, вид возможных зависимостей зависит от типа объекта. Вы также можете написать `RESTRICT` вместо `CASCADE`, чтобы включить поведение по умолчанию, когда объект можно удалить, только если от него не зависят никакие другие.

Примечание

Стандарт SQL требует явного указания `RESTRICT` или `CASCADE` в команде `DROP`. Но это требование на самом деле не выполняется ни в одной СУБД, при этом одни системы по умолчанию подразумевают `RESTRICT`, а другие — `CASCADE`.

Если в команде `DROP` перечисляются несколько объектов, `CASCADE` требуется указывать, только когда есть зависимости вне заданной группы. Например, в команде `DROP TABLE tab1, tab2` при наличии внешнего ключа, ссылающегося на `tab1` из `tab2`, можно не указывать `CASCADE`, чтобы она выполнилась успешно.

Для пользовательских функций или процедур, тело которых задано строковой константой, Postgres Pro отслеживает зависимости, связанные с внешне видимыми свойствами функции, такими как типы аргументов и результата, но *не* зависимости, которые могут быть выявлены только при анализе тела функции. В качестве примера рассмотрите следующий сценарий:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                             'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

(Описание функций языка SQL можно найти в Разделе 36.5.) Postgres Pro будет понимать, что функция `get_color_note` зависит от типа `rainbow`: при удалении типа будет принудительно удалена функция, так как тип её аргумента оказывается неопределённым. Но Postgres Pro не будет учитывать зависимость `get_color_note` от таблицы `my_colors` и не удалит функцию при удалении таблицы. Но у этого подхода есть не только минус, но и плюс. В случае отсутствия таблицы эта функция останется рабочей в некотором смысле: хотя при попытке выполнить её возникнет ошибка, но при создании новой таблицы с тем же именем функция снова будет работать.

С другой стороны, при определении функции или процедуры в стиле стандарта SQL её тело разбирается, и все зависимости, распознаваемые синтаксическим анализатором, сохраняются. Таким образом, если записать приведённую выше функцию как

```
CREATE FUNCTION get_color_note (rainbow) RETURNS text
BEGIN ATOMIC
  SELECT note FROM my_colors WHERE color = $1;
END;
```

, то будет известна зависимость функции от таблицы `my_colors` и команда `DROP` её обработает.

Глава 6. Модификация данных

В предыдущей главе мы обсуждали, как создавать таблицы и другие структуры для хранения данных. Теперь пришло время заполнить таблицы данными. В этой главе мы расскажем, как добавлять, изменять и удалять данные из таблиц. А из следующей главы вы наконец узнаете, как извлекать нужные вам данные из базы данных.

6.1. Добавление данных

Сразу после создания таблицы она не содержит никаких данных. Поэтому, чтобы она была полезна, в неё прежде всего нужно добавить данные. По сути данные добавляются в таблицу по одной строке. И хотя вы конечно можете добавить в таблицу несколько строк, добавить в неё меньше, чем строку, невозможно. Даже если вы указываете значения только некоторых столбцов, создаётся полная строка.

Чтобы создать строку, вы будете использовать команду INSERT. В этой команде необходимо указать имя таблицы и значения столбцов. Например, рассмотрим таблицу товаров из Главы 5:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

Добавить в неё строку можно было бы так:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

Значения данных перечисляются в порядке столбцов в таблице и разделяются запятыми. Обычно в качестве значений указываются константы, но это могут быть и скалярные выражения.

Показанная выше запись имеет один недостаток — вам необходимо знать порядок столбцов в таблице. Чтобы избежать этого, можно перечислить столбцы явно. Например, следующие две команды дадут тот же результат, что и показанная выше:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

Многие считают, что лучше всегда явно указывать имена столбцов.

Если значения определяются не для всех столбцов, лишние столбцы можно опустить. В таком случае эти столбцы получат значения по умолчанию. Например:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

Вторая форма является расширением Postgres Pro. Она заполняет столбцы слева по числу переданных значений, а все остальные столбцы принимают значения по умолчанию.

Для ясности можно также явно указать значения по умолчанию для отдельных столбцов или всей строки:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

Одна команда может вставить сразу несколько строк:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

Также возможно вставить результат запроса (который может не содержать строк либо содержать одну или несколько):

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

Это позволяет использовать все возможности механизма запросов SQL (см. Главу 7) для вычисления вставляемых строк.

> **Подсказка**
>
> Когда нужно добавить сразу множество строк, возможно будет лучше использовать команду COPY. Она не такая гибкая, как INSERT, но гораздо эффективнее. Дополнительно об ускорении массовой загрузки данных можно узнать в Разделе 14.4.

6.2. Изменение данных

Модификация данных, уже сохранённых в БД, называется изменением. Изменить можно все строки таблицы, либо подмножество всех строк, либо только избранные строки. Каждый столбец при этом можно изменять независимо от других.

Для изменения данных в существующих строках используется команда UPDATE. Ей требуется следующая информация:

1. Имя таблицы и изменяемого столбца
2. Новое значение столбца
3. Критерий отбора изменяемых строк

Если вы помните, в Главе 5 говорилось, что в SQL в принципе нет уникального идентификатора строк. Таким образом, не всегда возможно явно указать на строку, которую требуется изменить. Поэтому необходимо указать условия, каким должны соответствовать требуемая строка. Только если в таблице есть первичный ключ (вне зависимости от того, объявляли вы его или нет), можно однозначно адресовать отдельные строки, определив условие по первичному ключу. Этим пользуются графические инструменты для работы с базой данных, дающие возможность редактировать данные по строкам.

Например, следующая команда увеличивает цену всех товаров, имевших до этого цену 5, до 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

В результате может измениться ноль, одна или множество строк. И если этому запросу не будет удовлетворять ни одна строка, это не будет ошибкой.

Давайте рассмотрим эту команду подробнее. Она начинается с ключевого слова `UPDATE`, за которым идёт имя таблицы. Как обычно, имя таблицы может быть записано в полной форме, в противном случае она будет найдена по пути. Затем идёт ключевое слово `SET`, за которым следует имя столбца, знак равенства и новое значение столбца. Этим значением может быть любое скалярное выражение, а не только константа. Например, если вы захотите поднять цену всех товаров на 10%, это можно сделать так:

```
UPDATE products SET price = price * 1.10;
```

Как видно из этого примера, выражение нового значения может ссылаться на существующие значения столбцов в строке. Мы также опустили в нём предложение `WHERE`. Это означает, что будут изменены все строки в таблице. Если же это предложение присутствует, изменяются только строки, которые соответствуют условию `WHERE`. Заметьте, что хотя знак равенства в предложении `SET` обозначает операцию присваивания, а такой же знак в предложении `WHERE` используется для сравнения, это не приводит к неоднозначности. И конечно, в условии `WHERE` не обязательно должна быть проверка равенства, а могут применяться и другие операторы (см. Главу 9). Необходимо только, чтобы это выражение возвращало логический результат.

В команде `UPDATE` можно изменить значения сразу нескольких столбцов, перечислив их в предложении `SET`. Например:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Удаление данных

Мы рассказали о том, как добавлять данные в таблицы и как изменять их. Теперь вам осталось узнать, как удалить данные, которые оказались не нужны. Так же, как добавлять данные можно только целыми строками, удалять их можно только по строкам. В предыдущем разделе мы отметили, что в SQL нет возможности напрямую адресовать отдельные строки, так что удалить избранные строки можно, только сформулировав для них подходящие условия. Но если в таблице есть первичный ключ, с его помощью можно однозначно выделить определённую строку. При этом можно так же удалить группы строк, соответствующие условию, либо сразу все строки таблицы.

Для удаления строк используется команда DELETE; её синтаксис очень похож на синтаксис команды UPDATE. Например, удалить все строки из таблицы с товарами, имеющими цену 10, можно так:

```
DELETE FROM products WHERE price = 10;
```

Если вы напишете просто:

```
DELETE FROM products;
```

будут удалены все строки таблицы! Будьте осторожны!

6.4. Возврат данных из изменённых строк

Иногда бывает полезно получать данные из модифицируемых строк в процессе их обработки. Это возможно с использованием предложения `RETURNING`, которое можно задать для команд `INSERT`, `UPDATE`, `DELETE` и `MERGE`. Применение `RETURNING` позволяет обойтись без дополнительного запроса к базе для сбора данных и это особенно ценно, когда как-то иначе трудно получить изменённые строки надёжным образом.

В предложении `RETURNING` допускается то же содержимое, что и в выходном списке команды `SELECT` (см. Раздел 7.3). Оно может содержать имена столбцов целевой таблицы команды или значения выражений с этими столбцами. Также часто применяется краткая запись `RETURNING *`, выбирающая все столбцы целевой таблицы по порядку.

В команде `INSERT` данные, выдаваемые в `RETURNING`, образуются из строки в том виде, в каком она была вставлена. Это не очень полезно при простом добавлении, так как в результате будут получены те же данные, что были переданы клиентом. Но это может быть очень удобно при использовании вычисляемых значений по умолчанию. Например, если в таблице есть столбец `serial`, в котором генерируются уникальные идентификаторы, команда `RETURNING` может возвратить идентификатор, назначенный новой строке:

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

Предложение `RETURNING` также очень полезно с `INSERT ... SELECT`.

В команде `UPDATE` данные, выдаваемые в `RETURNING`, образуются новым содержимым изменённой строки. Например:

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

В команде `DELETE` данные, выдаваемые в `RETURNING`, образуются содержимым удалённой строки. Например:

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
```

```
  RETURNING *;
```

В команде `MERGE` `RETURNING` позволяет получить данные, которые представляют собой содержимое исходной строки плюс содержимое вставленной, изменённой или удалённой целевой строки. Поскольку исходная и целевая строки часто имеют много одинаковых столбцов, указание `RETURNING *` может привести к появлению большого количества дублирующихся столбцов, поэтому лучше дополнить его, чтобы получить только исходную или целевую строку. Например:

```
MERGE INTO products p USING new_products n ON p.product_no = n.product_no
  WHEN NOT MATCHED THEN INSERT VALUES (n.product_no, n.name, n.price)
  WHEN MATCHED THEN UPDATE SET name = n.name, price = n.price
  RETURNING p.*;
```

Если для целевой таблицы заданы триггеры (см. Главу 37), в `RETURNING` выдаются данные из строки, изменённой триггерами. Таким образом, `RETURNING` часто применяется и для того, чтобы проверить содержимое столбцов, изменяемых триггерами.

Глава 7. Запросы

В предыдущих главах рассказывалось, как создать таблицы, как заполнить их данными и как изменить эти данные. Теперь мы наконец обсудим, как получить данные из базы данных.

7.1. Обзор

Процесс или команда получения данных из базы данных называется *запросом*. В SQL запросы формулируются с помощью команды `SELECT`. В общем виде команда `SELECT` записывается так:

```
[WITH запросы_with] SELECT список_выборки FROM табличное_выражение
 [определение_сортировки]
```

В следующих разделах подробно описываются список выборки, табличное выражение и определение сортировки. Запросы `WITH` являются расширенной возможностью PostgreSQL и будут рассмотрены в последнюю очередь.

Простой запрос выглядит так:

```
SELECT * FROM table1;
```

Если предположить, что в базе данных есть таблица `table1`, эта команда получит все строки с содержимым всех столбцов из `table1`. (Метод выдачи результата определяет клиентское приложение. Например, программа psql выведет на экране ASCII-таблицу, хотя клиентские библиотеки позволяют извлекать отдельные значения из результата запроса.) Здесь список выборки задан как `*`, это означает, что запрос должен вернуть все столбцы табличного выражения. В списке выборки можно также указать подмножество доступных столбцов или составить выражения с этими столбцами. Например, если в `table1` есть столбцы `a`, `b` и `c` (и возможно, другие), вы можете выполнить следующий запрос:

```
SELECT a, b + c FROM table1;
```

(в предположении, что столбцы `b` и `c` имеют числовой тип данных). Подробнее это описано в Разделе 7.3.

`FROM table1` — это простейший тип табличного выражения, в котором просто читается одна таблица. Вообще табличные выражения могут быть сложными конструкциями из базовых таблиц, соединений и подзапросов. А можно и вовсе опустить табличное выражение и использовать команду `SELECT` как калькулятор:

```
SELECT 3 * 4;
```

В этом может быть больше смысла, когда выражения в списке выборки возвращают меняющиеся результаты. Например, можно вызвать функцию так:

```
SELECT random();
```

7.2. Табличные выражения

Табличное выражение вычисляет таблицу. Это выражение содержит предложение `FROM`, за которым могут следовать предложения `WHERE`, `GROUP BY` и `HAVING`. Тривиальные табличные выражения просто ссылаются на физическую таблицу, её называют также базовой, но в более сложных выражениях такие таблицы можно преобразовывать и комбинировать самыми разными способами.

Необязательные предложения `WHERE`, `GROUP BY` и `HAVING` в табличном выражении определяют последовательность преобразований, осуществляемых с данными таблицы, полученной в предложении `FROM`. В результате этих преобразований образуется виртуальная таблица, строки которой передаются списку выборки, вычисляющему выходные строки запроса.

7.2.1. Предложение `FROM`

Предложение `FROM` образует таблицу из одной или нескольких ссылок на таблицы, разделённых запятыми.

```
FROM табличная_ссылка [, табличная_ссылка [, ...]]
```

Здесь табличной ссылкой может быть имя таблицы (возможно, с именем схемы), производная таблица, например подзапрос, соединение таблиц или сложная комбинация этих вариантов. Если в предложении `FROM` перечисляются несколько ссылок, для них применяется перекрёстное соединение (то есть декартово произведение их строк; см. ниже). Список `FROM` преобразуется в промежуточную виртуальную таблицу, которая может пройти через преобразования `WHERE`, `GROUP BY` и `HAVING`, и в итоге определит результат табличного выражения.

Когда в табличной ссылке указывается таблица, являющаяся родительской в иерархии наследования, в результате будут получены строки не только этой таблицы, но и всех её дочерних таблиц. Чтобы выбрать строки только одной родительской таблицы, перед её именем нужно добавить ключевое слово `ONLY`. Учтите, что при этом будут получены только столбцы указанной таблицы — дополнительные столбцы дочерних таблиц не попадут в результат.

Если же вы не добавляете `ONLY` перед именем таблицы, вы можете дописать после него `*`, тем самым указав, что должны обрабатываться и все дочерние таблицы. Практических причин использовать этот синтаксис больше нет, так как поиск в дочерних таблицах теперь производится по умолчанию. Однако эта запись поддерживается для совместимости со старыми версиями.

7.2.1.1. Соединённые таблицы

Соединённая таблица — это таблица, полученная из двух других (реальных или производных от них) таблиц в соответствии с правилами соединения конкретного типа. Общий синтаксис описания соединённой таблицы:

```
T1 тип_соединения T2 [ условие_соединения ]
```

Соединения любых типов могут вкладываться друг в друга или объединяться: и *`T1`*, и *`T2`* могут быть результатами соединения. Для однозначного определения порядка соединений предложения `JOIN` можно заключать в скобки. Если скобки отсутствуют, предложения `JOIN` обрабатываются слева направо.

Типы соединений

Перекрёстное соединение

```
T1 CROSS JOIN T2
```

Соединённую таблицу образуют все возможные сочетания строк из *`T1`* и *`T2`* (т. е. их декартово произведение), а набор её столбцов объединяет в себе столбцы *`T1`* со следующими за ними столбцами *`T2`*. Если таблицы содержат N и M строк, соединённая таблица будет содержать N * M строк.

`FROM T1 CROSS JOIN T2` равнозначно `FROM T1 INNER JOIN T2 ON TRUE` (см. ниже). Эта запись также равнозначна `FROM T1, T2`.

Примечание

Последняя запись не полностью эквивалентна первым при указании более чем двух таблиц, так как `JOIN` связывает таблицы сильнее, чем запятая. Например, `FROM T1 CROSS JOIN T2 INNER JOIN T3 ON условие` не равнозначно `FROM T1, T2 INNER JOIN T3 ON условие`, так как *`условие`* может ссылаться на *`T1`* в первом случае, но не во втором.

Соединения с сопоставлениями строк

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON логическое_выражение
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( список столбцов
 соединения )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

Слова `INNER` и `OUTER` необязательны во всех формах. По умолчанию подразумевается `INNER` (внутреннее соединение), а при указании `LEFT`, `RIGHT` и `FULL` — внешнее соединение.

Условие соединения указывается в предложении `ON` или `USING`, либо неявно задаётся ключевым словом `NATURAL`. Это условие определяет, какие строки двух исходных таблиц считаются «соответствующими» друг другу (это подробно рассматривается ниже).

Возможные типы соединений с сопоставлениями строк:

`INNER JOIN`

Для каждой строки R1 из T1 в результирующей таблице содержится строка для каждой строки в T2, удовлетворяющей условию соединения с R1.

`LEFT OUTER JOIN`

Сначала выполняется внутреннее соединение (INNER JOIN). Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений столбцов T2 вставляются NULL. Таким образом, в результирующей таблице всегда будет минимум одна строка для каждой строки из T1.

`RIGHT OUTER JOIN`

Сначала выполняется внутреннее соединение (INNER JOIN). Затем в результат добавляются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений столбцов T1 вставляются NULL. Это соединение является обратным к левому (LEFT JOIN): в результирующей таблице всегда будет минимум одна строка для каждой строки из T2.

`FULL OUTER JOIN`

Сначала выполняется внутреннее соединение. Затем в результат добавляются все строки из T1, которым не соответствуют никакие строки в T2, а вместо значений столбцов T2 вставляются NULL. И наконец, в результат включаются все строки из T2, которым не соответствуют никакие строки в T1, а вместо значений столбцов T1 вставляются NULL.

Предложение `ON` определяет наиболее общую форму условия соединения: в нём указываются выражения логического типа, подобные тем, что используются в предложении `WHERE`. Пара строк из *`T1`* и *`T2`* соответствуют друг другу, если выражение `ON` возвращает для них true.

`USING` — это сокращённая запись условия, полезная в ситуации, когда с обеих сторон соединения столбцы имеют одинаковые имена. Она принимает список общих имён столбцов через запятую и формирует условие соединения с равенством этих столбцов. Например, запись соединения *`T1`* и *`T2`* с `USING (a, b)` формирует условие `ON T1.a = T2.a AND T1.b = T2.b`.

Более того, при выводе `JOIN USING` исключаются избыточные столбцы: оба сопоставленных столбца выводить не нужно, так как они содержат одинаковые значения. Тогда как `JOIN ON` выдаёт все столбцы из *`T1`*, а за ними все столбцы из *`T2`*, `JOIN USING` выводит один столбец для каждой пары (в указанном порядке), за ними все оставшиеся столбцы из *`T1`* и, наконец, все оставшиеся столбцы *`T2`*.

Наконец, `NATURAL` — сокращённая форма `USING`: она образует список `USING` из всех имён столбцов, существующих в обеих входных таблицах. Как и с `USING`, эти столбцы оказываются в выходной таблице в единственном экземпляре. Если столбцов с одинаковыми именами не находится, `NATURAL JOIN` действует как `CROSS JOIN`.

Примечание

Предложение `USING` разумно защищено от изменений в соединяемых отношениях, так как оно связывает только явно перечисленные столбцы. `NATURAL` считается более рискованным, так как при любом изменении схемы в одном или другом отношении, когда по-

> являются столбцы с совпадающими именами, при соединении будут связываться и эти новые столбцы.

Для наглядности предположим, что у нас есть таблицы `t1`:

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

и `t2`:

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

С ними для разных типов соединений мы получим следующие результаты:

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
```

```
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
     |      |   5 | zzz
(4 rows)
```

Условие соединения в предложении `ON` может также содержать выражения, не связанные непосредственно с соединением. Это может быть полезно в некоторых запросах, но не следует использовать это необдуманно. Рассмотрите следующий запрос:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

Заметьте, что если поместить ограничение в предложение `WHERE`, вы получите другой результат:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

Это связано с тем, что ограничение, помещённое в предложение `ON`, обрабатывается *до* операции соединения, тогда как ограничение в `WHERE` — *после*. Это не имеет значения при внутренних соединениях, но важно при внешних.

7.2.1.2. Псевдонимы таблиц и столбцов

Таблицам и ссылкам на сложные таблицы в запросе можно дать временное имя, по которому к ним можно будет обращаться в рамках запроса. Такое имя называется *псевдонимом таблицы*.

Определить псевдоним таблицы можно, написав

```
FROM табличная_ссылка AS псевдоним
```

или

```
FROM табличная_ссылка псевдоним
```

Ключевое слово `AS` является необязательным. Вместо *`псевдоним`* здесь может быть любой идентификатор.

Псевдонимы часто применяются для назначения коротких идентификаторов длинным именам таблиц с целью улучшения читаемости запросов. Например:

```
SELECT * FROM "очень_длинное_имя_таблицы" s JOIN "другое_длинное_имя" a ON s.id =
 a.num;
```

Псевдоним становится новым именем таблицы в рамках текущего запроса, т. е. после назначения псевдонима использовать исходное имя таблицы в другом месте запроса нельзя. Таким образом, следующий запрос недопустим:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- неправильно
```

Хотя в основном псевдонимы используются для удобства, они бывают необходимы, когда таблица соединяется сама с собой, например:

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

В случае неоднозначности определения псевдонимов можно использовать скобки. В следующем примере первый оператор назначает псевдоним `b` второму экземпляру `my_table`, а второй оператор назначает псевдоним результату соединения:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

В другой форме назначения псевдонима временные имена даются не только таблицам, но и её столбцам:

```
FROM табличная_ссылка [AS] псевдоним ( столбец1 [, столбец2 [, ...]] )
```

Если псевдонимов столбцов оказывается меньше, чем фактически столбцов в таблице, остальные столбцы сохраняют свои исходные имена. Эта запись особенно полезна для замкнутых соединений или подзапросов.

Когда псевдоним применяется к результату `JOIN`, он скрывает оригинальные имена таблиц внутри `JOIN`. Например, это допустимый SQL-запрос:

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

а запрос:

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

ошибочный, так как псевдоним таблицы `a` не виден снаружи определения псевдонима `c`.

7.2.1.3. Подзапросы

Подзапросы, образующие таблицы, должны заключаться в скобки. Им могут назначаться псевдонимы таблиц и, если необходимо, псевдонимы столбцов (как описано в Подразделе 7.2.1.2). Например:

```
FROM (SELECT * FROM table1) AS псевдоним
```

Этот пример равносилен записи `FROM table1 AS псевдоним`. Более интересные ситуации, которые нельзя свести к простому соединению, возникают, когда в подзапросе используются агрегирующие функции или группировка.

Подзапросом может также быть список `VALUES`:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

Такому подзапросу может быть назначен псевдоним. Назначать псевдонимы столбцам списка `VALUES` не требуется, но вообще это хороший приём. Подробнее это описано в Разделе 7.7.

Согласно стандарту SQL подзапросу должен назначаться псевдоним таблицы. Postgres Pro позволяет опускать `AS` и псевдоним, тем не менее хорошей практикой считается указывать псевдоним в коде SQL, который может быть портирован в другую систему.

7.2.1.4. Табличные функции

Табличные функции — это функции, выдающие набор строк, содержащих либо базовые типы данных (скалярных типов), либо составные типы (табличные строки). Они применяются в запросах как таблицы, представления или подзапросы в предложении `FROM`. Столбцы, возвращённые табличными функциями, можно включить в выражения `SELECT`, `JOIN` или `WHERE` так же, как столбцы таблиц, представлений или подзапросов.

Табличные функции можно также скомбинировать, используя запись `ROWS FROM`. Результаты функций будут возвращены в параллельных столбцах; число строк в этом случае будет наибольшим из результатов всех функций, а результаты функций с меньшим количеством строк будут дополнены значениями NULL.

```
вызов_функции [WITH ORDINALITY] [[AS] псевдоним_таблицы [(псевдоним_столбца [, ...])]]
ROWS FROM( вызов_функции [, ...] ) [WITH ORDINALITY] [[AS] псевдоним_таблицы
 [(псевдоним_столбца [, ...])]]
```

Если указано предложение `WITH ORDINALITY`, к столбцам результатов функций будет добавлен ещё один, с типом `bigint`. В этом столбце нумеруются строки результирующего набора, начиная с 1. (Это обобщение стандартного SQL-синтаксиса `UNNEST ... WITH ORDINALITY`.) По умолчанию, этот столбец называется `ordinality`, но ему можно присвоить и другое имя с помощью указания `AS`.

Специальную табличную функцию `UNNEST` можно вызвать с любым числом параметров-массивов, а возвращает она соответствующее число столбцов, как если бы `UNNEST` (Раздел 9.19) вызывалась для каждого параметра в отдельности, а результаты объединялись с помощью конструкции `ROWS FROM`.

```
UNNEST( выражение_массива [, ...] ) [WITH ORDINALITY] [[AS] псевдоним_таблицы
 [(псевдоним_столбца [, ...])]]
```

Если *`псевдоним_таблицы`* не указан, в качестве имени таблицы используется имя функции; в случае с конструкцией `ROWS FROM()` — имя первой функции.

Если псевдонимы столбцов не указаны, то для функции, возвращающей базовый тип данных, именем столбца будет имя функции. Для функций, возвращающих составной тип, имена результирующих столбцов определяются индивидуальными атрибутами типа.

Несколько примеров:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);
```

```
SELECT * FROM vw_getfoo;
```

В некоторых случаях бывает удобно определить табличную функцию, возвращающую различные наборы столбцов при разных вариантах вызова. Это можно сделать, объявив функцию, не имеющую выходных параметров (`OUT`) и возвращающую псевдотип `record`. Используя такую функцию, ожидаемую структуру строк нужно описать в самом запросе, чтобы система знала, как разобрать запрос и составить его план. Записывается это так:

```
вызов_функции [AS] псевдоним (определение_столбца [, ...])
вызов_функции AS [псевдоним] (определение_столбца [, ...])
ROWS FROM( ... вызов_функции AS (определение_столбца [, ...]) [, ...] )
```

Без `ROWS FROM()` список *определения_столбцов* заменяет список псевдонимов, который можно также добавить в предложении `FROM`; имена в определениях столбцов служат псевдонимами. С `ROWS FROM()` список *определения_столбцов* можно добавить к каждой функции отдельно, либо в случае с одной функцией и без предложения `WITH ORDINALITY`, список *определения_столбцов* можно записать вместо списка с псевдонимами столбцов после `ROWS FROM()`.

Взгляните на этот пример:

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

Здесь функция dblink (из модуля dblink) выполняет удалённый запрос. Она объявлена как функция, возвращающая тип `record`, так как он подойдёт для запроса любого типа. В этом случае фактический набор столбцов функции необходимо описать в вызывающем её запросе, чтобы анализатор запроса знал, например, как преобразовать `*`.

В этом примере используется конструкция `ROWS FROM`:

```
SELECT *
FROM ROWS FROM
    (
        json_to_recordset('[{"a":40,"b":"foo"},{"a":"100","b":"bar"}]')
            AS (a INTEGER, b TEXT),
        generate_series(1, 3)
    ) AS x (p, q, s)
ORDER BY p;

  p  |  q  | s
-----+-----+---
  40 | foo | 1
 100 | bar | 2
     |     | 3
```

Она объединяет результаты двух функций в одном отношении `FROM`. В данном случае `json_to_recordset()` должна выдавать два столбца, первый `integer` и второй `text`, а результат `generate_series()` используется непосредственно. Предложение `ORDER BY` упорядочивает значения первого столбца как целочисленные.

7.2.1.5. Подзапросы `LATERAL`

Перед подзапросами в предложении `FROM` можно добавить ключевое слово `LATERAL`. Это позволит ссылаться в них на столбцы предшествующих элементов списка `FROM`. (Без `LATERAL` каждый подзапрос выполняется независимо и поэтому не может обращаться к другим элементам `FROM`.)

Перед табличными функциями в предложении `FROM` также можно указать `LATERAL`, но для них это ключевое слово необязательно; в аргументах функций в любом случае можно обращаться к столбцам в предыдущих элементах `FROM`.

Элемент `LATERAL` может находиться на верхнем уровне списка `FROM` или в дереве `JOIN`. В последнем случае он может также ссылаться на любые элементы в левой части `JOIN`, справа от которого он находится.

Когда элемент `FROM` содержит ссылки `LATERAL`, запрос выполняется следующим образом: сначала для строки элемента `FROM` с целевыми столбцами, или набора строк из нескольких элементов `FROM`, содержащих целевые столбцы, вычисляется элемент `LATERAL` со значениями этих столбцов. Затем результирующие строки обычным образом соединяются со строками, из которых они были вычислены. Эта процедура повторяется для всех строк исходных таблиц.

`LATERAL` можно использовать так:

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

Здесь это не очень полезно, так как тот же результат можно получить более простым и привычным способом:

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

Применять `LATERAL` имеет смысл в основном, когда для вычисления соединяемых строк необходимо обратиться к столбцам других таблиц. В частности, это полезно, когда нужно передать значение функции, возвращающей набор данных. Например, если предположить, что `vertices(polygon)` возвращает набор вершин многоугольника, близкие вершины многоугольников из таблицы polygons можно получить так:

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

Этот запрос можно записать и так:

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

или переформулировать другими способами. (Как уже упоминалось, в данном примере ключевое слово `LATERAL` не требуется, но мы добавили его для ясности.)

Особенно полезно бывает использовать `LEFT JOIN` с подзапросом `LATERAL`, чтобы исходные строки оказывались в результате, даже если подзапрос `LATERAL` не возвращает строк. Например, если функция `get_product_names()` выдаёт названия продуктов, выпущенных определённым производителем, но о продукции некоторых производителей информации нет, мы можем найти, каких именно, примерно так:

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

7.2.2. Предложение `WHERE`

Предложение `WHERE` записывается так:

```
WHERE условие_ограничения
```

где *`условие_ограничения`* — любое выражение значения (см. Раздел 4.2), выдающее результат типа `boolean`.

После обработки предложения `FROM` каждая строка полученной виртуальной таблицы проходит проверку по условию ограничения. Если результат условия равен true, эта строка остаётся в выходной таблице, а иначе (если результат равен false или NULL) отбрасывается. В условии ограничения, как правило, задействуется минимум один столбец из таблицы, полученной на выходе `FROM`.

Хотя строго говоря, это не требуется, но в противном случае предложение `WHERE` будет бессмысленным.

Примечание

Условие для внутреннего соединения можно записать как в предложении `WHERE`, так и в предложении `JOIN`. Например, это выражение:

```
FROM a, b WHERE a.id = b.id AND b.val > 5
```

равнозначно этому:

```
FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
```

и возможно, даже этому:

```
FROM a NATURAL JOIN b WHERE b.val > 5
```

Какой вариант выбрать, в основном дело вкуса и стиля. Вариант с `JOIN` внутри предложения `FROM`, возможно, не лучший с точки зрения совместимости с другими СУБД, хотя он и описан в стандарте SQL. Но для внешних соединений других вариантов нет: их можно записывать только во `FROM`. Предложения `ON` и `USING` во внешних соединениях *не* равнозначны условию `WHERE`, так как они могут добавлять строки (для входных строк без соответствия), а также удалять их из конечного результата.

Несколько примеров запросов с `WHERE`:

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

`fdt` — название таблицы, порождённой в предложении `FROM`. Строки, которые не соответствуют условию `WHERE`, исключаются из `fdt`. Обратите внимание, как в качестве выражений значения используются скалярные подзапросы. Как и любые другие запросы, подзапросы могут содержать сложные табличные выражения. Заметьте также, что `fdt` используется в подзапросах. Дополнение имени `c1` в виде `fdt.c1` необходимо только, если в порождённой таблице в подзапросе также оказывается столбец `c1`. Полное имя придаёт ясность даже там, где без него можно обойтись. Этот пример показывает, как область именования столбцов внешнего запроса распространяется на все вложенные в него внутренние запросы.

7.2.3. Предложения `GROUP BY` и `HAVING`

Строки порождённой входной таблицы, прошедшие фильтр `WHERE`, можно сгруппировать с помощью предложения `GROUP BY`, а затем оставить в результате только нужные группы строк, используя предложение `HAVING`.

```
SELECT список_выборки
    FROM ...
    [WHERE ...]
    GROUP BY группирующий_столбец [, группирующий_столбец]...
```

Предложение `GROUP BY` группирует строки таблицы, объединяя их в одну группу при совпадении значений во всех перечисленных столбцах. Порядок, в котором указаны столбцы, не имеет значения. В результате наборы строк с одинаковыми значениями преобразуются в отдельные строки, представляющие все строки группы. Это может быть полезно для устранения избыточности выходных данных и/или для вычисления агрегатных функций, применённых к этим группам. Например:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

Во втором запросе мы не могли написать `SELECT * FROM test1 GROUP BY x`, так как для столбца `y` нет единого значения, связанного с каждой группой. Однако столбцы, по которым выполняется группировка, можно использовать в списке выборки, так как они имеют единственное значение в каждой группе.

Вообще говоря, в группированной таблице столбцы, не включённые в список `GROUP BY`, можно использовать только в агрегатных выражениях. Пример такого агрегатного выражения:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Здесь `sum` — агрегатная функция, вычисляющая единственное значение для всей группы. Подробную информацию о существующих агрегатных функциях можно найти в Разделе 9.21.

Подсказка

Группировка без агрегатных выражений по сути выдаёт набор различающихся значений столбцов. Этот же результат можно получить с помощью предложения `DISTINCT` (см. Подраздел 7.3.3).

Взгляните на следующий пример: в нём вычисляется общая сумма продаж по каждому продукту (а не общая сумма по всем продуктам):

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

В этом примере столбцы `product_id`, `p.name` и `p.price` должны присутствовать в списке `GROUP BY`, так как они используются в списке выборки. Столбец `s.units` может отсутствовать в списке `GROUP BY`, так как он используется только в агрегатном выражении (`sum(...)`), вычисляющем сумму продаж. Для каждого продукта этот запрос возвращает строку с итоговой суммой по всем продажам данного продукта.

Если бы в таблице products по столбцу `product_id` был создан первичный ключ, тогда в данном примере было бы достаточно сгруппировать строки по `product_id`, так как название и цена продукта *функционально зависят* от кода продукта и можно однозначно определить, какое название и цену возвращать для каждой группы по ID.

В стандарте SQL `GROUP BY` может группировать только по столбцам исходной таблицы, но расширение Postgres Pro позволяет использовать в `GROUP BY` столбцы из списка выборки. Также возможна группировка по выражениям, а не просто именам столбцов.

Если таблица была сгруппирована с помощью `GROUP BY`, но интерес представляют только некоторые группы, отфильтровать их можно с помощью предложения `HAVING`, действующего подобно `WHERE`. Записывается это так:

```
SELECT список_выборки FROM ... [WHERE ...] GROUP BY ... HAVING логическое_выражение
```

В предложении `HAVING` могут использоваться и группирующие выражения, и выражения, не участвующие в группировке (в этом случае это должны быть агрегирующие функции).

Пример:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

И ещё один более реалистичный пример:

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

В данном примере предложение `WHERE` выбирает строки по столбцу, не включённому в группировку (выражение истинно только для продаж за последние четыре недели), тогда как предложение `HAVING` отфильтровывает группы с общей суммой продаж больше 5000. Заметьте, что агрегатные выражения не обязательно должны быть одинаковыми во всех частях запроса.

Если в запросе есть вызовы агрегатных функций, но нет предложения `GROUP BY`, строки всё равно будут группироваться: в результате окажется одна строка группы (или возможно, ни одной строки, если эта строка будет отброшена предложением `HAVING`). Это справедливо и для запросов, которые содержат только предложение `HAVING`, но не содержат вызовы агрегатных функций и предложение `GROUP BY`.

7.2.4. `GROUPING SETS`, `CUBE` и `ROLLUP`

Более сложные, чем описанные выше, операции группировки возможны с концепцией *наборов группирования*. Данные, выбранные предложениями `FROM` и `WHERE`, группируются отдельно для каждого заданного набора группирования, затем для каждой группы вычисляются агрегатные функции как для простых предложений `GROUP BY`, и в конце возвращаются результаты. Например:

```
=> SELECT * FROM items_sold;
```

```
 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)

=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING SETS ((brand),
 (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
(5 rows)
```

В каждом внутреннем списке `GROUPING SETS` могут задаваться ноль или более столбцов или выражений, которые воспринимаются так же, как если бы они были непосредственно записаны в предложении `GROUP BY`. Пустой набор группировки означает, что все строки сводятся к одной группе (которая выводится, даже если входных строк нет), как описано выше для агрегатных функций без предложения `GROUP BY`.

Ссылки на группирующие столбцы или выражения заменяются в результирующих строках значениями NULL для тех группирующих наборов, в которых эти столбцы отсутствуют. Чтобы можно было понять, результатом какого группирования стала конкретная выходная строка, предназначена функция, описанная в Таблице 9.64.

Для указания двух распространённых видов наборов группирования предусмотрена краткая запись. Предложение формы

```
ROLLUP ( e1, e2, e3, ... )
```

представляет заданный список выражений и всех префиксов списка, включая пустой список; то есть оно равнозначно записи

```
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
    ( )
)
```

Оно часто применяется для анализа иерархических данных, например, для суммирования зарплаты по отделам, подразделениям и компании в целом.

Предложение формы

```
CUBE ( e1, e2, ... )
```

представляет заданный список и все его возможные подмножества (степень множества). Таким образом, запись

```
CUBE ( a, b, c )
```

равнозначна

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b    ),
```

```
    ( a,    c ),
    ( a       ),
    (    b, c ),
    (    b    ),
    (       c ),
    (         )
)
```

Элементами предложений `CUBE` и `ROLLUP` могут быть либо отдельные выражения, либо вложенные списки элементов в скобках. Вложенные списки обрабатываются как атомарные единицы, с которыми формируются отдельные наборы группирования. Например:

```
CUBE ( (a, b), (c, d) )
```

равнозначно

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (       c, d ),
    (            )
)
```

и

```
ROLLUP ( a, (b, c), d )
```

равнозначно

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

Конструкции `CUBE` и `ROLLUP` могут применяться либо непосредственно в предложении `GROUP BY`, либо вкладываться внутрь предложения `GROUPING SETS`. Если одно предложение `GROUPING SETS` вкладывается внутрь другого, результат будет таким же, как если бы все элементы внутреннего предложения были записаны непосредственно во внешнем.

Если в одном предложении `GROUP BY` задаётся несколько элементов группирования, окончательный список наборов группирования образуется как декартово произведение этих элементов. Например:

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

равнозначно

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

Если задаётся несколько элементов группирования, окончательный список наборов группирования может содержать дублирующиеся результаты. Например:

```
GROUP BY ROLLUP (a, b), ROLLUP (a, c)
```

равнозначно

```
GROUP BY GROUPING SETS (
```

```
    (a, b, c),
    (a, b),
    (a, b),
    (a, c),
    (a),
    (a),
    (a, c),
    (a),
    ()
)
```

Если эти дублирующиеся результаты нежелательны, их можно убрать, добавив `DISTINCT` непосредственно в предложение `GROUP BY`. Таким образом:

```
GROUP BY DISTINCT ROLLUP (a, b), ROLLUP (a, c)
```

равнозначно

```
GROUP BY GROUPING SETS (
    (a, b, c),
    (a, b),
    (a, c),
    (a),
    ()
)
```

Это не то же самое, что `SELECT DISTINCT`, поскольку результирующие строки всё же могут содержать дублирующиеся записи. Если любой из столбцов, не участвующих в группировании, содержит значение NULL, в результате оно не будет отличаться от NULL, полученного при группировании этого столбца.

Примечание

Конструкция `(a, b)` обычно воспринимается в выражениях как конструктор строки. Однако в предложении `GROUP BY` на верхнем уровне выражений запись `(a, b)` воспринимается как список выражений, как описано выше. Если вам по какой-либо причине *нужен* именно конструктор строки в выражении группирования, используйте запись `ROW(a, b)`.

7.2.5. Обработка оконных функций

Если запрос содержит оконные функции (см. Раздел 3.5, Раздел 9.22 и Подраздел 4.2.8), эти функции вычисляются после каждой группировки, агрегатных выражений и фильтрации `HAVING`. Другими словами, если в запросе есть агрегатные функции, предложения `GROUP BY` или `HAVING`, оконные функции видят не исходные строки, полученные из `FROM`/`WHERE`, а сгруппированные.

Когда используются несколько оконных функций, все оконные функции, имеющие в своих определениях равнозначные предложения `PARTITION BY` и `ORDER BY`, гарантированно видят один порядок сортировки входных строк, даже если `ORDER BY` не определяет этот порядок однозначно. Однако относительно функций с разными формулировками `PARTITION BY` и `ORDER BY` никаких гарантий не даётся. (В таких случаях между проходами вычислений оконных функций обычно требуется дополнительный этап сортировки и эта сортировка может не сохранять порядок строк, равнозначный с точки зрения `ORDER BY`.)

В настоящее время оконные функции всегда требуют предварительно отсортированных данных, так что результат запроса будет отсортирован согласно тому или иному предложению `PARTITION BY`/`ORDER BY` оконных функций. Однако полагаться на это не следует. Если вы хотите, чтобы результаты сортировались определённым образом, явно добавьте предложение `ORDER BY` на верхнем уровне запроса.

7.3. Списки выборки

Как говорилось в предыдущем разделе, табличное выражение в `SELECT` создаёт промежуточную виртуальную таблицу, возможно объединяя таблицы, представления, группируя и исключая лишние строки и т. д. Полученная таблица передаётся для обработки в *список выборки*. Этот список выбирает, какие *столбцы* промежуточной таблицы должны выводиться в результате и как именно.

7.3.1. Элементы списка выборки

Простейший список выборки образует элемент `*`, который выбирает все столбцы из полученного табличного выражения. Список выборки также может содержать список выражений значения через запятую (как определено в Разделе 4.2). Например, это может быть список имён столбцов:

```
SELECT a, b, c FROM ...
```

Имена столбцов `a`, `b` и `c` представляют либо фактические имена столбцов таблиц, перечисленных в предложении `FROM`, либо их псевдонимы, определённые как описано в Подразделе 7.2.1.2. Пространство имён в списке выборки то же, что и в предложении `WHERE`, если не используется группировка. В противном случае оно совпадает с пространством имён предложения `HAVING`.

Если столбец с заданным именем есть в нескольких таблицах, необходимо также указать имя таблицы, например так:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

Обращаясь к нескольким таблицам, бывает удобно получить сразу все столбцы одной из таблиц:

```
SELECT tbl1.*, tbl2.a FROM ...
```

Подробнее запись `*имя_таблицы*.*` описывается в Подразделе 8.16.5.

Если в списке выборки используется обычное выражение значения, по сути при этом в возвращаемую таблицу добавляется новый виртуальный столбец. Выражение значения вычисляется один раз для каждой строки результата со значениями столбцов в данной строке. Хотя выражения в списке выборки не обязательно должны обращаться к столбцам табличного выражения из предложения `FROM`; они могут содержать, например и простые арифметические выражения.

7.3.2. Метки столбцов

Элементам в списке выборки можно назначить имена для последующей обработки, например, для указания в предложении `ORDER BY` или для вывода в клиентском приложении. Например:

```
SELECT a AS value, b + c AS sum FROM ...
```

Если выходное имя столбца не определено (с помощью `AS`), система назначает имя сама. Для простых ссылок на столбцы этим именем становится имя целевого столбца, а для вызовов функций это имя функции. Для сложных выражений система генерирует некоторое подходящее имя.

Слово `AS` обычно можно опустить, но в некоторых случаях, когда желаемое имя столбца совпадает с ключевым словом Postgres Pro, нужно написать `AS` или заключить имя столбца в кавычки во избежание неоднозначности. (В Приложении C перечислены ключевые слова, для которых нужно использовать `AS` в метке столбца.) Например, `FROM` — ключевое слово, поэтому такой вариант не будет работать:

```
SELECT a from, b + c AS sum FROM ...
```

а любой из этих сработает:

```
SELECT a AS from, b + c AS sum FROM ...
SELECT a "from", b + c AS sum FROM ...
```

Для предотвращения конфликта с ключевыми словами, которые могут появиться в будущем, рекомендуется всегда писать `AS` или заключать метки выходных столбцов в кавычки.

Примечание

Именование выходных столбцов отличается от того, что происходит в предложении `FROM` (см. Подраздел 7.2.1.2). Один столбец можно переименовать дважды, но на выходе окажется имя, назначенное в списке выборки.

7.3.3. `DISTINCT`

После обработки списка выборки в результирующей таблице можно дополнительно исключить дублирующиеся строки. Для этого сразу после `SELECT` добавляется ключевое слово `DISTINCT`:

```
SELECT DISTINCT список_выборки ...
```

(Чтобы явно включить поведение по умолчанию, когда возвращаются все строки, вместо `DISTINCT` можно указать ключевое слово `ALL`.)

Две строки считаются разными, если они содержат различные значения минимум в одном столбце. При этом значения NULL полагаются равными.

Кроме того, можно явно определить, какие строки будут считаться различными, следующим образом:

```
SELECT DISTINCT ON (выражение [, выражение ...]) список_выборки ...
```

Здесь *выражение* — обычное выражение значения, вычисляемое для всех строк. Строки, для которых перечисленные выражения дают один результат, считаются дублирующимися и возвращается только первая строка из такого набора. Заметьте, что «первая строка» набора может быть любой, если только запрос не включает сортировку, гарантирующую однозначный порядок строк, поступающих в фильтр `DISTINCT`. (Обработка `DISTINCT ON` производится после сортировки `ORDER BY`.)

Предложение `DISTINCT ON` не описано в стандарте SQL и иногда его применение считается плохим стилем из-за возможной неопределённости в результатах. При разумном использовании `GROUP BY` и подзапросов во `FROM` можно обойтись без этой конструкции, но часто она бывает удобнее.

7.4. Сочетание запросов (`UNION`, `INTERSECT`, `EXCEPT`)

Результаты двух запросов можно обработать, используя операции над множествами: объединение, пересечение и вычитание. Эти операции записываются соответственно так:

```
запрос1 UNION [ALL] запрос2
запрос1 INTERSECT [ALL] запрос2
запрос1 EXCEPT [ALL] запрос2
```

Здесь *запрос1* и *запрос2* — это запросы, в которых могут использоваться все возможности, рассмотренные до этого.

`UNION` по сути добавляет результаты второго запроса к результатам первого (хотя никакой порядок возвращаемых строк при этом не гарантируется). Более того, эта операция убирает дублирующиеся строки из результата так же, как это делает `DISTINCT`, если только не указано `UNION ALL`.

`INTERSECT` возвращает все строки, содержащиеся в результате и первого, и второго запроса. Дублирующиеся строки отфильтровываются, если не указано `ALL`.

`EXCEPT` возвращает все строки, которые есть в результате первого запроса, но отсутствуют в результате второго. (Иногда это называют *разницей* двух запросов.) И здесь дублирующиеся строки отфильтровываются, если не указано `ALL`.

Чтобы можно было вычислить объединение, пересечение или разницу результатов двух запросов, эти запросы должны быть «совместимыми для объединения», что означает, что они должны иметь одинаковое число столбцов и соответствующие столбцы должны быть совместимых типов, как описывается в Разделе 10.5.

Операции над множествами можно объединять, например

```
запрос1 UNION запрос2 EXCEPT запрос3
```

что равнозначно

```
(запрос1 UNION запрос2) EXCEPT запрос3
```

Как показано выше, вы можете использовать круглые скобки, чтобы управлять очередью обработки. Без круглых скобок `UNION` и `EXCEPT` объединяются слева направо, но оператор `INTERSECT` имеет больший приоритет, чем первые два. Таким образом,

```
запрос1 UNION запрос2 INTERSECT запрос3
```

означает

```
запрос1 UNION (запрос2 INTERSECT запрос3)
```

Вы также можете заключить отдельный *`запрос`* в круглые скобки. Это важно, если в *`запросе`* необходимо использовать любое из предложений, обсуждаемых в следующих разделах, например `LIMIT`. Без круглых скобок возникнет синтаксическая ошибка или предложение будет восприниматься как относящееся к результату операции над множествами, а не к её аргументам. Например,

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10
```

не вызовет ошибку, но будет означать

```
(SELECT a FROM b UNION SELECT x FROM y) LIMIT 10
```

а не

```
SELECT a FROM b UNION (SELECT x FROM y LIMIT 10)
```

7.5. Сортировка строк (`ORDER BY`)

После того как запрос выдал таблицу результатов (после обработки списка выборки), её можно отсортировать. Если сортировка не задана, строки возвращаются в неопределённом порядке. Фактический порядок строк в этом случае будет зависеть от плана соединения и сканирования, а также от порядка данных на диске, поэтому полагаться на него нельзя. Определённый порядок выводимых строк гарантируется, только если этап сортировки задан явно.

Порядок сортировки определяет предложение `ORDER BY`:

```
SELECT список_выборки
    FROM табличное_выражение
    ORDER BY выражение_сортировки1 [ASC | DESC] [NULLS { FIRST | LAST }]
             [, выражение_сортировки2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

Выражениями сортировки могут быть любые выражения, допустимые в списке выборки запроса. Например:

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

Когда указывается несколько выражений, последующие значения позволяют отсортировать строки, в которых совпали все предыдущие значения. Каждое выражение можно дополнить ключевыми словами `ASC` или `DESC`, которые выбирают сортировку соответственно по возрастанию или убыванию. По умолчанию принят порядок по возрастанию (`ASC`). При сортировке по возрастанию сначала идут меньшие значения, где понятие «меньше» определяется оператором `<`. Подобным образом, сортировка по возрастанию определяется оператором `>`. [1]

Для определения места значений NULL можно использовать указания `NULLS FIRST` и `NULLS LAST`, которые помещают значения NULL соответственно до или после значений не NULL. По умолчанию значения NULL считаются больше любых других, то есть подразумевается `NULLS FIRST` для порядка `DESC` и `NULLS LAST` в противном случае.

---

[1] На деле Postgres Pro определяет порядок сортировки для `ASC` и `DESC` по *классу оператора B-дерева по умолчанию* для типа данных выражения. Обычно типы данных создаются так, что этому порядку соответствуют операторы `<` и `>`, но возможно разработать собственный тип данных, который будет вести себя по-другому.

Заметьте, что порядки сортировки определяются независимо для каждого столбца. Например, `ORDER BY x, y DESC` означает `ORDER BY x ASC, y DESC`, и это не то же самое, что `ORDER BY x DESC, y DESC`.

Здесь *`выражение_сортировки`* может быть меткой столбца или номером выводимого столбца, как в данном примере:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

Оба эти запроса сортируют результат по первому столбцу. Заметьте, что имя выводимого столбца должно оставаться само по себе, его нельзя использовать в выражении. Например, это *ошибка*:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- неправильно
```

Это ограничение позволяет уменьшить неоднозначность. Тем не менее неоднозначность возможна, когда в `ORDER BY` указано простое имя, но оно соответствует и имени выходного столбца, и столбцу из табличного выражения. В этом случае используется выходной столбец. Эта ситуация может возникнуть, только когда с помощью `AS` выходному столбцу назначается то же имя, что имеет столбец в другой таблице.

`ORDER BY` можно применить к результату комбинации `UNION`, `INTERSECT` и `EXCEPT`, но в этом случае возможна сортировка только по номерам или именам столбцов, но не по выражениям.

7.6. `LIMIT` и `OFFSET`

Указания `LIMIT` и `OFFSET` позволяют получить только часть строк из тех, что выдал остальной запрос:

```
SELECT список_выборки
    FROM табличное_выражение
    [ ORDER BY ... ]
    [ LIMIT { число | ALL } ]
    [ OFFSET начало ]
```

Если указывается число LIMIT, в результате возвращается не больше заданного числа строк (меньше может быть, если сам запрос выдал меньшее количество строк). `LIMIT ALL` равносильно отсутствию указания `LIMIT`, как и `LIMIT` с аргументом NULL.

`OFFSET` указывает пропустить указанное число строк, прежде чем начать выдавать строки. `OFFSET 0` равносильно отсутствию указания `OFFSET`, как и `OFFSET` с аргументом NULL.

Если указано и `OFFSET`, и `LIMIT`, сначала система пропускает `OFFSET` строк, а затем начинает подсчитывать строки для ограничения `LIMIT`.

Применяя `LIMIT`, важно использовать также предложение `ORDER BY`, чтобы строки результата выдавались в определённом порядке. Иначе будут возвращаться непредсказуемые подмножества строк. Вы можете запросить строки с десятой по двадцатую, но какой порядок вы имеете в виду? Порядок будет неизвестен, если не добавить `ORDER BY`.

Оптимизатор запроса учитывает ограничение `LIMIT`, строя планы выполнения запросов, поэтому вероятнее всего планы (а значит и порядок строк) будут меняться при разных `LIMIT` и `OFFSET`. Таким образом, различные значения `LIMIT`/`OFFSET`, выбирающие разные подмножества результатов запроса, *приведут к несогласованности результатов*, если не установить предсказуемую сортировку с помощью `ORDER BY`. Это не ошибка, а неизбежное следствие того, что SQL не гарантирует вывод результатов запроса в некотором порядке, если порядок не определён явно предложением `ORDER BY`.

Строки, пропускаемые согласно предложению `OFFSET`, тем не менее должны вычисляться на сервере. Таким образом, при больших значениях `OFFSET` работает неэффективно.

7.7. Списки `VALUES`

Предложение `VALUES` позволяет создать «постоянную таблицу», которую можно использовать в запросе, не создавая и не наполняя таблицу в БД. Синтаксис предложения:

```
VALUES ( выражение [, ...] ) [, ...]
```

Для каждого списка выражений в скобках создаётся строка таблицы. Все списки должны иметь одинаковое число элементов (т. е. число столбцов в таблице) и соответствующие элементы во всех списках должны иметь совместимые типы данных. Фактический тип данных столбцов результата определяется по тем же правилам, что и для `UNION` (см. Раздел 10.5).

Как пример:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

вернёт таблицу из двух столбцов и трёх строк. Это равносильно такому запросу:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

По умолчанию Postgres Pro назначает столбцам таблицы `VALUES` имена `column1`, `column2` и т. д. Имена столбцов не определены в стандарте SQL и в другой СУБД они могут быть другими, поэтому обычно лучше переопределить имена списком псевдонимов, например так:

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
   3 | three
(3 rows)
```

Синтаксически список `VALUES` с набором выражений равнозначен:

```
SELECT список_выборки FROM табличное_выражение
```

и допускается везде, где допустим `SELECT`. Например, вы можете использовать его в составе `UNION` или добавить к нему *определение_сортировки* (`ORDER BY`, `LIMIT` и/или `OFFSET`). `VALUES` чаще всего используется как источник данных для команды `INSERT`, а также как подзапрос.

За дополнительными сведениями обратитесь к справке VALUES.

7.8. Запросы `WITH` (Общие табличные выражения)

`WITH` предоставляет способ записывать дополнительные операторы для применения в больших запросах. Эти операторы, которые также называют общими табличными выражениями (Common Table Expressions, CTE), можно представить как определения временных таблиц, существующих только для одного запроса. Дополнительным оператором в предложении `WITH` может быть `SELECT`, `INSERT`, `UPDATE`, `DELETE` или `MERGE`, а само предложение `WITH` присоединяется к основному оператору, которым также может быть `SELECT`, `INSERT`, `UPDATE`, `DELETE` или `MERGE`.

7.8.1. `SELECT` в `WITH`

Основное предназначение `SELECT` в предложении `WITH` заключается в разбиении сложных запросов на простые части. Например, запрос:

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
```

```
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

выводит итоги по продажам только для передовых регионов. Предложение `WITH` определяет два дополнительных оператора `regional_sales` и `top_regions` так, что результат `regional_sales` используется в `top_regions`, а результат `top_regions` используется в основном запросе `SELECT`. Этот пример можно было бы переписать без `WITH`, но тогда нам понадобятся два уровня вложенных подзапросов `SELECT`. Показанным выше способом это можно сделать немного проще.

7.8.2. Рекурсивные запросы

Необязательное указание `RECURSIVE` превращает `WITH` из просто удобной синтаксической конструкции в средство реализации того, что невозможно в стандартном SQL. Используя `RECURSIVE`, запрос `WITH` может обращаться к собственному результату. Очень простой пример, суммирующий числа от 1 до 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

В общем виде рекурсивный запрос `WITH` всегда записывается как *не рекурсивная часть*, потом `UNION` (или `UNION ALL`), а затем *рекурсивная часть*, где только в рекурсивной части можно обратиться к результату запроса. Такой запрос выполняется следующим образом:

Вычисление рекурсивного запроса

1. Вычисляется не рекурсивная часть. Для `UNION` (но не `UNION ALL`) отбрасываются дублирующиеся строки. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную *рабочую таблицу*.
2. Пока рабочая таблица не пуста, повторяются следующие действия:
    a. Вычисляется рекурсивная часть так, что рекурсивная ссылка на сам запрос обращается к текущему содержимому рабочей таблицы. Для `UNION` (но не `UNION ALL`) отбрасываются дублирующиеся строки и строки, дублирующие ранее полученные. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную *промежуточную таблицу*.
    b. Содержимое рабочей таблицы заменяется содержимым промежуточной таблицы, а затем промежуточная таблица очищается.

Примечание

Хотя указание `RECURSIVE` позволяет определять рекурсивные запросы, внутри такие запросы обрабатываются итерационно.

В показанном выше примере в рабочей таблице на каждом этапе содержится всего одна строка и в ней последовательно накапливаются значения от 1 до 100. На сотом шаге, благодаря условию `WHERE`, не возвращается ничего, так что вычисление запроса завершается.

Рекурсивные запросы обычно применяются для работы с иерархическими или древовидными структурами данных. В качестве полезного примера можно привести запрос, находящий все непосредственные и косвенные составные части продукта, используя только таблицу с прямыми связями:

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity * pr.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

7.8.2.1. Порядок поиска

Когда при выполнении рекурсивного запроса производится обход дерева, результаты можно получать в разном порядке: «сначала в глубину» или «сначала в ширину». Для этого можно в дополнение к другим столбцам вычислить упорядочивающий столбец и использовать его для сортировки результатов. Обратите внимание, что такой столбец не определяет порядок обхода строк запросом — этот порядок, как всегда, зависит от реализации SQL. Упорядочивающий столбец лишь позволяет удобным образом упорядочить полученные результаты.

Чтобы отсортировать результаты в порядке «сначала в глубину», для каждой строки результата вычисляется массив уже просмотренных строк. Например, рассмотрим следующий запрос, который выполняет поиск в таблице `tree` по полю `link`:

```
WITH RECURSIVE search_tree(id, link, data) AS (
    SELECT t.id, t.link, t.data
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree;
```

Чтобы добавить информацию для упорядочивания «сначала в глубину», вы можете написать так:

```
WITH RECURSIVE search_tree(id, link, data, path) AS (
    SELECT t.id, t.link, t.data, ARRAY[t.id]
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data, path || t.id
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY path;
```

В общем случае, когда для выявления строки нужно использовать несколько полей, следует использовать массив строк. Например, если нужно отследить поля `f1` и `f2`:

```
WITH RECURSIVE search_tree(id, link, data, path) AS (
    SELECT t.id, t.link, t.data, ARRAY[ROW(t.f1, t.f2)]
    FROM tree t
```

```
  UNION ALL
    SELECT t.id, t.link, t.data, path || ROW(t.f1, t.f2)
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY path;
```

Подсказка

`ROW()` можно опустить, когда нужно отслеживать только одно поле (как обычно и бывает). При этом будет использоваться не массив данных составного типа, а простой массив, что более эффективно.

Чтобы получить результаты, отсортированные «сначала в ширину», можно добавить столбец, отслеживающий глубину поиска, например:

```
WITH RECURSIVE search_tree(id, link, data, depth) AS (
    SELECT t.id, t.link, t.data, 0
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data, depth + 1
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY depth;
```

Для обеспечения стабильности сортировки добавьте столбцы данных в качестве столбцов вторичной сортировки.

Подсказка

Этот алгоритм рекурсивного вычисления запроса выдаёт в результате узлы, упорядоченные «сначала в ширину». И всё же это зависит от реализации, поэтому полагаться на это поведение не следует. Порядок строк внутри каждого уровня, конечно, не определён, поэтому в любом случае может потребоваться явное упорядочивание.

Для определения столбца, который будет вычисляться для упорядочивания «сначала в глубину» или «сначала в ширину», есть встроенный синтаксис. Например:

```
WITH RECURSIVE search_tree(id, link, data) AS (
    SELECT t.id, t.link, t.data
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
) SEARCH DEPTH FIRST BY id SET ordercol
SELECT * FROM search_tree ORDER BY ordercol;

WITH RECURSIVE search_tree(id, link, data) AS (
    SELECT t.id, t.link, t.data
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
) SEARCH BREADTH FIRST BY id SET ordercol
```

```
SELECT * FROM search_tree ORDER BY ordercol;
```

Внутри этот синтаксис преобразуется в формы, подобные вышеприведённым формам, составленным вручную. В предложении `SEARCH` указывается, какой тип поиска необходим — «сначала в глубину» или «сначала в ширину», список отслеживаемых для сортировки столбцов и имя столбца, который будет содержать данные, используемые для упорядочивания. Этот столбец будет неявно добавлен в результирующие строки в CTE.

7.8.2.2. Выявление циклов

Работая с рекурсивными запросами, важно обеспечить, чтобы рекурсивная часть запроса в конце концов не выдала никаких кортежей (строк), в противном случае цикл будет бесконечным. Иногда для этого достаточно применять `UNION` вместо `UNION ALL`, так как при этом будут отбрасываться строки, которые уже есть в результате. Однако часто в цикле выдаются строки, не совпадающие полностью с предыдущими: в таких случаях может иметь смысл проверить одно или несколько полей, чтобы определить, не была ли текущая точка достигнута раньше. Стандартный способ решения подобных задач — вычислить массив с уже обработанными значениями. Например, снова рассмотрите следующий запрос, просматривающий таблицу `graph` по полю `link`:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 0
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

Этот запрос зациклится, если связи `link` содержат циклы. Так как нам нужно получать в результате «depth», одно лишь изменение `UNION ALL` на `UNION` не позволит избежать зацикливания. Вместо этого мы должны как-то определить, что уже достигали текущей строки, пройдя некоторый путь. Для этого, добавив два столбца `is_cycle` и `path`, мы получаем запрос, защищённый от зацикливания:

```
WITH RECURSIVE search_graph(id, link, data, depth, is_cycle, path) AS (
    SELECT g.id, g.link, g.data, 0,
      false,
      ARRAY[g.id]
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      g.id = ANY(path),
      path || g.id
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT is_cycle
)
SELECT * FROM search_graph;
```

Помимо предотвращения циклов, значения массива часто бывают полезны сами по себе для представления «пути», приведшего к определённой строке.

В общем случае, когда для выявления цикла нужно проверять несколько полей, следует использовать массив строк. Например, если нужно сравнить поля `f1` и `f2`:

```
WITH RECURSIVE search_graph(id, link, data, depth, is_cycle, path) AS (
    SELECT g.id, g.link, g.data, 0,
      false,
      ARRAY[ROW(g.f1, g.f2)]
    FROM graph g
  UNION ALL
```

```
    SELECT g.id, g.link, g.data, sg.depth + 1,
      ROW(g.f1, g.f2) = ANY(path),
      path || ROW(g.f1, g.f2)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT is_cycle
)
SELECT * FROM search_graph;
```

Подсказка

Часто для выявления цикла достаточно одного поля, и тогда `ROW()` можно опустить. При этом будет использоваться не массив данных составного типа, а простой массив, что более эффективно.

Для упрощения выявления циклов есть встроенный синтаксис. Запрос выше можно записать так:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
) CYCLE id SET is_cycle USING path
SELECT * FROM search_graph;
```

и внутри он будет приведён к вышеуказанной форме. В предложении `CYCLE` сначала указывается список столбцов, отслеживаемых для выявления цикла, затем имя столбца, показывающего признак выявления цикла, и, наконец, имя ещё одного столбца, в котором будет отслеживаться путь. Столбцы, указывающие цикл и путь, будут неявно добавлены в результирующие строки в CTE.

Подсказка

Столбец с указанием пути цикла вычисляется так же, как и столбец для упорядочивания результатов «сначала в глубину», что показано в предыдущем подразделе. Запрос может содержать одновременно `SEARCH` и `CYCLE`, но в этом случае для упорядочивания «сначала в глубину» будут производиться лишние вычисления, поэтому эффективнее использовать только `CYCLE` и сортировать результаты по столбцу пути. Однако использование в запросе и `SEARCH`, и `CYCLE` оправдано, когда нужен порядок «сначала в ширину».

Для тестирования запросов, которые могут зацикливаться, есть хороший приём — добавить `LIMIT` в родительский запрос. Например, следующий запрос зациклится, если не добавить предложение `LIMIT`:

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

Но в данном случае этого не происходит, так как в Postgres Pro запрос `WITH` выдаёт столько строк, сколько фактически принимает родительский запрос. В производственной среде использовать этот приём не рекомендуется, так как другие системы могут вести себя по-другому. Кроме того, это не будет работать, если внешний запрос сортирует результаты рекурсивного запроса или соединяет их с другой таблицей, так как в подобных случаях внешний запрос обычно всё равно выбирает результат запроса `WITH` полностью.

7.8.3. Материализация общих табличных выражений

Запросы `WITH` имеют полезное свойство — обычно они вычисляются только раз для всего родительского запроса, даже если этот запрос или соседние запросы `WITH` обращаются к ним неоднократно. Таким образом, сложные вычисления, результаты которых нужны в нескольких местах, можно выносить в запросы `WITH` в целях оптимизации. Кроме того, такие запросы позволяют избежать нежелательных вычислений функций с побочными эффектами. Однако есть и обратная сторона — оптимизатор не может распространить ограничения родительского запроса на неоднократно задействуемый запрос `WITH`, так как это может повлиять на использование результата `WITH` во всех местах, тогда как должно повлиять только в одном. Многократно задействуемый запрос `WITH` будет выполняться буквально и возвращать все строки, включая те, что потом может отбросить родительский запрос. (Но как было сказано выше, вычисление может остановиться раньше, если в ссылке на этот запрос затребуется только ограниченное число строк.)

Однако если запрос `WITH` является нерекурсивным и свободным от побочных эффектов (то есть это `SELECT`, не вызывающий изменчивых функций), он может быть свёрнут в родительский запрос, что позволит оптимизировать совместно два уровня запросов. По умолчанию это происходит, только если запрос `WITH` задействуется в родительском запросе всего в одном месте, а не в нескольких. Это поведение можно переопределить, добавив указание `MATERIALIZED`, чтобы выделить вычисление запроса `WITH`, или указание `NOT MATERIALIZED`, чтобы принудительно свернуть его в родительский запрос. В последнем случае возникает риск многократного вычисления запроса `WITH`, но в итоге это может быть выгодно, если в каждом случае использования `WITH` из всего результата запроса остаётся только небольшая часть.

Простой пример для демонстрации этих правил:

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w WHERE key = 123;
```

Этот запрос `WITH` будет свёрнут в родительский и будет выполнен с тем же планом, что и запрос:

```
SELECT * FROM big_table WHERE key = 123;
```

В частности, если в таблице создан индекс по столбцу `key`, он может использоваться для получения строк с `key = 123`. В другом случае:

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

запрос `WITH` будет материализован, то есть создастся временная копия таблицы `big_table`, которая будет соединена с собой же, без использования какого-либо индекса. Этот запрос будет выполняться гораздо эффективнее в таком виде:

```
WITH w AS NOT MATERIALIZED (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

В этом случае ограничения родительского запроса могут применяться непосредственно при сканировании `big_table`.

Пример, в котором вариант `NOT MATERIALIZED` может быть нежелательным:

```
WITH w AS (
    SELECT key, very_expensive_function(val) as f FROM some_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.f = w2.f;
```

В данном случае благодаря материализации запроса `WITH` ресурсоёмкая функция `very_expensive_function` вычисляется только один раз для строки таблицы, а не дважды.

Примеры выше показывают только предложение `WITH` с `SELECT`, но таким же образом его можно использовать с командами `INSERT`, `UPDATE`, `DELETE` и `MERGE`. В каждом случае он по сути создаёт временную таблицу, к которой можно обратиться в основной команде.

7.8.4. Изменение данных в `WITH`

В предложении `WITH` можно использовать операторы, изменяющие данные (`INSERT`, `UPDATE`, `DELETE` или `MERGE`). Это позволяет выполнять в одном запросе сразу несколько разных операций. Например:

```
WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

Этот запрос фактически перемещает строки из `products` в `products_log`. Оператор `DELETE` в `WITH` удаляет указанные строки из `products` и возвращает их содержимое в предложении `RETURNING`; а затем главный запрос читает это содержимое и вставляет в таблицу `products_log`.

Следует заметить, что предложение `WITH` в данном случае присоединяется к оператору `INSERT`, а не к `SELECT`, вложенному в `INSERT`. Это необходимо, так как `WITH` может содержать операторы, изменяющие данные, только на верхнем уровне запроса. Однако при этом применяются обычные правила видимости `WITH`, так что к результату `WITH` можно обратиться и из вложенного оператора `SELECT`.

Операторы, изменяющие данные, в `WITH` обычно дополняются предложением `RETURNING` (см. Раздел 6.4), как показано в этом примере. Важно понимать, что временная таблица, которую можно будет использовать в остальном запросе, создаётся из результата `RETURNING`, а *не* целевой таблицы оператора. Если оператор, изменяющий данные, в `WITH` не дополнен предложением `RETURNING`, временная таблица не создаётся и обращаться к ней в остальном запросе нельзя. Однако такой запрос всё равно будет выполнен. Например, допустим следующий не очень практичный запрос:

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

Он удалит все строки из таблиц `foo` и `bar`. При этом число задействованных строк, которое получит клиент, будет подсчитываться только по строкам, удалённым из `bar`.

Рекурсивные ссылки в операторах, изменяющих данные, не допускаются. В некоторых случаях это ограничение можно обойти, обратившись к конечному результату рекурсивного `WITH`, например так:

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

Этот запрос удаляет все непосредственные и косвенные составные части продукта.

Операторы, изменяющие данные в `WITH`, выполняются только один раз и всегда полностью, вне зависимости от того, принимает ли их результат основной запрос. Заметьте, что это отличается от поведения `SELECT` в `WITH`: как говорилось в предыдущем разделе, `SELECT` выполняется только до тех пор, пока его результаты востребованы основным запросом.

Вложенные операторы в `WITH` выполняются одновременно друг с другом и с основным запросом. Таким образом, порядок, в котором операторы в `WITH` будут фактически изменять данные, непредсказуем. Все эти операторы выполняются с одним *снимком данных* (см. Главу 13), так что они не могут «видеть», как каждый из них меняет целевые таблицы. Это уменьшает эффект непредсказуемости фактического порядка изменения строк и означает, что `RETURNING` — единственный вариант передачи изменений от вложенных операторов `WITH` основному запросу. Например, в данном случае:

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

внешний оператор `SELECT` выдаст цены, которые были до действия `UPDATE`, тогда как в запросе

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

внешний `SELECT` выдаст изменённые данные.

Неоднократное изменение одной и той же строки в рамках одного оператора не поддерживается. Иметь место будет только одно из нескольких изменений и надёжно определить, какое именно, часто довольно сложно (а иногда и вовсе невозможно). Это так же касается случая, когда строка удаляется и изменяется в том же операторе: в результате может быть выполнено только обновление. Поэтому в общем случае следует избегать подобного наложения операций. В частности, избегайте подзапросов `WITH`, которые могут повлиять на строки, изменяемые основным оператором или операторами, вложенные в него. Результат действия таких запросов будет непредсказуемым.

В настоящее время, для оператора, изменяющего данные в `WITH`, в качестве целевой нельзя использовать таблицу, для которой определено условное правило или правило `ALSO` или `INSTEAD`, если оно состоит из нескольких операторов.

Глава 8. Типы данных

Postgres Pro предоставляет пользователям богатый ассортимент встроенных типов данных. Кроме того, пользователи могут создавать свои типы в Postgres Pro, используя команду CREATE TYPE.

Таблица 8.1 содержит все встроенные типы данных общего пользования. Многие из альтернативных имён, приведённых в столбце «Псевдонимы», используются внутри Postgres Pro по историческим причинам. В этот список не включены некоторые устаревшие типы и типы для внутреннего применения.

Таблица 8.1. Типы данных

Имя	**Псевдонимы**	**Описание**
`bigint`	`int8`	знаковое целое из 8 байт
`bigserial`	`serial8`	восьмибайтное целое с автоувеличением
`bit [ (n) ]`		битовая строка фиксированной длины
`bit varying [ (n) ]`	`varbit [ (n) ]`	битовая строка переменной длины
`boolean`	`bool`	логическое значение (true/false)
`box`		прямоугольник в плоскости
`bytea`		двоичные данные («массив байт»)
`character [ (n) ]`	`char [ (n) ]`	символьная строка фиксированной длины
`character varying [ (n) ]`	`varchar [ (n) ]`	символьная строка переменной длины
`cidr`		сетевой адрес IPv4 или IPv6
`circle`		круг в плоскости
`date`		календарная дата (год, месяц, день)
`double precision`	`float`, `float8`	число двойной точности с плавающей точкой (8 байт)
`inet`		адрес узла IPv4 или IPv6
`integer`	`int`, `int4`	знаковое четырёхбайтное целое
`interval [ поля ] [ (p) ]`		интервал времени
`json`		текстовые данные JSON
`jsonb`		двоичные данные JSON, разобранные
`line`		прямая в плоскости
`lseg`		отрезок в плоскости
`macaddr`		MAC-адрес
`macaddr8`		адрес MAC (Media Access Control) (в формате EUI-64)
`money`		денежная сумма
`numeric [ (p, s) ]`	`decimal [ (p, s) ]`	вещественное число заданной точности
`path`		геометрический путь в плоскости
`pg_lsn`		последовательный номер в журнале Postgres Pro
`pg_snapshot`		снимок идентификатора транзакций

Имя	**Псевдонимы**	**Описание**
`point`		геометрическая точка в плоскости
`polygon`		замкнутый геометрический путь в плоскости
`real`	`float4`	число одинарной точности с плавающей точкой (4 байта)
`smallint`	`int2`	знаковое двухбайтное целое
`smallserial`	`serial2`	двухбайтное целое с автоувеличением
`serial`	`serial4`	четырёхбайтное целое с автоувеличением
`text`		символьная строка переменной длины
`time [ (`*p*`) ] [ without time zone ]`		время суток (без часового пояса)
`time [ (`*p*`) ] with time zone`	`timetz`	время суток с учётом часового пояса
`timestamp [ (`*p*`) ] [ without time zone ]`		дата и время (без часового пояса)
`timestamp [ (`*p*`) ] with time zone`	`timestamptz`	дата и время с учётом часового пояса
`tsquery`		запрос текстового поиска
`tsvector`		документ для текстового поиска
`txid_snapshot`		снимок идентификаторов транзакций для пользовательского уровня (устаревший тип; см. `pg_snapshot`)
`uuid`		универсальный уникальный идентификатор
`xml`		XML-данные

Совместимость

В стандарте SQL описаны следующие типы (или их имена): `bigint`, `bit`, `bit varying`, `boolean`, `char`, `character varying`, `character`, `varchar`, `date`, `double precision`, `integer`, `interval`, `numeric`, `decimal`, `real`, `smallint`, `time` (с часовым поясом и без), `timestamp` (с часовым поясом и без), `xml`.

Каждый тип данных имеет внутреннее представление, скрытое функциями ввода и вывода. При этом многие встроенные типы стандартны и имеют очевидные внешние форматы. Однако есть типы, уникальные для Postgres Pro, например геометрические пути, и есть типы, которые могут иметь разные форматы, например, дата и время. Некоторые функции ввода и вывода не являются в точности обратными друг к другу, то есть результат функции вывода может не совпадать со входным значением из-за потери точности.

8.1. Числовые типы

Числовые типы включают двух-, четырёх- и восьмибайтные целые, четырёх- и восьмибайтные числа с плавающей точкой, а также десятичные числа с задаваемой точностью. Все эти типы перечислены в Таблице 8.2.

Таблица 8.2. Числовые типы

Имя	Размер	Описание	Диапазон
`smallint`	2 байта	целое в небольшом диапазоне	-32768 .. +32767
`integer`	4 байта	типичный выбор для целых чисел	-2147483648 .. +2147483647
`bigint`	8 байт	целое в большом диапазоне	-9223372036854775808 .. 9223372036854775807
`decimal`	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
`numeric`	переменный	вещественное число с указанной точностью	до 131072 цифр до десятичной точки и до 16383 — после
`real`	4 байта	вещественное число с переменной точностью	точность в пределах 6 десятичных цифр
`double precision`	8 байт	вещественное число с переменной точностью	точность в пределах 15 десятичных цифр
`smallserial`	2 байта	небольшое целое с автоувеличением	1 .. 32767
`serial`	4 байта	целое с автоувеличением	1 .. 2147483647
`bigserial`	8 байт	большое целое с автоувеличением	1 .. 9223372036854775807

Синтаксис констант числовых типов описан в Подразделе 4.1.2. Для этих типов определён полный набор соответствующих арифметических операторов и функций. За дополнительными сведениями обратитесь к Главе 9. Подробнее эти типы описаны в следующих разделах.

8.1.1. Целочисленные типы

Типы `smallint`, `integer` и `bigint` хранят целые числа, то есть числа без дробной части, имеющие разные допустимые диапазоны. Попытка сохранить значение, выходящее за рамки диапазона, приведёт к ошибке.

Чаще всего используется тип `integer`, как наиболее сбалансированный выбор ширины диапазона, размера и быстродействия. Тип `smallint` обычно применяется, только когда крайне важно уменьшить размер данных на диске. Тип `bigint` предназначен для тех случаев, когда числа не умещаются в диапазон типа `integer`.

В SQL определены только типы `integer` (или `int`), `smallint` и `bigint`. Имена типов `int2`, `int4` и `int8` выходят за рамки стандарта, хотя могут работать и в некоторых других СУБД.

8.1.2. Числа с произвольной точностью

Тип `numeric` позволяет хранить числа с очень большим количеством цифр. Он особенно рекомендуется для хранения денежных сумм и других величин, где важна точность. Вычисления с типом `numeric` дают точные результаты, где это возможно, например, при сложении, вычитании и умножении. Однако операции со значениями `numeric` выполняются гораздо медленнее, чем с целыми числами или с типами с плавающей точкой, описанными в следующем разделе.

Ниже мы используем следующие термины: *масштаб* значения `numeric` определяет количество десятичных цифр в дробной части, справа от десятичной точки, а *точность* — общее количество значимых цифр в числе, т. е. количество цифр по обе стороны десятичной точки. Например, число

23.5141 имеет точность 6 и масштаб 4. Целочисленные значения можно считать числами с масштабом 0.

Для столбца типа `numeric` можно настроить и максимальную точность, и максимальный масштаб. Столбец типа `numeric` объявляется следующим образом:

```
NUMERIC(точность, масштаб)
```

Точность должна быть положительной, а масштаб может быть положительным или отрицательным (см. ниже). Альтернативный вариант

```
NUMERIC(точность)
```

устанавливает масштаб 0. Форма:

```
NUMERIC
```

без указания точности и масштаба создаёт столбец типа «неограниченное число», в котором можно сохранять числовые значения любой длины до предела, обусловленного реализацией. В столбце этого типа входные значения не будут приводиться к какому-либо масштабу, тогда как в столбцах `numeric` с явно заданным масштабом значения подгоняются под этот масштаб. (Стандарт SQL утверждает, что по умолчанию должен устанавливаться масштаб 0, т. е. значения должны приводиться к целым числам. Однако мы считаем это не очень полезным. Если для вас важна переносимость, всегда указывайте точность и масштаб явно.)

Примечание

Максимальная точность, которую можно указать в объявлении типа `numeric`, равна 1000. Для столбцов типа «неограниченный `numeric`» существуют пределы, описанные в Таблице 8.2.

Если масштаб значения, которое нужно сохранить, превышает объявленный масштаб столбца, система округлит его до заданного количества цифр после точки. Если же после этого количество цифр слева в сумме с масштабом превысит объявленную точность, произойдёт ошибка. Например, столбец, объявленный как

```
NUMERIC(3, 1)
```

, будет содержать значения, округлённые до одного десятичного знака, от -99.9 до 99.9 включительно.

Начиная с Postgres Pro 15, столбец типа `numeric` можно объявить с отрицательным масштабом. В этом случае значения будут округляться до позиции слева от десятичной точки. При этом точность так же задаёт максимальное число цифр, без учёта округления. Таким образом, в столбце, объявленном как

```
NUMERIC(2, -3)
```

, число будет округляться до ближайшей тысячи, и в нём можно будет хранить значения от -99000 до 99000 включительно. Также вполне возможно задать масштаб, превышающий объявленную точность. Такой столбец сможет содержать только дробные значения, в которых количество нулей сразу после точки должно равняться разности масштаба и точности. Например, столбец, объявленный как

```
NUMERIC(3, 5)
```

, будет округлять значения до пяти знаков после точки и сможет хранить значения от -0.00999 до 0.00999 включительно.

Примечание

Postgres Pro позволяет использовать в объявлении типа `numeric` любое значение масштаба в диапазоне от -1000 до 1000. Однако стандарт SQL требует, чтобы масштаб находился в диа-

> пазоне от 0 до значения *точности*. Масштабы вне этого диапазона могут не поддерживаться другими системами баз данных.

Числовые значения физически хранятся без каких-либо дополняющих нулей слева или справа. Таким образом, объявляемые точность и масштаб столбца определяют максимальный, а не фиксированный размер хранения. (В этом смысле тип `numeric` больше похож на тип `varchar(n)`, чем на `char(n)`.) Действительный размер хранения такого значения складывается из двух байт для каждой группы из четырёх цифр и дополнительных трёх-восьми байт.

В дополнение к обычным числовым значениям тип `numeric` принимает следующие специальные значения:

```
Infinity
-Infinity
NaN
```

Они представляют особые значения, описанные в IEEE 754, соответственно «бесконечность», «минус бесконечность» и «не число». Записывая эти значения в виде констант в команде SQL, их нужно заключать в апострофы, например так: `UPDATE table SET x = '-Infinity'`. Регистр символов в этих строках не важен. В качестве альтернативы значения бесконечности могут быть записаны как `inf` и `-inf`.

Значения бесконечности соответствуют ожиданиям с точки зрения математики. Например, `Infinity` плюс любое конечное значение равно `Infinity`, как и `Infinity` плюс `Infinity`; но `Infinity` минус `Infinity` даёт `NaN` (не число), потому что в результате получается неопределённость. Обратите внимание, что бесконечность может быть сохранена только в столбце типа «неограниченный `numeric`», потому что она теоретически превышает любой конечный предел точности.

Значение `NaN` (не число) используется для представления неопределённых результатов вычислений. Вообще, любая операция с `NaN` выдаёт в результате тоже `NaN`. Единственное исключение составляют операции, которые дают один и тот же результат для любого числового значения, конечного или бесконечного — этот результат будет выдаваться и для `NaN`. (Пример этого принципа: результатом возведения `NaN` в нулевую степень будет единица.)

> ### Примечание
>
> В большинстве реализаций «не число» (`NaN`) считается не равным любому другому значению (в том числе и самому `NaN`). Чтобы значения `numeric` можно было сортировать и использовать в древовидных индексах, Postgres Pro считает, что значения `NaN` равны друг другу и при этом больше любых числовых значений (не `NaN`).

Типы `decimal` и `numeric` равнозначны. Оба эти типа описаны в стандарте SQL.

При округлении значений тип `numeric` выдаёт число, большее по модулю, тогда как (на большинстве платформ) типы `real` и `double precision` выдают ближайшее чётное число. Например:

```
SELECT x,
  round(x::numeric) AS num_round,
  round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
```

```
  0.5 |          1 |         0
  1.5 |          2 |         2
  2.5 |          3 |         2
  3.5 |          4 |         4
(8 rows)
```

8.1.3. Типы с плавающей точкой

Типы данных `real` и `double precision` хранят приближённые числовые значения с переменной точностью. На всех поддерживаемых в настоящее время платформах эти типы реализуют стандарт IEEE 754 для двоичной арифметики с плавающей точкой (с одинарной и двойной точностью соответственно), в той мере, в какой его поддерживают процессор, операционная система и компилятор.

Неточность здесь выражается в том, что некоторые значения, которые нельзя преобразовать во внутренний формат, сохраняются приближённо, так что полученное значение может несколько отличаться от записанного. Управление подобными ошибками и их распространение в процессе вычислений является предметом изучения целого раздела математики и компьютерной науки, и здесь не рассматривается. Мы отметим только следующее:

- Если вам нужна точность при хранении и вычислениях (например, для денежных сумм), используйте вместо этого тип `numeric`.
- Если вы хотите выполнять с этими типами сложные вычисления, имеющие большую важность, тщательно изучите реализацию операций в вашей среде и особенно поведение в крайних случаях (бесконечность, антипереполнение).
- Проверка равенства двух чисел с плавающей точкой может не всегда давать ожидаемый результат.

На всех поддерживаемых сейчас платформах тип `real` может сохранить значения примерно от 1E-37 до 1E+37 с точностью не меньше 6 десятичных цифр. Тип `double precision` предлагает значения в диапазоне приблизительно от 1E-307 до 1E+308 и с точностью не меньше 15 цифр. Попытка сохранить слишком большие или слишком маленькие значения приведёт к ошибке. Если точность вводимого числа слишком велика, оно будет округлено. При попытке сохранить число, близкое к 0, но непредставимое как отличное от 0, произойдёт ошибка антипереполнения.

По умолчанию числа с плавающей точкой выводятся в текстовом виде в кратчайшем точном десятичном представлении; выводимое десятичное значение оказывается более близким к изначальному двоичному числу, чем любое другое значение, представимое с той же двоичной точностью. (Однако выводимое значение в текущей реализации никогда не находится *точно* посередине между двумя представимыми двоичными значениями, во избежание распространённой ошибки с функциями ввода, не учитывающими корректно правило округления до ближайшего чётного.) Выводимое значение может занимать не больше 17 значащих десятичных цифр для типа `float8` и не больше 9 цифр для типа `float4`.

> **Примечание**
>
> Преобразование в кратчайший точный вид производится гораздо быстрее, чем в традиционное представление с округлением.

Для совместимости с результатами, выдаваемыми старыми версиями Postgres Pro, и уменьшения точности выводимых чисел, когда это требуется, в параметре extra_float_digits можно выбрать также вариант округлённого десятичного вывода. Со значением 0 восстанавливается действовавшее ранее по умолчанию округление числа до 6 (для типа `float4`) или 15 (для `float8`) значащих десятичных цифр. При отрицательных значениях число значащих цифр уменьшается дополнительно; например, при -2 результат будет округлён до 4 или 13 цифр, соответственно.

При любом значении extra_float_digits, большем 0, выбирается кратчайшее точное представление.

Примечание

Приложения, которым были нужны точные числовые значения, раньше задавали для параметра extra_float_digits значение 3, чтобы получить их. Они могут продолжать использовать это значение для максимальной совместимости с разными версиями.

В дополнение к обычным числовым значениям типы с плавающей точкой принимают следующие специальные значения:

```
Infinity
-Infinity
NaN
```

Они представляют особые значения, описанные в IEEE 754, соответственно «бесконечность», «минус бесконечность» и «не число». Записывая эти значения в виде констант в команде SQL, их нужно заключать в апострофы, например так: `UPDATE table SET x = '-Infinity'`. Регистр символов в этих строках не важен. В качестве альтернативы значения бесконечности могут быть записаны как `inf` и `-inf`.

Примечание

Согласно IEEE 754, `NaN` не должно считаться равным любому другому значению с плавающей точкой (в том числе и самому `NaN`). Чтобы значения с плавающей точкой можно было сортировать и использовать в древовидных индексах, Postgres Pro считает, что значения `NaN` равны друг другу, и при этом больше любых числовых значений (не `NaN`).

Postgres Pro также поддерживает форматы `float` и `float(p)`, оговорённые в стандарте SQL, для указания неточных числовых типов. Здесь *p* определяет минимально допустимую точность в *двоичных* цифрах. Postgres Pro воспринимает запись от `float(1)` до `float(24)` как выбор типа `real`, а запись от `float(25)` до `float(53)` как выбор типа `double precision`. Значения *p* вне допустимого диапазона вызывают ошибку. Если `float` указывается без точности, подразумевается тип `double precision`.

8.1.4. Последовательные типы

Примечание

В этом разделе описывается специфичный для Postgres Pro способ создания столбца с автоувеличением. Другой способ, соответствующий стандарту SQL, заключается в использовании столбцов идентификации и рассматривается в описании Раздел 5.3.

Типы данных `smallserial`, `serial` и `bigserial` не являются настоящими типами, а представляют собой просто удобное средство для создания столбцов с уникальными идентификаторами (подобное свойству `AUTO_INCREMENT` в некоторых СУБД). В текущей реализации запись:

```
CREATE TABLE имя_таблицы (
    имя_столбца SERIAL
);
```

равнозначна следующим командам:

```
CREATE SEQUENCE имя_таблицы_имя_столбца_seq AS integer;
CREATE TABLE имя_таблицы (
    имя_столбца integer NOT NULL DEFAULT nextval('имя_таблицы_имя_столбца_seq')
);
ALTER SEQUENCE имя_таблицы_имя_столбца_seq OWNED BY имя_таблицы.имя_столбца;
```

То есть при определении такого типа создаётся целочисленный столбец со значением по умолчанию, извлекаемым из генератора последовательности. Чтобы в столбец нельзя было вставить NULL, в его определение добавляется ограничение `NOT NULL`. (Во многих случаях также имеет смысл добавить для этого столбца ограничения `UNIQUE` или `PRIMARY KEY` для защиты от ошибочного добавления дублирующихся значений, но автоматически это не происходит.) Последняя команда определяет, что последовательность «принадлежит» столбцу, так что она будет удалена при удалении столбца или таблицы.

Примечание

Так как типы `smallserial`, `serial` и `bigserial` реализованы через последовательности, в числовом ряду значений столбца могут образовываться пропуски (или «дыры»), даже если никакие строки не удалялись. Значение, выделенное из последовательности, считается «задействованным», даже если строку с этим значением не удалось вставить в таблицу. Это может произойти, например, при откате транзакции, добавляющей данные. См. описание `nextval()` в Разделе 9.17.

Чтобы вставить в столбец `serial` следующее значение последовательности, ему нужно присвоить значение по умолчанию. Это можно сделать, либо исключив его из списка столбцов в операторе `INSERT`, либо с помощью ключевого слова `DEFAULT`.

Имена типов `serial` и `serial4` равнозначны: они создают столбцы `integer`. Так же являются синонимами имена `bigserial` и `serial8`, но они создают столбцы `bigint`. Тип `bigserial` следует использовать, если за всё время жизни таблицы планируется использовать больше чем 2^{31} значений. И наконец, синонимами являются имена типов `smallserial` и `serial2`, но они создают столбец `smallint`.

Последовательность, созданная для столбца `serial`, автоматически удаляется при удалении связанного столбца. Последовательность можно удалить и отдельно от столбца, но при этом также будет удалено определение значения по умолчанию.

8.2. Денежные типы

Тип `money` хранит денежную сумму с фиксированной дробной частью; см. Таблицу 8.3. Точность дробной части определяется на уровне базы данных параметром lc_monetary. Для диапазона, показанного в таблице, предполагается, что число содержит два знака после запятой. Входные данные могут быть записаны по-разному, в том числе в виде целых и дробных чисел, а также в виде строки в денежном формате, например `'$1,000.00'`. Выводятся эти значения обычно в денежном формате, зависящем от региональных стандартов.

Таблица 8.3. Денежные типы

Имя	Размер	Описание	Диапазон
`money`	8 байт	денежная сумма	-92233720368547758.08 .. +92233720368547758.07

Так как выводимые значения этого типа зависят от региональных стандартов, попытка загрузить данные типа `money` в базу данных с другим параметром `lc_monetary` может быть неудачной. Во избежание подобных проблем, прежде чем восстанавливать копию в новую базу данных, убедитесь в том, что параметр `lc_monetary` в этой базе данных имеет то же значение, что и в исходной.

Значения типов `numeric`, `int` и `bigint` можно привести к типу `money`. Преобразования типов `real` и `double precision` так же возможны через тип `numeric`, например:

```
SELECT '12.34'::float8::numeric::money;
```

Однако использовать числа с плавающей точкой для денежных сумм не рекомендуется из-за возможных ошибок округления.

Значение `money` можно привести к типу `numeric` без потери точности. Преобразование в другие типы может быть неточным и также должно выполняться в два этапа:

```
SELECT '52093.89'::money::numeric::float8;
```

При делении значения типа `money` на целое число выполняется отбрасывание дробной части и получается целое, ближайшее к нулю. Чтобы получить результат с округлением, выполните деление значения с плавающей точкой или приведите значение типа `money` к `numeric` до деления, а затем приведите результат к типу `money`. (Последний вариант предпочтительнее, так как исключает риск потери точности.) Когда значение `money` делится на другое значение `money`, результатом будет значение типа `double precision` (то есть просто число, не денежная величина); денежные единицы измерения при делении сокращаются.

8.3. Символьные типы

Таблица 8.4. Символьные типы

Имя	Описание
`character varying(n)`, `varchar(n)`	строка ограниченной переменной длины
`character(n)`, `char(n)`, `bpchar(n)`	строка фиксированной длины, дополненная пробелами
`bpchar`	строка неограниченной переменной длины с удалением пробелов
`text`	строка неограниченной переменной длины

В Таблице 8.4 перечислены символьные типы общего назначения, доступные в Postgres Pro.

SQL определяет два основных символьных типа: `character varying(n)` и `character(n)`, где *n* — положительное число. Оба эти типа могут хранить текстовые строки длиной до *n* символов (не байт). Попытка сохранить в столбце такого типа более длинную строку приведёт к ошибке, если только все лишние символы не являются пробелами (тогда они будут усечены до максимально допустимой длины). (Это несколько странное исключение продиктовано стандартом SQL.) Однако если явно привести значение к типу `character varying(n)` или `character(n)`, то слишком длинное значение будет усечено до *n* символов без возникновения ошибки. (Это также является требованием стандарта SQL.) Если длина сохраняемой строки оказывается меньше объявленной, значения типа `character` будут дополнятся пробелами, а тип `character varying` просто сохранит короткую строку.

Помимо этого, Postgres Pro предлагает тип `text`, в котором можно хранить строки произвольной длины. Хотя тип `text` не описан в стандарте SQL, его поддерживают и некоторые другие СУБД SQL. Тип `text` — это встроенный строковый тип данных Postgres Pro, и большинство встроенных функций, работающих со строками, объявляются принимающими или возвращающими `text`, а не `character varying`. Во многих случаях `character varying` действует так, как если бы это был домен поверх `text`.

Имя типа `varchar` является псевдонимом для `character varying`, а `char` и `bpchar` (с указанием длины) — для `character`. Псевдонимы `varchar` и `char` определены в стандарте SQL, но `bpchar` — это расширение Postgres Pro.

Если *n* задано, оно должно быть больше нуля и меньше или равно 10485760. При использовании типов `character varying` (или `varchar`) без указания длины принимаются строки любой длины. При использовании типа `bpchar` без указания длины также принимаются строки любой длины, но конечные пробелы семантически не учитываются. Записи `character` (или `char`) без указания длины соответствует `character(1)`.

Значения типа `character` физически дополняются пробелами до *n* символов и хранятся, а затем отображаются в таком виде. Однако при сравнении двух значений типа `character` дополняющие пробелы считаются незначащими и игнорируются. С правилами сортировки, где пробельные символы являются значащими, это поведение может приводить к неожиданным результатам, например `SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)` вернёт true (условие будет истинным), хотя в локали C символ пробела считается больше символа новой строки. При приведении значения `character` к другому символьному типу дополняющие пробелы отбрасываются. Заметьте, что эти пробелы *несут* смысловую нагрузку в типах `character varying` и `text` и в проверках по шаблонам, то есть в `LIKE` и регулярных выражениях.

Какие именно символы можно сохранить в этих типах данных, зависит от того, какой набор символов был выбран при создании базы данных. Однако символ с кодом 0 (иногда называемый NUL) сохранить нельзя, вне зависимости от выбранного набора символов. За подробностями обратитесь к Разделу 22.3.

Для хранения короткой строки (до 126 байт) требуется дополнительный 1 байт плюс размер самой строки, включая дополняющие пробелы для типа `character`. Для строк длиннее требуется не 1, а 4 дополнительных байта. Система может автоматически сжимать длинные строки, так что физический размер на диске может быть меньше. Очень длинные текстовые строки переносятся в отдельные таблицы, чтобы они не замедляли работу с другими столбцами. В любом случае максимально возможный размер строки составляет около 1 ГБ. (Допустимое значение *n* в объявлении типа данных меньше этого числа. Это объясняется тем, что в зависимости от кодировки каждый символ может занимать несколько байт. Если вы желаете сохранять строки без определённого предела длины, используйте типы `text` или `character varying` без указания длины, а не задавайте какое-либо большое максимальное значение.)

Подсказка

По быстродействию эти три типа практически не отличаются друг от друга, не считая большего размера хранения для типа с дополняющими пробелами и нескольких машинных операций для проверки длины при сохранении строк в столбце с ограниченной длиной. Хотя в некоторых СУБД тип `character(n)` работает быстрее других, в Postgres Pro это не так; на деле `character(n)` обычно оказывается медленнее остальных типов из-за большего размера данных и более медленной сортировки. В большинстве случаев вместо него лучше применять `text` или `character varying`.

За информацией о синтаксисе строковых констант обратитесь к Подразделу 4.1.2.1, а об имеющихся операторах и функциях вы можете узнать в Главе 9.

Пример 8.1. Использование символьных типов

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1

  a   | char_length
------+-------------
 ok   |           2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ОШИБКА:  значение не умещается в тип character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- явное усечение
SELECT b, char_length(b) FROM test2;

   b   | char_length
-------+-------------
 ok    |           2
```

```
 good  |           5
 too l |           5
```

1 Функция `char_length` рассматривается в Разделе 9.4.

В Postgres Pro есть ещё два символьных типа фиксированной длины, приведённые в Таблице 8.5. Они предназначены не для широкого применения, а для использования только во внутренних системных каталогах. Тип `name` создан для хранения идентификаторов. В настоящее время его длина составляет 64 байта (63 ASCII-символа плюс завершающий ноль), но в исходном коде `C` она задаётся константой `NAMEDATALEN`. Эта константа определяется во время компиляции (и её можно менять в особых случаях), а кроме того, максимальная длина по умолчанию может быть увеличена в следующих версиях. Тип `"char"` (обратите внимание на кавычки) отличается от `char(1)` тем, что он занимает только один байт хранилища и поэтому может хранить только один ASCII-символ. Он используется в системных каталогах для простых перечислений.

Таблица 8.5. Специальные символьные типы

Имя	**Размер**	**Описание**
`"char"`	1 байт	внутренний однобайтный тип
`name`	64 байта	внутренний тип для имён объектов

8.4. Двоичные типы данных

Для хранения двоичных данных предназначен тип `bytea`; см. Таблицу 8.6.

Таблица 8.6. Двоичные типы данных

Имя	**Размер**	**Описание**
`bytea`	1 или 4 байта плюс сама двоичная строка	двоичная строка переменной длины

Двоичные строки представляют собой последовательность октетов (байт) и имеют два отличия от текстовых строк. Во-первых, в двоичных строках можно хранить байты с кодом 0 и другими «непечатаемыми» значениями (обычно это значения вне десятичного диапазона 32..126). В текстовых строках нельзя сохранять нулевые байты, а также значения и последовательности значений, не соответствующие выбранной кодировке базы данных. Во-вторых, в операциях с двоичными строками обрабатываются байты в чистом виде, тогда как текстовые строки обрабатываются в зависимости от языковых стандартов. То есть, двоичные строки больше подходят для данных, которые программист видит как «просто байты», а символьные строки — для хранения текста.

Тип `bytea` поддерживает два формата ввода и вывода: «шестнадцатеричный» и традиционный для PostgreSQL формат «спецпоследовательностей». Входные данные принимаются в обоих форматах, а формат выходных данных зависит от параметра конфигурации bytea_output; по умолчанию выбран шестнадцатеричный. (Заметьте, что шестнадцатеричный формат был введён в PostgreSQL 9.0; в ранних версиях и некоторых программах он не будет работать.)

Стандарт SQL определяет другой тип двоичных данных, `BLOB` (`BINARY LARGE OBJECT`, большой двоичный объект). Его входной формат отличается от форматов `bytea`, но функции и операторы в основном те же.

8.4.1. Шестнадцатеричный формат `bytea`

В «шестнадцатеричном» формате двоичные данные кодируются двумя шестнадцатеричными цифрами на байт, при этом первая цифра соответствует старшим 4 битам. К полученной строке добавляется префикс `\x` (чтобы она отличалась от формата спецпоследовательности). В некоторых контекстах обратную косую черту нужно экранировать, продублировав её (см. Подраздел 4.1.2.1). Вводимые шестнадцатеричные цифры могут быть в любом регистре, а между парами цифр допускаются пробельные символы (но не внутри пары и не в начале последовательности `\x`). Этот формат совместим со множеством внешних приложений и протоколов, к тому же обычно преобразуется быстрее, поэтому предпочтительнее использовать его.

Пример:

```
SET bytea_output = 'hex';

SELECT '\xDEADBEEF'::bytea;
   bytea
------------
 \xdeadbeef
```

8.4.2. Формат спецпоследовательностей `bytea`

Формат «спецпоследовательностей» традиционно использовался в Postgres Pro для значений типа `bytea`. В нём двоичная строка представляется в виде последовательности ASCII-символов, а байты, непредставимые в виде ASCII-символов, передаются в виде спецпоследовательностей. Этот формат может быть удобен, если с точки зрения приложения представление байт в виде символов имеет смысл. Но на практике это обычно создаёт путаницу, так как двоичные и символьные строки могут выглядеть одинаково, а кроме того выбранный механизм спецпоследовательностей довольно неуклюж. Поэтому в новых приложениях этот формат обычно не стоит использовать.

Передавая значения `bytea` в формате спецпоследовательности, байты с определёнными значениями *необходимо* записывать специальным образом, хотя так *можно* записывать и все значения. В общем виде для этого значение байта нужно преобразовать в трёхзначное восьмеричное число и добавить перед ним обратную косую черту. Саму обратную косую черту (символ с десятичным кодом 92) можно записать в виде двух таких символов. В Таблице 8.7 перечислены символы, которые нужно записывать спецпоследовательностями, и приведены альтернативные варианты записи, если они возможны.

Таблица 8.7. Спецпоследовательности записи значений `bytea`

Десятичное значение байта	Описание	Спецпоследовательность ввода	Пример	Шестнадцатеричное представление
0	нулевой байт	`'\000'`	`'\000'::bytea`	`\x00`
39	апостроф	`''''` или `'\047'`	`''''::bytea`	`\x27`
92	обратная косая черта	`'\\'` или `'\134'`	`'\\'::bytea`	`\x5c`
от 0 до 31 и от 127 до 255	«непечатаемые» байты	`E'\\xxx'` (восьмеричное значение)	`'\001'::bytea`	`\x01`

Требования экранирования *непечатаемых* символов определяются языковыми стандартами. Иногда такие символы могут восприниматься и без спецпоследовательностей.

Апострофы должны дублироваться, как показано в Таблице 8.7, потому что это обязательно для любой текстовой строки в команде SQL. При общем разборе текстовой строки внешние апострофы убираются, а каждая пара внутренних сводится к одному символу. Таким образом, функция ввода `bytea` видит всего один апостроф, который она обрабатывает как обычный символ в данных. Дублировать же обратную косую черту при вводе `bytea` не требуется: этот символ считается особым и меняет поведение функции ввода, как показано в Таблице 8.7.

В некоторых контекстах обратная косая черта должна дублироваться (относительно примеров выше), так как при общем разборе строковых констант пара таких символов будет сведена к одному; см. Подраздел 4.1.2.1.

Данные `Bytea` по умолчанию выводятся в шестнадцатеричном формате (`hex`). Если поменять значение bytea_output на `escape`, «непечатаемые» байты представляются в виде соответствующих трёхзначных восьмеричных значений, которые предваряются одной обратной косой чертой. Большинство «печатаемых» байтов представляются обычными символами из клиентского набора символов, например:

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
     bytea
----------------
 abc klm *\251T
```

Байт с десятичным кодом 92 (обратная косая черта) при выводе дублируется. Это иллюстрирует Таблица 8.8.

Таблица 8.8. Спецпоследовательности выходных значений `bytea`

Десятичное значение байта	Описание	Спецпоследовательность вывода	Пример	Выводимый результат
92	обратная косая черта	`\\`	`'\134'::bytea`	`\\`
от 0 до 31 и от 127 до 255	«непечатаемые» байты	`\`*`xxx`* (значение байта)	`'\001'::bytea`	`\001`
от 32 до 126	«печатаемые» байты	представление из клиентского набора символов	`'\176'::bytea`	`~`

В зависимости от применяемой клиентской библиотеки Postgres Pro, для преобразования значений `bytea` в спецстроки и обратно могут потребоваться дополнительные действия. Например, если приложение сохраняет в строках символы перевода строк, возможно их также нужно будет представить спецпоследовательностями.

8.5. Типы даты/времени

Postgres Pro поддерживает полный набор типов даты и времени SQL, показанный в Таблице 8.9. Операции, возможные с этими типами данных, описаны в Разделе 9.9. Все даты считаются по Григорианскому календарю, даже для времени до его введения (за дополнительными сведениями обратитесь к Разделу B.6).

Таблица 8.9. Типы даты/времени

Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
`timestamp [ (`*`p`*`) ] [ without time zone ]`	8 байт	дата и время (без часового пояса)	4713 до н. э.	294276 н. э.	1 микросекунда
`timestamp [ (`*`p`*`) ] with time zone`	8 байт	дата и время (с часовым поясом)	4713 до н. э.	294276 н. э.	1 микросекунда
`date`	4 байта	дата (без времени суток)	4713 до н. э.	5874897 н. э.	1 день
`time [ (`*`p`*`) ] [ without time zone ]`	8 байт	время суток (без даты)	00:00:00	24:00:00	1 микросекунда
`time [ (`*`p`*`) ] with time zone`	12 байт	время дня (без даты), с часовым поясом	00:00:00+1559	24:00:00-1559	1 микросекунда

Имя	Размер	Описание	Наименьшее значение	Наибольшее значение	Точность
`interval [` *`поля`* `] [ (` *`p`*`) ]`	16 байт	временной интервал	-178000000 лет	178000000 лет	1 микросекунда

Примечание

Стандарт SQL требует, чтобы тип `timestamp` подразумевал `timestamp without time zone` (время без часового пояса), и Postgres Pro следует этому. Для краткости `timestamp with time zone` можно записать как `timestamptz`; это расширение Postgres Pro.

Типы `time`, `timestamp` и `interval` принимают необязательное значение точности *`p`*, определяющее, сколько знаков после запятой должно сохраняться в секундах. По умолчанию точность не ограничивается. Допустимые значения *`p`* лежат в интервале от 0 до 6.

Тип `interval` дополнительно позволяет ограничить набор сохраняемых полей следующими фразами:

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Заметьте, что если указаны и *`поля`*, и точность *`p`*, указание *`поля`* должно включать `SECOND`, так как точность применима только к секундам.

Тип `time with time zone` определён стандартом SQL, но в его определении описаны свойства сомнительной ценности. В большинстве случаев сочетание типов `date`, `time`, `timestamp without time zone` и `timestamp with time zone` удовлетворяет все потребности в функциональности дат/времени, возникающие в приложениях.

8.5.1. Ввод даты/времени

Значения даты и времени принимаются практически в любом разумном формате, включая ISO 8601, SQL-совместимый, традиционный формат POSTGRES и другие. В некоторых форматах порядок даты, месяца и года во вводимой дате неоднозначен и поэтому поддерживается явное определение формата. Для этого предназначен параметр DateStyle. Когда он имеет значение `MDY`, выбирается интерпретация месяц-день-год, значению `DMY` соответствует день-месяц-год, а `YMD` — год-месяц-день.

Postgres Pro обрабатывает вводимые значения даты/времени более гибко, чем того требует стандарт SQL. Точные правила разбора даты/времени и распознаваемые текстовые поля, в том числе названия месяцев, дней недели и часовых поясов описаны в Приложении B.

Помните, что любые вводимые значения даты и времени нужно заключать в апострофы, как текстовые строки. За дополнительной информацией обратитесь к Подразделу 4.1.2.7. SQL предусматривает следующий синтаксис:

`тип [ (p) ] 'значение'`

Здесь *p* — необязательное указание точности, определяющее число знаков после точки в секундах. Точность может быть определена для типов `time`, `timestamp` и `interval` в интервале от 0 до 6. Если в определении константы точность не указана, она считается равной точности значения в строке (но не больше 6 цифр).

8.5.1.1. Даты

В Таблице 8.10 приведены некоторые допустимые значения типа `date`.

Таблица 8.10. Вводимые даты

Пример	Описание
1999-01-08	ISO 8601; 8 января в любом режиме (рекомендуемый формат)
January 8, 1999	воспринимается однозначно в любом режиме `datestyle`
1/8/1999	8 января в режиме `MDY` и 1 августа в режиме `DMY`
1/18/1999	18 января в режиме `MDY`; недопустимая дата в других режимах
01/02/03	2 января 2003 г. в режиме `MDY`; 1 февраля 2003 г. в режиме `DMY` и 3 февраля 2001 г. в режиме `YMD`
1999-Jan-08	8 января в любом режиме
Jan-08-1999	8 января в любом режиме
08-Jan-1999	8 января в любом режиме
99-Jan-08	8 января в режиме `YMD`; ошибка в других режимах
08-Jan-99	8 января; ошибка в режиме `YMD`
Jan-08-99	8 января; ошибка в режиме `YMD`
19990108	ISO 8601; 8 января 1999 в любом режиме
990108	ISO 8601; 8 января 1999 в любом режиме
1999.008	год и день года
J2451187	юлианский день
January 8, 99 BC	99 до н. э.

8.5.1.2. Время

Для хранения времени суток без даты предназначены типы `time [ (p) ] without time zone` и `time [ (p) ] with time zone`. Тип `time` без уточнения эквивалентен типу `time without time zone`.

Допустимые вводимые значения этих типов состоят из записи времени суток и необязательного указания часового пояса. (См. Таблицу 8.11 и Таблицу 8.12.) Если в значении для типа `time without time zone` указывается часовой пояс, он просто игнорируется. Так же будет игнорироваться дата, если её указать, за исключением случаев, когда в указанном часовом поясе принят переход на летнее время, например `America/New_York`. В данном случае указать дату необходимо, чтобы система могла определить, применяется ли обычное или летнее время. Соответствующее смещение часового пояса записывается в значении `time with time zone` и выводится без изменений, оно не привязано к активному часовому поясу.

Таблица 8.11. Вводимое время

Пример	Описание
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601

Пример	**Описание**
`040506`	ISO 8601
`04:05 AM`	то же, что и 04:05; AM не меняет значение времени
`04:05 PM`	то же, что и 16:05; часы должны быть <= 12
`04:05:06.789-8`	ISO 8601, с часовым поясом в виде смещения от UTC
`04:05:06-08:00`	ISO 8601, с часовым поясом в виде смещения от UTC
`04:05-08:00`	ISO 8601, с часовым поясом в виде смещения от UTC
`040506-08`	ISO 8601, с часовым поясом в виде смещения от UTC
`040506+0730`	ISO 8601, с часовым поясом, задаваемым нецелочисленным смещением от UTC
`040506+07:30:00`	смещение от UTC, заданное до секунд (не допускается в ISO 8601)
`04:05:06 PST`	часовой пояс задаётся аббревиатурой
`2003-04-12 04:05:06 America/New_York`	часовой пояс задаётся полным названием

Таблица 8.12. Вводимый часовой пояс

Пример	**Описание**
`PST`	аббревиатура (Pacific Standard Time, Стандартное тихоокеанское время)
`America/New_York`	полное название часового пояса
`PST8PDT`	указание часового пояса в стиле POSIX
`-8:00:00`	смещение часового пояса PST от UTC
`-8:00`	смещение часового пояса PST от UTC (расширенный формат ISO 8601)
`-800`	смещение часового пояса PST от UTC (стандартный формат ISO 8601)
`-8`	смещение часового пояса PST от UTC (стандартный формат ISO 8601)
`zulu`	принятое у военных сокращение UTC
`z`	краткая форма `zulu` (также определена в ISO 8601)

Подробнее узнать о том, как указывается часовой пояс, можно в Подразделе 8.5.3.

8.5.1.3. Даты и время

Допустимые значения типов timestamp состоят из записи даты и времени, после которого может указываться часовой пояс и необязательное уточнение `BC` или `AD`, определяющее эпоху до нашей эры и нашу эру соответственно. (`BC`/`AD` можно указать и перед часовым поясом, но предпочтительнее первый вариант.) Таким образом:

```
1999-01-08 04:05:06
```

и

```
1999-01-08 04:05:06 -8:00
```

допустимые варианты, соответствующие стандарту ISO 8601. В дополнение к этому поддерживается распространённый формат:

```
January 8 04:05:06 1999 PST
```

Стандарт SQL различает константы типов `timestamp without time zone` и `timestamp with time zone` по знаку «+» или «-» и смещению часового пояса, добавленному после времени. Следовательно, согласно стандарту, записи

```
TIMESTAMP '2004-10-19 10:23:54'
```

должен соответствовать тип `timestamp without time zone`, а

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

тип `timestamp with time zone`. Postgres Pro никогда не анализирует содержимое текстовой строки, чтобы определить тип значения, и поэтому обе записи будут обработаны как значения типа `timestamp without time zone`. Чтобы текстовая константа обрабатывалась как `timestamp with time zone`, укажите этот тип явно:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

В значении типа `timestamp without time zone` Postgres Pro просто игнорирует часовой пояс. То есть результирующее значение вычисляется только из полей даты/времени и не подстраивается под указанный часовой пояс.

Для значений `timestamp with time zone`, входная строка, содержащая явное указание часового пояса, будет преобразована в UTC (*Universal Coordinated Time*, Всемирное координированное время) с учётом смещения данного часового пояса. Если во входной строке не указан часовой пояс, подразумевается часовой пояс, заданный системным параметром TimeZone и время так же пересчитывается в UTC со смещением `timezone`. В обоих случаях значение хранится внутри системы как UTC, а изначально указанный или предполагаемый часовой пояс не сохраняется.

Когда значение `timestamp with time zone` выводится, оно всегда преобразуется из UTC в текущий часовой пояс `timezone` и отображается как локальное время. Чтобы получить время для другого часового пояса, нужно либо изменить `timezone`, либо воспользоваться конструкцией `AT TIME ZONE` (см. Подраздел 9.9.4).

В преобразованиях между `timestamp without time zone` и `timestamp with time zone` обычно предполагается, что значение `timestamp without time zone` содержит местное время (для часового пояса `timezone`). Другой часовой пояс для преобразования можно задать с помощью `AT TIME ZONE`.

8.5.1.4. Специальные значения

Postgres Pro для удобства поддерживает несколько специальных значений даты/времени, перечисленных в Таблице 8.13. Значения `infinity` и `-infinity` имеют особое представление в системе и они отображаются в том же виде, тогда как другие варианты при чтении преобразуются в значения даты/времени. (В частности, `now` и подобные строки преобразуются в актуальные значения времени в момент чтения.) Чтобы использовать эти значения в качестве констант в командах SQL, их нужно заключать в апострофы.

Таблица 8.13. Специальные значения даты/времени

Вводимая строка	Допустимые типы	Описание
`epoch`	`date`, `timestamp`	1970-01-01 00:00:00+00 (точка отсчёта времени в Unix)
`infinity`	`date`, `timestamp`, `interval`	время после максимальной допустимой даты

Вводимая строка	**Допустимые типы**	**Описание**
`-infinity`	`date`, `timestamp`, `interval`	время до минимальной допустимой даты
`now`	`date`, `time`, `timestamp`	время начала текущей транзакции
`today`	`date`, `timestamp`	время начала текущих суток (`00:00`)
`tomorrow`	`date`, `timestamp`	время начала следующих суток (`00:00`)
`yesterday`	`date`, `timestamp`	время начала предыдущих суток (`00:00`)
`allballs`	`time`	00:00:00.00 UTC

Для получения текущей даты/времени соответствующего типа можно также использовать следующие SQL-совместимые функции: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME` и `LOCALTIMESTAMP`. (См. Подраздел 9.9.5.) Заметьте, что во входных строках эти SQL-функции *не* распознаются.

Внимание

Входные значения `now`, `today`, `tomorrow` и `yesterday` вполне корректно работают в интерактивных SQL-командах, но когда команды сохраняются для последующего выполнения, например в подготовленных операторах, представлениях или определениях функций, их поведение может быть неожиданным. Такая строка может преобразоваться в конкретное значение времени, которое затем будет использоваться гораздо позже момента, когда оно было получено. В таких случаях следует использовать одну из SQL-функций. Например, `CURRENT_DATE + 1` будет работать надёжнее, чем `'tomorrow'::date`.

8.5.2. Вывод даты/времени

В качестве выходного формата типов даты/времени можно использовать один из четырёх стилей: ISO 8601, SQL (Ingres), традиционный формат POSTGRES (формат date в Unix) или German. По умолчанию выбран формат ISO. (Стандарт SQL требует, чтобы использовался именно ISO 8601. Другой формат называется «SQL» исключительно по историческим причинам.) Примеры всех стилей вывода перечислены в Таблице 8.14. Вообще со значениями типов `date` и `time` выводилась бы только часть даты или времени из показанных примеров, но со стилем POSTGRES значение даты без времени выводится в формате ISO.

Таблица 8.14. Стили вывода даты/время

Стиль	**Описание**	**Пример**
`ISO`	ISO 8601, стандарт SQL	`1997-12-17 07:37:16-08`
`SQL`	традиционный стиль	`12/17/1997 07:37:16.00 PST`
`Postgres`	изначальный стиль	`Wed Dec 17 07:37:16 1997 PST`
`German`	региональный стиль	`17.12.1997 07:37:16.00 PST`

Примечание

ISO 8601 указывает, что дата должна отделяться от времени буквой `T` в верхнем регистре. Postgres Pro принимает этот формат при вводе, но при выводе вставляет вместо `T` пробел, как

> показано выше. Это сделано для улучшения читаемости и для совместимости с *RFC 3339* и другими СУБД.

В стилях SQL и POSTGRES день выводится перед месяцем, если установлен порядок DMY, а в противном случае месяц выводится перед днём. (Как этот параметр также влияет на интерпретацию входных значений, описано в Подразделе 8.5.1) Соответствующие примеры показаны в Таблице 8.15.

Таблица 8.15. Соглашения о порядке компонентов даты

Параметр `datestyle`	**Порядок при вводе**	**Пример вывода**
`SQL, DMY`	*день*/*месяц*/*год*	`17/12/1997 15:37:16.00 CET`
`SQL, MDY`	*месяц*/*день*/*год*	`12/17/1997 07:37:16.00 PST`
`Postgres, DMY`	*день*/*месяц*/*год*	`Wed 17 Dec 07:37:16 1997 PST`

В формате ISO часовой пояс всегда отображается в виде числового смещения со знаком относительно всемирного координированного времени (UTC); для зон к востоку от Гринвича знак смещения положительный. Смещение будет отображаться как *hh* (только часы), если оно задаётся целым числом часов, как *hh*:*mm*, если оно задаётся целым числом минут, или как *hh*:*mm*:*ss*. (Третий вариант невозможен в современных стандартах часовых поясов, но он может понадобиться при работе с отметками времени, предшествующими принятию стандартизированных часовых поясов.) В других форматах даты часовой пояс отображается как буквенное сокращение, если оно принято в текущем поясе. В противном случае он отображается как числовое смещение со знаком в стандартном формате ISO 8601 (*hh* или *hhmm*).

Стиль даты/времени пользователь может выбрать с помощью команды `SET datestyle`, параметра DateStyle в файле конфигурации `postgresql.conf` или переменной окружения `PGDATESTYLE` на сервере или клиенте.

Для большей гибкости при форматировании выводимой даты/времени можно использовать функцию `to_char` (см. Раздел 9.8).

8.5.3. Часовые пояса

Часовые пояса и правила их применения определяются, как вы знаете, не только по географическим, но и по политическим соображениям. Часовые пояса во всём мире были более-менее стандартизированы в начале прошлого века, но они продолжают претерпевать изменения, в частности это касается перехода на летнее время. Для расчёта времени в прошлом Postgres Pro получает исторические сведения о правилах часовых поясов из распространённой базы данных IANA (Olson). Для будущего времени предполагается, что в заданном часовом поясе будут продолжать действовать последние принятые правила.

Postgres Pro стремится к совместимости со стандартом SQL в наиболее типичных случаях. Однако стандарт SQL допускает некоторые странности при смешивании типов даты и времени. Две очевидные проблемы:

- Хотя для типа `date` часовой пояс указать нельзя, это можно сделать для типа `time`. В реальности это не очень полезно, так как без даты нельзя точно определить смещение при переходе на летнее время.
- По умолчанию часовой пояс задаётся постоянным смещением от UTC. Это также не позволяет учесть летнее время при арифметических операций с датами, пересекающими границы летнего времени.

Поэтому мы советуем использовать часовой пояс с типами, включающими и время, и дату. Мы *не* рекомендуем использовать тип `time with time zone` (хотя Postgres Pro поддерживает его для старых приложений и совместимости со стандартом SQL). Для типов, включающих только дату или только время, в Postgres Pro предполагается местный часовой пояс.

Все значения даты и времени с часовым поясом представляются внутри в UTC, а при передаче клиентскому приложению они переводятся в местное время, при этом часовой пояс по умолчанию определяется параметром конфигурации TimeZone.

Postgres Pro позволяет задать часовой пояс тремя способами:

- Полное название часового пояса, например `America/New_York`. Все допустимые названия перечислены в представлении `pg_timezone_names` (см. Раздел 52.35). Определения часовых поясов Postgres Pro берёт из широко распространённой базы IANA, так что имена часовых поясов Postgres Pro будут воспринимать и другие приложения.
- Аббревиатура часового пояса, например `PST`. Такое определение просто задаёт смещение от UTC, в отличие от полных названий поясов, которые кроме того подразумевают и правила перехода на летнее время. Распознаваемые аббревиатуры перечислены в представлении `pg_timezone_abbrevs` (см. Раздел 52.34). Аббревиатуры можно использовать во вводимых значениях даты/времени и в операторе `AT TIME ZONE`, но не в параметрах конфигурации TimeZone и log_timezone.
- Помимо аббревиатур и названий часовых поясов Postgres Pro принимает указания часовых поясов в стиле POSIX, как описано в Разделе B.5. Этот вариант обычно менее предпочтителен, чем использование именованного часового пояса, но он может быть единственным возможным, если для нужного часового пояса нет записи в базе данных IANA.

Вкратце, различие между аббревиатурами и полными названиями заключаются в следующем: аббревиатуры представляют определённый сдвиг от UTC, а полное название подразумевает ещё и местное правило по переходу на летнее время, то есть, возможно, два сдвига от UTC. Например, `2014-06-04 12:00 America/New_York` представляет полдень по местному времени в Нью-Йорк, что для данного дня было бы летним восточным временем (EDT или UTC-4). Так что `2014-06-04 12:00 EDT` обозначает тот же момент времени. Но `2014-06-04 12:00 EST` задаёт стандартное восточное время (UTC-5), не зависящее о того, действовало ли летнее время в этот день.

Мало того, в некоторых юрисдикциях одна и та же аббревиатура часового пояса означала разные сдвиги UTC в разное время; например, аббревиатура московского времени `MSK` несколько лет означала UTC+3, а затем стала означать UTC+4. Postgres Pro обрабатывает такие аббревиатуры в соответствии с их значениями на заданную дату, но, как и с примером выше `EST`, это не обязательно будет соответствовать местному гражданскому времени в этот день.

Независимо от формы, регистр в названиях и аббревиатурах часовых поясов не важен. (В PostgreSQL до версии 8.2 он где-то имел значение, а где-то нет.)

Ни названия, ни аббревиатуры часовых поясов, не зашиты в самом сервере; они считываются из файлов конфигурации, находящихся в путях `.../share/timezone/` и `.../share/timezonesets/` относительно каталога установки (см. Раздел B.4).

Параметр конфигурации TimeZone можно установить в `postgresql.conf` или любым другим стандартным способом, описанным в Главе 18. Часовой пояс может быть также определён следующими специальными способами:

- Часовой пояс для текущего сеанса можно установить с помощью SQL-команды `SET TIME ZONE`. Это альтернативная запись команды `SET TIMEZONE TO`, более соответствующая SQL-стандарту.
- Если установлена переменная окружения `PGTZ`, клиенты libpq используют её значение, выполняя при подключении к серверу команду `SET TIME ZONE`.

8.5.4. Ввод интервалов

Значения типа `interval` могут быть записаны в следующей расширенной форме:

```
[@] количество единица [количество единица...] [направление]
```

где *`количество`* — это число (возможно, со знаком); *`единица`* — одно из значений: `microsecond`, `millisecond`, `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium` (которые обозначают соответственно микросекунды, миллисекунды, секунды, минуты, часы, дни, недели, месяцы, годы, десятилетия, века и тысячелетия), либо эти же слова во множественном числе, либо их сокращения; *`направление`* может принимать значение `ago` (назад) или быть пустым. Знак `@` является необязательным. Все заданные величины различных единиц суммируются вместе с учётом знака чисел. Указание `ago` меняет знак всех полей на противоположный. Этот синтаксис также используется при выводе интервала, если параметр IntervalStyle имеет значение `postgres_verbose`.

Количества дней, часов, минут и секунд можно определить, не указывая явно соответствующие единицы. Например, запись `'1 12:59:10'` равнозначна `'1 day 12 hours 59 min 10 sec'`. Сочетание года и месяца также можно записать через минус; например `'200-10'` означает то, же что и `'200 years 10 months'`. (На самом деле только эти краткие формы разрешены стандартом SQL и они используются при выводе, когда `IntervalStyle` имеет значение `sql_standard`.)

Интервалы можно также записывать в виде, определённом в ISO 8601, либо в «формате с кодами», описанном в разделе 4.4.3.2 этого стандарта, либо в «альтернативном формате», описанном в разделе 4.4.3.3. Формат с кодами выглядит так:

```
P количество единица [ количество единица ...] [ T [ количество единица ...]]
```

Строка должна начинаться с символа `P` и может включать также `T` перед временем суток. Допустимые коды единиц перечислены в Таблице 8.16. Коды единиц можно опустить или указать в любом порядке, но компоненты времени суток должны идти после символа `T`. В частности, значение кода `M` зависит от того, располагается ли он до или после `T`.

Таблица 8.16. Коды единиц временных интервалов ISO 8601

Код	Значение
Y	годы
M	месяцы (в дате)
W	недели
D	дни
H	часы
M	минуты (во времени)
S	секунды

В альтернативном формате:

```
P [ год-месяц-день ] [ T часы:минуты:секунды ]
```

строка должна начинаться с `P`, а `T` разделяет компоненты даты и времени. Значения выражаются числами так же, как и в датах ISO 8601.

При записи интервальной константы с указанием *`полей`* или присваивании столбцу типа interval строки с *`полями`*, интерпретация непомеченных величин зависит от *`полей`*. Например, `INTERVAL '1' YEAR` воспринимается как 1 год, а `INTERVAL '1'` — как 1 секунда. Кроме того, значения «справа» от меньшего значащего поля, заданного в определении *`полей`*, просто отбрасываются. Например, в записи `INTERVAL '1 day 2:03:04' HOUR TO MINUTE` будут отброшены секунды, но не день.

Согласно стандарту SQL, все компоненты значения interval должны быть одного знака, и ведущий минус применяется ко всем компонентам; например, минус в записи `'-1 2:03:04'` применяется и к дню, и к часам/минутам/секундам. Postgres Pro позволяет задавать для разных компонентов разные знаки и традиционно обрабатывает знак каждого компонента в текстовом представлении отдельно от других, так что в данном случае часы/минуты/секунды будут считаться положительными. Если параметр `IntervalStyle` имеет значение `sql_standard`, ведущий знак применяется ко всем компонентам (но только если они не содержат знаки явно). В противном случае действуют традиционные правила Postgres Pro. Во избежание неоднозначности рекомендуется добавлять знак к каждому компоненту с отрицательным значением.

Значения `interval` хранятся в виде трёх целочисленных полей: месяцы, дни и микросекунды. Эти поля хранятся отдельно, поскольку количество дней в месяце варьируется, а день может состоять из 23 или 25 часов при переходе на летнее/зимнее время. Входная строка интервала, в которой используются другие единицы измерения, нормализуется в этот формат, а затем стандартизированным образом реконструируется для вывода, например:

```
SELECT '2 years 15 months 100 weeks 99 hours 123456789 milliseconds'::interval;
               interval
---------------------------------------
 3 years 3 mons 700 days 133:17:36.789
```

Здесь недели, под которыми понимаются «7 дней», были сохранены отдельно, а меньшие и большие единицы измерения времени были объединены и нормализованы.

Значения полей могут содержать дробную часть, например `'1.5 weeks'` или `'01:02:03.45'`. Однако поскольку `interval` содержит только целые единицы, дробные значения должны быть преобразованы в более мелкие единицы. Дробные части единиц, превышающих месяцы, округляются до целого числа месяцев, например `'1.5 years'` преобразуется в `'1 year 6 mons'`. Дробные части недель и дней пересчитываются в целое число дней и микросекунд, из расчёта, что в месяце 30 дней, а в сутках — 24 часа, например, `'1.75 months'` становится `1 mon 22 days 12:00:00`. На выходе только секунды могут иметь дробную часть.

В Таблице 8.17 показано несколько примеров допустимых вводимых значений типа `interval`.

Таблица 8.17. Ввод интервалов

Пример	Описание
`1-2`	Стандартный формат SQL: 1 год и 2 месяца
`3 4:05:06`	Стандартный формат SQL: 3 дня 4 часа 5 минут 6 секунд
`1 year 2 months 3 days 4 hours 5 minutes 6 seconds`	Традиционный формат Postgres: 1 год 2 месяца 3 дня 4 часа 5 минут 6 секунд
`P1Y2M3DT4H5M6S`	«Формат с кодами» ISO 8601: то же значение, что и выше
`P0001-02-03T04:05:06`	«Альтернативный формат» ISO 8601: то же значение, что и выше

8.5.5. Вывод интервалов

Как объяснялось ранее, Postgres Pro хранит значения `interval` в виде месяцев, дней и микросекунд. При выводе значение месяцев преобразуется в годы и месяцы путём деления на 12. Значение в днях отображается как есть. Значение в микросекундах преобразуется в часы, минуты, секунды и доли секунды. Таким образом, месяцы, минуты и секунды никогда не будут выводиться вне диапазонов 0–11, 0–59 и 0–59 соответственно, тогда как отображаемые значения в годах, днях и часах могут быть довольно большими. (Чтобы перенести большие значения дней и часов в следующее поле, можно использовать функции `justify_days` и `justify_hours`.)

Формат вывода типа interval может определяться одним из четырёх стилей: `sql_standard`, `postgres`, `postgres_verbose` и `iso_8601`. Выбрать нужный стиль позволяет команда `SET intervalstyle` (по умолчанию выбран `postgres`). Примеры форматов разных стилей показаны в Таблице 8.18.

Стиль `sql_standard` выдаёт результат, соответствующий стандарту SQL, если значение интервала удовлетворяет ограничениям стандарта (и содержит либо только год и месяц, либо только день и время, и при этом все его компоненты одного знака). В противном случае выводится год-месяц, за которым идёт дата-время, а в компоненты для однозначности явно добавляются знаки.

Вывод в стиле `postgres` соответствует формату, который был принят в PostgreSQL до версии 8.4, когда параметр DateStyle имел значение `ISO`.

Вывод в стиле `postgres_verbose` соответствует формату, который был принят в PostgreSQL до версии 8.4, когда значением параметром `DateStyle` было не `ISO`.

Вывод в стиле `iso_8601` соответствует «формату с кодами» описанному в разделе 4.4.3.2 формата ISO 8601.

Таблица 8.18. Примеры стилей вывода интервалов

Стиль	Интервал год-месяц	Интервал день-время	Смешанный интервал
`sql_standard`	1-2	3 4:05:06	-1-2 +3 -4:05:06
`postgres`	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
`postgres_verbose`	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
`iso_8601`	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.6. Логический тип

В Postgres Pro есть стандартный SQL-тип `boolean`; см. Таблицу 8.19. Тип `boolean` может иметь следующие состояния: «true», «false» и третье состояние, «unknown», которое представляется SQL-значением NULL.

Таблица 8.19. Логический тип данных

Имя	Размер	Описание
`boolean`	1 байт	состояние: истина или ложь

Логические константы могут представляться в SQL-запросах следующими ключевыми словами SQL: `TRUE`, `FALSE` и `NULL`.

Функция ввода данных типа `boolean` воспринимает следующие строковые представления состояния «true»:

```
true
yes
on
1
```

и следующие представления состояния «false»:

```
false
no
off
0
```

Также воспринимаются уникальные префиксы этих строк, например `t` или `n`. Регистр символов не имеет значения, а пробельные символы в начале и в конце строки игнорируются.

Функция вывода данных типа `boolean` всегда выдаёт `t` или `f`, как показано в Примере 8.2.

Пример 8.2. Использование типа `boolean`

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
```

```
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

Ключевые слова `TRUE` и `FALSE` являются предпочтительными (соответствующими стандарту SQL) для записи логических констант в SQL-запросах. Но вы также можете использовать строковые представления, которые допускает синтаксис строковых констант, описанный в Подразделе 4.1.2.7, например, `'yes'::boolean`.

Заметьте, что при анализе запроса `TRUE` и `FALSE` автоматически считаются значениями типа `boolean`, но для `NULL` это не так, потому что ему может соответствовать любой тип. Поэтому в некоторых контекстах может потребоваться привести `NULL` к типу `boolean` явно, например так: `NULL::boolean`. С другой стороны, приведение строковой константы к логическому типу можно опустить в тех контекстах, где анализатор запроса может понять, что буквальное значение должно иметь тип `boolean`.

8.7. Типы перечислений

Типы перечислений (enum) определяют статический упорядоченный набор значений, так же как и типы `enum`, существующие в ряде языков программирования. В качестве перечисления можно привести дни недели или набор состояний.

8.7.1. Объявление перечислений

Тип перечислений создаются с помощью команды CREATE TYPE, например так:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Созданные типы enum можно использовать в определениях таблиц и функций, как и любые другие:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

8.7.2. Порядок

Порядок значений в перечислении определяется последовательностью, в которой были указаны значения при создании типа. Перечисления поддерживаются всеми стандартными операторами сравнения и связанными агрегатными функциями. Например:

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
```

```
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
-------
 Larry
(1 row)
```

8.7.3. Безопасность типа

Все типы перечислений считаются уникальными и поэтому значения разных типов нельзя сравнивать. Взгляните на этот пример:

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ОШИБКА:  неверное значение для перечисления happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ОШИБКА:  оператор не существует: mood = happiness
```

Если вам действительно нужно сделать что-то подобное, вы можете либо реализовать собственный оператор, либо явно преобразовать типы в запросе:

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

8.7.4. Тонкости реализации

В метках значений регистр имеет значение, т. е. `'happy'` и `'HAPPY'` — не одно и то же. Также в метках имеют значение пробелы.

Хотя типы-перечисления предназначены прежде всего для статических наборов значений, имеется возможность добавлять новые значения в существующий тип-перечисление и переименовывать значения (см. ALTER TYPE). Однако удалять существующие значения из перечисления, а также изменять их порядок, нельзя — для получения нужного результата придётся удалить и воссоздать это перечисление.

Значение enum занимает на диске 4 байта. Длина текстовой метки значения ограничена параметром компиляции `NAMEDATALEN`; в стандартных сборках Postgres Pro он ограничивает длину 63 байтами.

Сопоставления внутренних значений enum с текстовыми метками хранятся в системном каталоге `pg_enum`. Он может быть полезен в ряде случаев.

8.8. Геометрические типы

Геометрические типы данных представляют объекты в двумерном пространстве. Все существующие в Postgres Pro геометрические типы перечислены в Таблице 8.20.

Таблица 8.20. Геометрические типы

Имя	Размер	Описание	Представление
`point`	16 байт	Точка на плоскости	(x,y)
`line`	24 байта	Бесконечная прямая	{A,B,C}
`lseg`	32 байта	Отрезок	[(x1,y1),(x2,y2)]
`box`	32 байта	Прямоугольник	(x1,y1),(x2,y2)
`path`	16+16n байт	Закрытый путь (подобный многоугольнику)	((x1,y1),...)
`path`	16+16n байт	Открытый путь	[(x1,y1),...]
`polygon`	40+16n байт	Многоугольник (подобный закрытому пути)	((x1,y1),...)
`circle`	24 байта	Окружность	<(x,y),r> (центр окружности и радиус)

Во всех этих типах отдельные координаты хранятся как числа типа `double precision` (`float8`).

Для выполнения различных геометрических операций, в частности масштабирования, вращения и определения пересечений, PostgreSQL предлагает богатый набор функций и операторов. Они рассматриваются в Разделе 9.11.

8.8.1. Точки

Точки — это основной элемент, на базе которого строятся все остальные геометрические типы. Значения типа `point` записываются в одном из двух форматов:

```
( x , y )
  x , y
```

где *x* и *y* — координаты точки на плоскости, выраженные числами с плавающей точкой.

Выводятся точки в первом формате.

8.8.2. Прямые

Прямые представляются линейным уравнением *A*x + *B*y + *C* = 0, где *A* и *B* не могут быть равны 0 одновременно. Значения типа `line` вводятся и выводятся в следующем виде:

```
{ A, B, C }
```

Кроме того, для ввода может использоваться любая из этих форм:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где `(x1,y1)` и `(x2,y2)` — две различные точки на данной прямой.

8.8.3. Отрезки

Отрезок представляется парой точек, определяющих концы отрезка. Значения типа `lseg` записываются в одной из следующих форм:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где `(x1,y1)` и `(x2,y2)` — концы отрезка.

Выводятся отрезки в первом формате.

8.8.4. Прямоугольники

Прямоугольник представляется двумя точками, находящимися в противоположных его углах. Значения типа `box` записываются в одной из следующих форм:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

где `(x1,y1)` и `(x2,y2)` — противоположные углы прямоугольника.

Выводятся прямоугольники во второй форме.

Во вводимом значении могут быть указаны любые два противоположных угла, но затем они будут упорядочены, так что внутри сохранятся правый верхний и левый нижний углы, в таком порядке.

8.8.5. Пути

Пути представляют собой списки соединённых точек. Пути могут быть *закрытыми*, когда подразумевается, что первая и последняя точка в списке соединены, или *открытыми*, в противном случае.

Значения типа `path` записываются в одной из следующих форм:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

где точки задают узлы сегментов, составляющих путь. Квадратные скобки (`[]`) указывают, что путь открытый, а круглые (`()`) — закрытый. Когда внешние скобки опускаются, как в показанных выше последних трёх формах, считается, что путь закрытый.

Пути выводятся в первой или второй форме, в соответствии с типом.

8.8.6. Многоугольники

Многоугольники представляются списками точек (вершин). Многоугольники похожи на закрытые пути, но важное семантическое отличие состоит в том, что многоугольник считается содержащим область внутри него, а путь — нет.

Важное различие в реализации между многоугольниками и путями заключается в том, что сохранённое представление многоугольника включает его наименьший окружающий прямоугольник. Это ускоряет определённые операции поиска, хотя вычисление окружающего прямоугольника добавляет издержки при построении новых многоугольников.

Значения типа `polygon` записываются в одной из следующих форм:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
```

```
( x1 , y1   , ... ,   xn , yn )
  x1 , y1   , ... ,   xn , yn
```

где точки задают узлы сегментов, образующих границу многоугольника.

Выводятся многоугольники в первом формате.

8.8.7. Окружности

Окружности задаются координатами центра и радиусом. Значения типа `circle` записываются в одном из следующих форматов:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

где `(x,y)` — центр окружности, а `r` — её радиус.

Выводятся окружности в первом формате.

8.9. Типы, описывающие сетевые адреса

Postgres Pro предлагает типы данных для хранения адресов IPv4, IPv6 и MAC, показанные в Таблице 8.21. Для хранения сетевых адресов лучше использовать эти типы, а не простые текстовые строки, так как Postgres Pro проверяет вводимые значения данных типов и предоставляет специализированные операторы и функции для работы с ними (см. Раздел 9.12).

Таблица 8.21. Типы, описывающие сетевые адреса

Имя	Размер	Описание
`cidr`	7 или 19 байт	Сети IPv4 и IPv6
`inet`	7 или 19 байт	Узлы и сети IPv4 и IPv6
`macaddr`	6 байт	MAC-адреса
`macaddr8`	8 байт	MAC-адреса (в формате EUI-64)

При сортировке типов `inet` и `cidr`, адреса IPv4 всегда идут до адресов IPv6, в том числе адреса IPv4, включённые в IPv6 или сопоставленные с ними, например ::10.2.3.4 или ::ffff:10.4.3.2.

8.9.1. `inet`

Тип `inet` содержит IPv4- или IPv6-адрес узла и может также содержать его подсеть, всё в одном поле. Подсеть представляется числом бит, определяющих адрес сети в адресе узла (или «маску сети»). Если маска сети равна 32 для адреса IPv4, такое значение представляет не подсеть, а определённый узел. Адреса IPv6 имеют длину 128 бит, поэтому уникальный адрес узла задаётся с маской 128 бит. Заметьте, что когда нужно, чтобы принимались только адреса сетей, следует использовать тип `cidr`, а не `inet`.

Вводимые значения такого типа должны иметь формат *`адрес/y`*, где *`адрес`* — адрес IPv4 или IPv6, а *`y`* — число бит в маске сети. Если компонент *`/y`* опущен, маска сети считается равной 32 для IPv4 и 128 для IPv6, так что это значение будет представлять один узел. При выводе компонент *`/y`* опускается, если сетевой адрес определяет адрес одного узла.

8.9.2. `cidr`

Тип `cidr` содержит определение сети IPv4 или IPv6. Входные и выходные форматы соответствуют соглашениям CIDR (Classless Internet Domain Routing, Бесклассовая межсетевая адресация). Определение сети записывается в формате *`адрес/y`*, где *`адрес`* — минимальный адрес в сети, представленный в виде адреса IPv4 или IPv6, а *`y`* — число бит в маске сети. Если *`y`* не указывается, это значение вычисляется по старой классовой схеме нумерации сетей, но при этом оно может быть увеличено, чтобы в него вошли все байты введённого адреса. Если в сетевом адресе справа от маски сети окажутся биты со значением 1, он будет считаться ошибочным.

В Таблице 8.22 показаны несколько примеров адресов.

Таблица 8.22. Примеры допустимых значений типа `cidr`

Вводимое значение `cidr`	**Выводимое значение `cidr`**	`abbrev( cidr)`
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. Различия `inet` и `cidr`

Существенным различием типов данных `inet` и `cidr` является то, что `inet` принимает значения с ненулевыми битами справа от маски сети, а `cidr` — нет. Например, значение `192.168.0.1/24` является допустимым для типа `inet`, но не для `cidr`.

Подсказка

Если вас не устраивает выходной формат значений `inet` или `cidr`, попробуйте функции `host`, `text` и `abbrev`.

8.9.4. `macaddr`

Тип `macaddr` предназначен для хранения MAC-адреса, примером которого является адрес сетевой платы Ethernet (хотя MAC-адреса применяются и для других целей). Вводимые значения могут задаваться в следующих форматах:

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

Все эти примеры определяют один и тот же адрес. Шестнадцатеричные цифры от `a` до `f` могут быть и в нижнем, и в верхнем регистре. Выводятся MAC-адреса всегда в первой форме.

В стандарте IEEE 802-2001 стандартной считается вторая форма MAC-адресов (с минусами), а первая (с двоеточиями) предполагает обратный порядок бит, от старшего к младшему, так что 08-00-2b-01-02-03 = 10:00:D4:80:40:C0. В настоящее время этому соглашению практически никто не следует, и уместно оно было только для устаревших сетевых протоколов (таких как Token Ring). Postgres Pro не меняет порядок бит и во всех принимаемых форматах подразумевается традиционный порядок LSB.

Последние пять входных форматов не описаны ни в каком стандарте.

8.9.5. `macaddr8`

Тип `macaddr8` хранит MAC-адреса в формате EUI-64, применяющиеся, например, для аппаратных адресов плат Ethernet (хотя MAC-адреса используются и для других целей). Этот тип может принять и 6-байтовые, и 8-байтовые адреса MAC и сохраняет их в 8 байтах. MAC-адреса, заданные в 6-байтовом формате, хранятся в формате 8 байт, а 4-ый и 5-ый байт содержат FF и FE, соответственно. Заметьте, что для IPv6 используется модифицированный формат EUI-64, в котором 7-ой бит должен быть установлен в 1 после преобразования из EUI-48. Для выполнения этого изменения предоставляется функция `macaddr8_set7bit`. Вообще говоря, этот тип принимает любые строки, состоящие из пар шестнадцатеричных цифр (выровненных по границам байт), которые могут согласованно разделяться одинаковыми символами `':'`, `'-'` или `'.'`. Шестнадцатеричных цифр должно быть либо 16 (для 8 байт), либо 12 (для 6 байт). Начальные и конечные пробелы игнорируются. Ниже показаны примеры допустимых входных строк:

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
```

Во всех этих примерах задаётся один и тот же адрес. Для цифр с `a` по `f` принимаются буквы и в верхнем, и в нижнем регистре. Вывод всегда представляется в первом из показанных форматов.

Последние шесть входных форматов из показанных выше не являются стандартизированными.

Чтобы преобразовать традиционный 48-битный MAC-адрес в формате EUI-48 в модифицированный формат EUI-64 для включения в состав адреса IPv6 в качестве адреса узла, используйте функцию `macaddr8_set7bit` следующим образом:

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

    macaddr8_set7bit
-------------------------
 0a:00:2b:ff:fe:01:02:03
(1 row)
```

8.10. Битовые строки

Битовые строки представляют собой последовательности из 1 и 0. Их можно использовать для хранения или отображения битовых масок. В SQL есть два битовых типа: `bit(n)` и `bit varying(n)`, где *n* — положительное целое число.

Длина значения типа `bit` должна в точности равняться *n*; при попытке сохранить данные длиннее или короче произойдёт ошибка. Данные типа `bit varying` могут иметь переменную длину, но не превышающую *n*; строки большей длины не будут приняты. Запись `bit` без указания длины равнозначна записи `bit(1)`, тогда как `bit varying` без указания длины подразумевает строку неограниченной длины.

> ### Примечание
>
> При попытке привести значение битовой строки к типу `bit(n)`, оно будет усечено или дополнено нулями справа до длины ровно *n* бит, ошибки при этом не будет. Подобным образом, если явно привести значение битовой строки к типу `bit varying(n)`, она будет усечена справа, если её длина превышает *n* бит.

Синтаксис констант битовых строк описан в Подразделе 4.1.2.5, а все доступные битовые операторы и функции перечислены в Разделе 9.6.

Пример 8.3. Использование битовых строк

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ОШИБКА:  длина битовой строки (2) не соответствует типу bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

Для хранения битовой строки используется по 1 байту для каждой группы из 8 бит, плюс 5 или 8 байт дополнительно в зависимости от длины строки (но длинные строки могут быть сжаты или вынесены отдельно, как описано в Разделе 8.3 применительно к символьным строкам).

8.11. Типы, предназначенные для текстового поиска

Postgres Pro предоставляет два типа данных для поддержки полнотекстового поиска. Текстовым поиском называется операция анализа набора *документов* с текстом на естественном языке, в результате которой находятся фрагменты, наиболее соответствующие *запросу*. Тип `tsvector` представляет документ в виде, оптимизированном для текстового поиска, а `tsquery` представляет запрос текстового поиска в подобном виде. Более подробно это описывается в Главе 12, а все связанные функции и операторы перечислены в Разделе 9.13.

8.11.1. `tsvector`

Значение типа `tsvector` содержит отсортированный список неповторяющихся *лексем*, т. е. слов, *нормализованных* так, что все словоформы сводятся к одной (подробнее это описано в Главе 12). Сортировка и исключение повторяющихся слов производится автоматически при вводе значения, как показано в этом примере:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

Для представления в виде лексем пробелов или знаков препинания их нужно заключить в апострофы:

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                 tsvector
-------------------------------------------
```

```
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

(В данном и следующих примерах мы используем строку в долларах, чтобы не дублировать все апострофы в таких строках.) При этом включаемый апостроф или обратную косую черту нужно продублировать:

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
------------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Также для лексем можно указать их целочисленные *позиции*:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

Позиция обычно указывает положение исходного слова в документе. Информация о расположении слов затем может использоваться для *оценки близости*. Позиция может задаваться числом от 1 до 16383; большие значения просто заменяются на 16383. Если для одной лексемы дважды указывается одно положение, такое повторение отбрасывается.

Лексемам, для которых заданы позиции, также можно назначить *вес*, выраженный буквами `A`, `B`, `C` или `D`. Вес `D` подразумевается по умолчанию и поэтому он не показывается при выводе:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Веса обычно применяются для отражения структуры документа, например для придания особого значения словам в заголовке по сравнению со словами в обычном тексте. Назначенным весам можно сопоставить числовые приоритеты в функциях ранжирования результатов.

Важно понимать, что тип `tsvector` сам по себе не выполняет нормализацию слов; предполагается, что в сохраняемом значении слова уже нормализованы приложением. Например:

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

Для большинства англоязычных приложений приведённые выше слова будут считаться ненормализованными, но для `tsvector` это не важно. Поэтому исходный документ обычно следует обработать функцией `to_tsvector`, нормализующей слова для поиска:

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

И это подробнее описано в Главе 12.

8.11.2. `tsquery`

Значение `tsquery` содержит искомые лексемы, объединяемые логическими операторами `&` (И), `|` (ИЛИ) и `!` (НЕ), а также оператором поиска фраз `<->` (ПРЕДШЕСТВУЕТ). Также допускается вариация оператора ПРЕДШЕСТВУЕТ вида `<N>`, где *N* — целочисленная константа, задающая расстояние между двумя искомыми лексемами. Запись оператора `<->` равнозначна `<1>`.

Для группировки операторов могут использоваться скобки. Без скобок эти операторы имеют разные приоритеты, в порядке убывания: `!` (НЕ), `<->` (ПРЕДШЕСТВУЕТ), `&` (И) и `|` (ИЛИ).

Несколько примеров:

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

Лексемам в `tsquery` можно дополнительно сопоставить буквы весов, при этом они будут соответствовать только тем лексемам в `tsvector`, которые имеют какой-либо из этих весов:

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

Кроме того, в лексемах `tsquery` можно использовать знак `*` для поиска по префиксу:

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

Этот запрос найдёт все слова в `tsvector`, начинающиеся с приставки «super».

Апострофы в лексемах этого типа можно использовать так же, как и в лексемах в `tsvector`; и так же, как и для типа `tsvector`, необходимая нормализация слова должна выполняться до приведения значения к типу `tsquery`. Для такой нормализации удобно использовать функцию `to_tsquery`:

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

Заметьте, что функция `to_tsquery` будет обрабатывать префиксы подобно другим словам, поэтому следующее сравнение возвращает true:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
```

так как `postgres` преобразуется стеммером в `postgr`:

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
  to_tsvector   | to_tsquery
---------------+------------
 'postgradu':1 | 'postgr':*
```

и эта приставка находится в преобразованной форме слова `postgraduate`.

8.12. Тип UUID

Тип данных `uuid` сохраняет универсальные уникальные идентификаторы (Universally Unique Identifiers, UUID), определённые в *RFC 4122*, ISO/IEC 9834-8:2005 и связанных стандартах. (В некоторых системах это называется GUID, глобальным уникальным идентификатором.) Этот идентификатор представляет собой 128-битное значение, генерируемое специальным алгоритмом, практически гарантирующим, что этим же алгоритмом оно не будет получено больше нигде в мире. Таким образом, эти идентификаторы будут уникальными и в распределённых системах, а не только в единственной базе данных, как значения генераторов последовательностей.

UUID записывается в виде последовательности шестнадцатеричных цифр в нижнем регистре, разделённых знаками минуса на несколько групп, в таком порядке: группа из 8 цифр, за ней три группы из 4 цифр и, наконец, группа из 12 цифр, что в сумме составляет 32 цифры и представляет 128 бит. Пример UUID в этом стандартном виде:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

Postgres Pro также принимает альтернативные варианты: цифры в верхнем регистре, стандартную запись, заключённую в фигурные скобки, запись без минусов или с минусами, разделяющими любые группы из четырёх цифр. Например:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

Выводится значение этого типа всегда в стандартном виде.

Как можно сгенерировать UUID в Postgres Pro, описывается в Разделе 9.14.

8.13. Тип XML

Тип `xml` предназначен для хранения XML-данных. Его преимущество по сравнению с обычным типом `text` в том, что он проверяет вводимые значения на допустимость по правилам XML и для работы с ним есть типобезопасные функции; см. Раздел 9.15. Для использования этого типа дистрибутив должен быть скомпилирован в конфигурации `configure --with-libxml`.

Тип `xml` может сохранять правильно оформленные «документы», в соответствии со стандартом XML, а также фрагменты «содержимого», определяемые как менее ограниченные *«узлы документа»* в модели данных XQuery и XPath. Другими словами, это означает, что во фрагментах содержимого может быть несколько элементов верхнего уровня или текстовых узлов. Является ли некоторое значение типа `xml` полным документом или фрагментом содержимого, позволяет определить выражение `xml-значение IS DOCUMENT`.

Информацию о совместимости и ограничениях типа данных `xml` можно найти в Разделе D.3.

8.13.1. Создание XML-значений

Чтобы получить значение типа `xml` из текстовой строки, используйте функцию `xmlparse`:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Примеры:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</
chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Хотя в стандарте SQL описан только один способ преобразования текстовых строк в XML-значения, специфический синтаксис Postgres Pro:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

тоже допустим.

Тип `xml` не проверяет вводимые значения по схеме DTD (Document Type Declaration, Объявления типа документа), даже если в них присутствуют ссылка на DTD. В настоящее время в PostgreSQL также нет встроенной поддержки других разновидностей схем, например XML Schema.

Обратная операция, получение текстовой строки из `xml`, выполняется с помощью функции `xmlserialize`:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } значение AS тип [ [ NO ] INDENT ] )
```

Здесь допустимый *тип* — `character`, `character varying` или `text` (или их псевдонимы). И в данном случае стандарт SQL предусматривает только один способ преобразования `xml` в тип текстовых строк, но Postgres Pro позволяет просто привести значение к нужному типу.

С параметром `INDENT` результат будет визуально улучшен, а с `NO INDENT` (по умолчанию) сохранится исходное форматирование результата. Если значение приводится к символьному типу, сохраняется исходное форматирование.

При преобразовании текстовой строки в тип `xml` или наоборот без использования функций `XMLPARSE` и `XMLSERIALIZE`, выбор режима `DOCUMENT` или `CONTENT` определяется параметром конфигурации сеанса «XML option», установить который можно следующей стандартной командой:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

или такой командой в духе Postgres Pro:

```
SET xmloption TO { DOCUMENT | CONTENT };
```

По умолчанию этот параметр имеет значение `CONTENT`, так что допускаются все формы XML-данных.

8.13.2. Обработка кодировки

Если на стороне сервера и клиента и в XML-данных используются разные кодировки символов, с этим могут возникать проблемы. Когда запросы передаются на сервер, а их результаты возвращаются клиенту в обычном текстовом режиме, Postgres Pro преобразует все передаваемые текстовые данные в кодировку для соответствующей стороны; см. Раздел 22.3. В том числе это происходит и со строковыми представлениями XML-данных, подобными тем, что показаны в предыдущих примерах. Обычно это означает, что объявления кодировки, содержащиеся в XML-данных, могут не соответствовать действительности, когда текстовая строка преобразуется из одной кодировки в другую при передаче данных между клиентом и сервером, так как подобные включённые в данные объявления не будут изменены автоматически. Для решения этой проблемы объявления кодировки, содержащиеся в текстовых строках, вводимых в тип `xml`, просто *игнорируются* и предполагается, что XML-содержимое представлено в текущей кодировке сервера. Как следствие, для правильной обработки таких строк с XML-данными клиент должен передавать их в своей текущей кодировке. Для сервера не важно, будет ли клиент для этого преобразовывать документы в свою кодировку, или изменит её, прежде чем передавать ему данные. При выводе значения типа `xml` не содержат объявления кодировки, а клиент должен предполагать, что все данные поступают в его текущей кодировке.

Если параметры запроса передаются на сервер и он возвращает результаты клиенту в двоичном режиме, кодировка символов не преобразуется, так что возникает другая ситуация. В этом случае объявление кодировки в XML принимается во внимание, а если его нет, то предполагается, что данные закодированы в UTF-8 (это соответствует стандарту XML; заметьте, что Postgres Pro не поддерживает UTF-16). При выводе в данные будет добавлено объявление кодировки, выбранной на стороне клиента (но если это UTF-8, объявление будет опущено).

Само собой, XML-данные в Postgres Pro будут обрабатываться гораздо эффективнее, когда и в XML-данных, и на стороне клиента, и на стороне сервера используется одна кодировка. Так как внутри XML-данные представляются в UTF-8, оптимальный вариант, когда на сервере также выбрана кодировка UTF-8.

Внимание

Некоторые XML-функции способны работать исключительно с ASCII-данными, если кодировка сервера не UTF-8. В частности, это известная особенность функций `xmltable()` и `xpath()`.

8.13.3. Обращение к XML-значениям

Тип `xml` отличается от других тем, что для него не определены никакие операторы сравнения, так как чётко определённого и универсального алгоритма сравнения XML-данных не существует. Одно из следствий этого — нельзя отфильтровать строки таблицы, сравнив столбец `xml` с искомым значением. Поэтому обычно XML-значения должны дополняться отдельным ключевым полем, например ID. Можно также сравнивать XML-значения, преобразовав их сначала в текстовые строки, но заметьте, что с учётом специфики XML-данных этот метод практически бесполезен.

Из-за отсутствия операторов сравнения для типа `xml`, для столбца этого типа также нельзя создать индекс. Поэтому, когда требуется быстрый поиск в XML данных, обойти это ограничение можно, приведя данные к типу текстовой строки и проиндексировав эти строки, либо проиндексировав выражение XPath. Конечно сам запрос при этом следует изменить, чтобы поиск выполнялся по индексированному выражению.

Для ускорения поиска в XML-данных также можно использовать функции полнотекстового поиска в Postgres Pro. Однако это требует определённой подготовки данных, что дистрибутив Postgres Pro пока не поддерживает.

8.14. Типы JSON

Типы JSON предназначены для хранения данных JSON (JavaScript Object Notation, Запись объекта JavaScript) согласно стандарту *RFC 7159*. Такие данные можно хранить и в типе `text`, но типы JSON лучше тем, что проверяют, соответствует ли вводимое значение формату JSON. Для работы с ними есть также несколько специальных функций и операторов; см. Раздел 9.16.

В Postgres Pro имеются два типа для хранения данных JSON: `json` и `jsonb`. Для реализации эффективного механизма запросов к этим типам данных в Postgres Pro также имеется тип `jsonpath`, описанный в Подразделе 8.14.7.

Типы данных `json` и `jsonb` принимают на вход *почти* одинаковые наборы значений, а отличаются они главным образом с точки зрения эффективности. Тип `json` сохраняет точную копию введённого текста, которую функции обработки должны разбирать заново при каждом выполнении запроса, тогда как данные `jsonb` сохраняются в разобранном двоичном формате, что несколько замедляет ввод из-за преобразования, но значительно ускоряет обработку, не требуя многократного разбора текста. Кроме того, `jsonb` поддерживает индексацию, что тоже может быть очень полезно.

Так как тип `json` сохраняет точную копию введённого текста, он сохраняет семантически незначащие пробелы между элементами, а также порядок ключей в JSON-объектах. И если JSON-объект внутри содержит повторяющиеся ключи, этот тип сохранит все пары ключ/значение. (Функции обработки будут считать действительной последнюю пару.) Тип `jsonb`, напротив, не сохраняет пробелы, порядок ключей и значения с дублирующимися ключами. Если во входных данных оказываются дублирующиеся ключи, сохраняется только последнее значение.

Для большинства приложений предпочтительнее хранить данные JSON в типе `jsonb` (если нет особых противопоказаний, например важны прежние предположения о порядке ключей объектов).

В RFC 7159 говорится, что строки JSON должны быть представлены в кодировке UTF-8. Поэтому данные JSON не будут полностью соответствовать спецификации, если кодировка базы данных не UTF-8. При этом нельзя будет вставить в JSON символы, непредставимые в кодировке сервера, и наоборот, допустимыми будут символы, представимые в кодировке сервера, но не в UTF-8.

RFC 7159 разрешает включать в строки JSON спецпоследовательности Unicode в виде `\u`*`XXXX`*. В функцию ввода для типа `json` эти спецпоследовательности допускаются вне зависимости от кодировки базы данных, и проверяется только правильность их синтаксиса (за `\u` должны следовать четыре шестнадцатеричных цифры). Однако функция ввода для типа `jsonb` более строгая: она не допускает спецпоследовательности Unicode для символов, которые не могут быть представлены в кодировке базы. Тип `jsonb` также не принимает `\u0000` (так как это значение не может быть представлено в типе `text` Postgres Pro) и требует, чтобы суррогатные пары Unicode использовались для представления символов вне основной многоязыковой плоскости (BMP) правильно. Корректные спецпоследовательности Unicode преобразуются для хранения в один соответствующий символ (это подразумевает сворачивание суррогатных пар в один символ).

> **Примечание**
>
> Многие из функций обработки JSON, описанные в Разделе 9.16, преобразуют спецпоследовательности Unicode в обычные символы, поэтому могут выдавать подобные ошибки, даже если им на вход поступает тип `json`, а не `jsonb`. То, что функция ввода в тип `json` не производит этих проверок, можно считать историческим артефактом, хотя это и позволяет просто сохранять (но не обрабатывать) в JSON спецкоды Unicode в базе данных с кодировкой, в которой представленные таким образом символы отсутствуют.

При преобразовании вводимого текста JSON в тип `jsonb`, примитивные типы, описанные в RFC 7159, по сути отображаются в собственные типы Postgres Pro как показано в Таблице 8.23. Таким образом, к содержимому типа `jsonb` предъявляются некоторые дополнительные требования, продиктованные ограничениями представления нижележащего типа данных, которые не распространяются ни на тип `json`, ни на формат JSON вообще. В частности, тип `jsonb` не принимает числа, выходящие за диапазон типа данных Postgres Pro `numeric`, тогда как с `json` такого ограничения нет. Такие ограничения, накладываемые реализацией, допускаются согласно RFC 7159. Однако на практике такие проблемы более вероятны в других реализациях, так как обычно примитивный тип JSON `number` представляется в виде числа с плавающей точкой двойной точности IEEE 754 (что RFC 7159 явно признаёт и допускает). При использовании JSON в качестве формата обмена данными с такими системами следует учитывать риски потери точности чисел, хранившихся в Postgres Pro.

И напротив, как показано в таблице, есть некоторые ограничения в формате ввода примитивных типов JSON, не актуальные для соответствующих типов Postgres Pro.

Таблица 8.23. Примитивные типы JSON и соответствующие им типы Postgres Pro

Примитивный тип JSON	Тип Postgres Pro	Примечания
`string`	`text`	`\u0000` не допускается как спецпоследовательность Unicode, представляющая символ, который отсутствует в кодировке базы
`number`	`numeric`	Значения `NaN` и `infinity` не допускаются
`boolean`	`boolean`	Допускаются только варианты `true` и `false` (в нижнем регистре)
`null`	(нет)	`NULL` в SQL имеет другой смысл

8.14.1. Синтаксис вводимых и выводимых значений JSON

Синтаксис ввода-вывода типов данных JSON соответствует стандарту RFC 7159.

Примеры допустимых выражений с типом `json` (или `jsonb`):

```
-- Простое скалярное/примитивное значение
-- Простыми значениями могут быть числа, строки в кавычках, true, false или null
SELECT '5'::json;

-- Массив из нуля и более элементов (элементы могут быть разных типов)
SELECT '[1, 2, "foo", null]'::json;

-- Объект, содержащий пары ключей и значений
-- Заметьте, что ключи объектов — это всегда строки в кавычках
```

```
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Массивы и объекты могут вкладываться произвольным образом
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

Как было сказано ранее, когда значение JSON вводится и затем выводится без дополнительной обработки, тип `json` выводит тот же текст, что поступил на вход, а `jsonb` не сохраняет семантически незначащие детали, такие как пробелы. Например, посмотрите на эти различия:

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                      json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                      jsonb
--------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

Первая семантически незначимая деталь, заслуживающая внимания: с `jsonb` числа выводятся по правилам нижележащего типа `numeric`. На практике это означает, что числа, заданные в записи с `E`, будут выведены без неё, например:

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

Однако как видно из этого примера, `jsonb` сохраняет конечные нули дробного числа, хотя они и не имеют семантической значимости, в частности для проверки на равенство.

Список встроенных функций и операторов, позволяющих создавать и обрабатывать значения JSON, приведён в Разделе 9.16.

8.14.2. Проектирование документов JSON

Представлять данные в JSON можно гораздо более гибко, чем в традиционной реляционной модели данных, что очень привлекательно там, где нет жёстких условий. И оба этих подхода вполне могут сосуществовать и дополнять друг друга в одном приложении. Однако даже для приложений, которым нужна максимальная гибкость, рекомендуется, чтобы документы JSON имели некоторую фиксированную структуру. Эта структура обычно не навязывается жёстко (хотя можно декларативно диктовать некоторые бизнес-правила), но когда она предсказуема, становится гораздо проще писать запросы, которые извлекают полезные данные из набора «документов» (информации) в таблице.

Данные JSON, как и данные любых других типов, хранящиеся в таблицах, находятся под контролем механизма параллельного доступа. Хотя хранить большие документы вполне возможно, не забывайте, что при любом изменении устанавливается блокировка всей строки (на уровне строки). Поэтому для оптимизации блокировок транзакций, изменяющих данные, стоит ограничить размер документов JSON разумными пределами. В идеале каждый документ JSON должен собой представлять атомарный информационный блок, который, согласно бизнес-логике, нельзя разделить на меньшие, индивидуально изменяемые блоки.

8.14.3. Проверки на вхождение и существование `jsonb`

Проверка *вхождения* — важная особенность типа `jsonb`, не имеющая аналога для типа `json`. Эта проверка определяет, входит ли один документ `jsonb` в другой. В следующих примерах возвращается истинное значение (кроме упомянутых исключений):

```
-- Простые скалярные/примитивные значения включают только одно идентичное значение:
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- Массив с правой стороны входит в массив слева:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- Порядок элементов в массиве не важен, поэтому это условие тоже выполняется:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- А повторяющиеся элементы массива не имеют значения:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- Объект с одной парой справа входит в объект слева:
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb @>
 '{"version": 9.4}'::jsonb;

-- Массив справа не считается входящим в
-- массив слева, хотя в последний и вложен подобный массив:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- выдаёт false

-- Но если добавить уровень вложенности, проверка на вхождение выполняется:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- Аналогично, это вхождением не считается:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;  -- выдаёт false

-- Ключ с пустым объектом на верхнем уровне входит в объект с таким ключом:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

Общий принцип этой проверки в том, что входящий объект должен соответствовать объекту, содержащему его, по структуре и данным, возможно, после исключения из содержащего объекта лишних элементов массива или пар ключ/значение. Но помните, что порядок элементов массива для проверки на вхождение не имеет значения, а повторяющиеся элементы массива считаются только один раз.

В качестве особого исключения для требования идентичности структур, массив может содержать примитивное значение:

```
-- В этот массив входит примитивное строковое значение:
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- Это исключение действует только в одну сторону -- здесь вхождения нет:
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- выдаёт false
```

Для типа `jsonb` введён также оператор *существования*, который является вариацией на тему вхождения: он проверяет, является ли строка (заданная в виде значения `text`) ключом объекта или элементом массива на верхнем уровне значения `jsonb`. В следующих примерах возвращается истинное значение (кроме упомянутых исключений):

```
-- Строка существует в качестве элемента массива:
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- Строка существует в качестве ключа объекта:
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Значения объектов не рассматриваются:
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- выдаёт false

-- Как и вхождение, существование определяется на верхнем уровне:
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- выдаёт false
```

```
-- Строка считается существующей, если она соответствует примитивной строке JSON:
SELECT '"foo"'::jsonb ? 'foo';
```

Объекты JSON для проверок на существование и вхождение со множеством ключей или элементов подходят больше, чем массивы, так как, в отличие от массивов, они внутри оптимизируются для поиска, и поиск элемента не будет линейным.

> ### Подсказка
>
> Так как вхождение в JSON проверяется с учётом вложенности, правильно написанный запрос может заменить явную выборку внутренних объектов. Например, предположим, что у нас есть столбец `doc`, содержащий объекты на верхнем уровне, и большинство этих объектов содержит поля `tags` с массивами вложенных объектов. Данный запрос найдёт записи, в которых вложенные объекты содержат ключи `"term":"paris"` и `"term":"food"`, и при этом пропустит такие ключи, находящиеся вне массива `tags`:
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
> ```
>
> Этого же результата можно добиться, например, так:
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
> ```
>
> Но данный подход менее гибкий и часто также менее эффективный.
>
> С другой стороны, оператор существования JSON не учитывает вложенность: он будет искать заданный ключ или элемент массива только на верхнем уровне значения JSON.

Различные операторы вхождения и существования, а также все другие операторы и функции для работы с JSON документированы в Разделе 9.16.

8.14.4. Индексация `jsonb`

Для эффективного поиска ключей или пар ключ/значение в большом количестве документов `jsonb` можно успешно применять индексы GIN. Для этого предоставляются два «класса операторов» GIN, предлагающие выбор между производительностью и гибкостью.

Класс операторов GIN по умолчанию для `jsonb` поддерживает запросы с операторами существования ключа (`?`, `?|` и `?&`), оператором включения (`@>`) и операторами соответствия для `jsonpath` (`@?` и `@@`). (Подробнее семантика, реализуемая этими операторами, описана в Таблице 9.46.) Пример создания индекса с этим классом операторов:

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

Дополнительный класс операторов GIN `jsonb_path_ops` не поддерживает операторы существования ключа, но поддерживает `@>`, `@?` и `@@`. Пример создания индекса с этим классом операторов:

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

Рассмотрим пример таблицы, в которой хранятся документы JSON, получаемые от сторонней веб-службы, с документированным определением схемы. Типичный документ:

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
```

```
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

Мы сохраняем эти документы в таблице `api`, в столбце `jdoc` типа `jsonb`. Если по этому столбцу создаётся GIN-индекс, он может применяться в подобных запросах:

```
-- Найти документы, в которых ключ "company" имеет значение "Magnafone"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';
```

Однако в следующих запросах он не будет использоваться, потому что, несмотря на то, что оператор `?` — индексируемый, он применяется не к индексированному столбцу `jdoc` непосредственно:

```
-- Найти документы, в которых ключ "tags" содержит ключ или элемент массива "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

И всё же, правильно применяя индексы выражений, в этом запросе можно задействовать индекс. Если запрос определённых элементов в ключе `"tags"` выполняется часто, вероятно стоит определить такой индекс:

```
CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));
```

Теперь предложение `WHERE jdoc -> 'tags' ? 'qui'` будет выполняться как применение индексируемого оператора `?` к индексируемому выражению `jdoc -> 'tags'`. (Подробнее об индексах выражений можно узнать в Разделе 11.7.)

Ещё один подход к использованию проверок на существование:

```
-- Найти документы, в которых ключ "tags" содержит элемент массива "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

Этот запрос может задействовать простой GIN-индекс по столбцу `jdoc`. Но заметьте, что такой индекс будет хранить копии всех ключей и значений в поле `jdoc`, тогда как индекс выражения из предыдущего примера хранит только данные внутри объекта с ключом `tags`. Хотя подход с простым индексом гораздо более гибкий (так как он поддерживает запросы по любому ключу), индексы конкретных выражений скорее всего будут меньше и быстрее, чем простые индексы.

Также индексы GIN поддерживают операторы `@?` и `@@`, которые сопоставляют `jsonpath` с данными. Примеры:

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @? '$.tags[*] ? (@ == "qui")';
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@ '$.tags[*] == "qui"';
```

Для этих операторов индекс GIN извлекает из `jsonpath` предложения вида *цепочка_обращения* `==` *константа* и выполняет поиск по ключам и значениям из этих предложений. Цепочка обращения может включать указания обращения `.`*ключ*, `[*]` и `[`*индекс*`]`. Класс операторов `jsonb_ops` также поддерживает указания `.*` и `.**`, которые не поддерживает класс операторов `jsonb_path_ops`.

Класс операторов `jsonb_path_ops` поддерживает только запросы с операторами `@>`, `@?` и `@@`, но он значительно производительнее класса по умолчанию `jsonb_ops`. Индекс `jsonb_path_ops` обычно гораздо меньше индекса `jsonb_ops` для тех же данных и более точен при поиске, особенно если запросы обращаются к ключам, часто встречающимся в данных. Таким образом, с ним операции поиска выполняются гораздо эффективнее, чем с классом операторов по умолчанию.

Техническое различие между GIN-индексами `jsonb_ops` и `jsonb_path_ops` состоит в том, что для первых создаются независимые элементы индекса для каждого ключа/значения в данных, тогда как для вторых создаются элементы только для значений. [1] По сути, каждый элемент индекса `jsonb_path_ops` представляет собой хеш значения и ключа(ей), приводящего к нему; например, при индексации `{"foo": {"bar": "baz"}}` будет создан один элемент индекса с хешем, рассчитан-

---

[1] Поэтому понятие «значение» включает и элементы массивов, хотя в терминологии JSON иногда элементы массивов считаются отличными от значений внутри объектов.

ным по всем трём значениям: `foo`, `bar` и `baz`. Таким образом, проверка на вхождение этой структуры будет использовать крайне точный поиск по индексу, но определить, является ли `foo` ключом, с помощью такого индекса нельзя. С другой стороны, индекс `jsonb_ops` создаст три отдельных элемента индекса, представляющих `foo`, `bar` и `baz` по отдельности; для выполнения проверки на вхождение будут проверены строки таблицы, содержащие все эти три значения. Хотя GIN-индексы позволяют вычислить AND довольно эффективно, такой поиск всё же будет менее точным и более медленным, чем равнозначный поиск с `jsonb_path_ops`, особенно если любое одно из этих трёх значений содержится в большом количестве строк.

Недостаток класса `jsonb_path_ops` заключается в том, что он не учитывает в индексе структуры JSON, не содержащие никаких значений `{"a": {}}`. Для поиска по документам, содержащих такие структуры, потребуется выполнить полное сканирование индекса, что довольно долго, поэтому `jsonb_path_ops` не очень подходит для приложений, часто выполняющих такие запросы.

Тип `jsonb` также поддерживает индексы `btree` и `hash`. Они полезны, только если требуется проверять равенство JSON-документов в целом. Порядок сортировки `btree` для типа `jsonb` редко имеет большое значение, но для полноты он приводится ниже:

```
Объект > Массив > Логическое значение > Число > Строка > null

Объект с n парами > Объект с n - 1 парами

Массив с n элементами > Массив с n - 1 элементами
```

за исключением того, что (по историческим причинам) пустой массив верхнего уровня сортируется как меньший, чем *`null`*. Объекты с равным количеством пар сравниваются в таком порядке:

```
ключ-1, значение-1, ключ-2 ...
```

Заметьте, что ключи объектов сравниваются согласно порядку при хранении; в частности, из-за того, что короткие ключи хранятся перед длинными, результаты могут оказаться несколько не интуитивными:

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

Массивы с равным числом элементом упорядочиваются аналогично:

```
элемент-1, элемент-2 ...
```

Примитивные значения JSON сравниваются по тем же правилам сравнения, что и нижележащие типы данных Postgres Pro. Строки сравниваются с учётом порядка сортировки по умолчанию в текущей базе данных.

8.14.5. Обращение по индексу к элементам `jsonb`

Тип данных `jsonb` поддерживает извлечение и изменение элементов в стиле обращения к элементам массива. Указывать на вложенные значения можно, задавая цепочку обращений к элементам, при этом будут действовать правила использования аргумента `path` функции `jsonb_set`. Если значение `jsonb` является массивом, числовые индексы начинаются с нуля, а отрицательные целые числа отсчитывают элементы с конца массива к началу. Обращения к срезам массивов не поддерживаются. Результат обращения по индексу всегда имеет тип jsonb.

Используя операцию обращения по индексу в предложении `SET` оператора `UPDATE`, можно изменять значения `jsonb`. Путь такого обращения должен быть «проходимым» для всех указанных значений, если они существуют. Например, путь `val['a']['b']['c']` можно пройти полностью до `c`, если `val`, `val['a']` и `val['a']['b']` — объекты. Если же значение `val['a']` или `val['a']['b']` не определено, будет создан пустой объект, заполняемый по мере необходимости. Однако, если значение собственно `val` или любое из промежуточных значений существует и является не объектом, а, например, строкой, числом или сущностью `jsonb null`, пройти этот путь невозможно, поэтому возникает ошибка и транзакция прерывается.

Пример синтаксиса обращения по индексу:

```
-- Извлечь значение объекта по ключу
```

```
SELECT ('{"a": 1}'::jsonb)['a'];

-- Извлечь значение вложенного объекта по пути ключа
SELECT ('{"a": {"b": {"c": 1}}}'::jsonb)['a']['b']['c'];

-- Извлечь элемент массива по индексу
SELECT ('[1, "2", null]'::jsonb)[1];

-- Изменить значение объекта по ключу. Обратите внимание на апострофы вокруг '1':
 присваиваемое
-- значение также должно быть типа jsonb
 UPDATE table_name SET jsonb_field['key'] = '1';

-- Это вызовет ошибку, если jsonb_field['a']['b'] в какой-либо записи является не
 объектом.
-- Например, в {"a": 1} ключу 'a' соответствует числовое значение
-- ключа 'a'.
UPDATE table_name SET jsonb_field['a']['b']['c'] = '1';

-- Отфильтровать записи предложением WHERE с обращением по ключу. Поскольку результат
-- обращения по ключу будет иметь тип jsonb, такой же тип должно иметь сравниваемое с
 ним значение.
-- Двойные кавычки добавлены, чтобы строка "value" стала допустимой строкой jsonb.
SELECT * FROM table_name WHERE jsonb_field['key'] = '"value"';
```

Присваивание `jsonb` при обращении по индексу в некоторых особых случаях работает не так, как с функцией `jsonb_set`. Когда исходное значение `jsonb` — `NULL`, присваивание при обращении по ключу будет работать, как будто это значение — пустое значение JSON (типа массив или объект, в зависимости от типа ключа):

```
-- Там, где поле jsonb_field было NULL, оно станет {"a": 1}
UPDATE table_name SET jsonb_field['a'] = '1';

-- Там, где поле jsonb_field было NULL, оно станет [1]
UPDATE table_name SET jsonb_field[0] = '1';
```

Если индекс указан для массива, содержащего недостаточно элементов, в него будут добавляться элементы со значением `NULL` до тех пор, пока индекс не станет достижимым и пока не станет возможным задать значение.

```
-- Там, где поле jsonb_field было [], оно станет [null, null, 2];
-- там, где поле jsonb_field было [0], оно станет [0, null, 2]
UPDATE table_name SET jsonb_field[2] = '2';
```

Значение `jsonb` будет принимать присваивания по несуществующим путям, если последний существующий элемент, который нужно пройти, — объект или массив, в зависимости от соответствующего ключа (элемент, на который указывает последний компонент пути, не проходится и может быть любым). Вложенные структуры массивов и объектов будут созданы и для первых дополнятся элементами `null` до нужной позиции (заданной в пути), в которую может быть помещено присваиваемое значение.

```
-- Там, где поле jsonb_field было {}, оно станет {"a": [{"b": 1}]}
UPDATE table_name SET jsonb_field['a'][0]['b'] = '1';

-- Там, где поле jsonb_field было [], оно станет [null, {"a": 1}]
UPDATE table_name SET jsonb_field[1]['a'] = '1';
```

8.14.6. Трансформации

Для различных процедурных языков представлены дополнительные расширения, реализующие трансформации для типа `jsonb`.

Расширения для PL/Perl называются `jsonb_plperl` и `jsonb_plperlu`. Когда они используются, значения `jsonb` отображаются в соответствующие структуры Perl: массивы, хеши или скаляры.

Расширение для PL/Python называется `jsonb_plpython3u`. Когда оно используется, значения `jsonb` отображаются в соответствующие структуры Python: словари, списки или скаляры.

Из этих расширений «доверенным» считается `jsonb_plperl`, то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных. Остальные расширения могут устанавливать только суперпользователи.

8.14.7. Тип jsonpath

Тип `jsonpath` предназначен для реализации поддержки языка путей SQL/JSON в Postgres Pro, позволяющего эффективно выполнять запросы к данным JSON. Он обеспечивает двоичное представление разобранного выражения пути SQL/JSON, определяющего, какие элементы должны извлекаться из данных JSON для дальнейшей обработки в функциях SQL/JSON.

Семантика предикатов и операторов языка путей SQL/JSON в целом соответствует SQL. В то же время, чтобы с данными JSON можно было оперировать естественным образом, в синтаксисе путей SQL/JSON приняты некоторые соглашения JavaScript:

- Точка (`.`) применяется для доступа к члену объекта.
- Квадратные скобки (`[]`) применяются для обращения к массиву.
- Элементы массивов в SQL/JSON нумеруются с 0, тогда как обычные массивы SQL — с 1.

Числовые константы в выражениях путей SQL/JSON подчиняются правилам JavaScript, которые отличаются от SQL и JSON в некоторых мелких деталях. Например, пути SQL/JSON допускают `.1` и `1.`, которые недопустимы в JSON. Поддерживаются недесятичные целочисленные константы и разделители-подчёркивания, например, `1_000_000`, `0x1EEE_FFFF`, `0o273`, `0b100101`. В путях SQL/JSON (и в JavaScript, но не в самом SQL) не должно быть разделителя-подчёркивания непосредственно после префикса системы счисления.

Выражение пути SQL/JSON обычно записывается в SQL-запросе в виде символьной константы SQL, и поэтому должно заключаться в апострофы, а любой апостроф, который нужно заключить в это значение, должен дублироваться (см. Подраздел 4.1.2.1). Нередко строковые константы требуется использовать и внутри выражений путей. На такие константы распространяются соглашения JavaScript/ECMAScript: они должны заключаться в двойные кавычки, а для представления символов, которые сложно ввести иначе, используются спецпоследовательности с обратной косой чертой. В частности, символ двойных кавычек внутри строковой константы записывается как `\"`, а собственно обратная косая черта как `\\`. В число других спецпоследовательностей, воспринимаемых в строках JavaScript, входят: `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, выражающие различные управляющие символы ASCII, `\x`*`NN`*, выражающая символ кодом в виде только двух шестнадцатеричных цифр, `\u`*`NNNN`*, выражающая символ Unicode кодом в виде четырёх шестнадцатеричных цифр, а также `\u{`*`N...`*`}`, позволяющая для записи кода символа Unicode использовать от 1 до 6 шестнадцатеричных цифр.

Выражение пути состоит из последовательности элементов пути, которые могут быть следующими:

- Константы примитивных типов JSON: текст Unicode, числа и значения true, false и null.
- Переменные пути перечислены в Таблице 8.24.
- Операторы обращения перечислены в Таблице 8.25.
- Операторы и методы `jsonpath` перечислены в Подразделе 9.16.2.3.
- Скобки, применяющиеся для образования выражений фильтра и изменения порядка вычисления пути.

Более подробно использование выражений `jsonpath` с функциями запросов SQL/JSON описано в Подразделе 9.16.2.

Таблица 8.24. Переменные `jsonpath`

Переменная	Описание
`$`	Переменная, представляющая значение JSON, фигурирующее в запросе (*элемент контекста*).
`$varname`	Именованная переменная. Её значение может быть задано в параметре *`vars`*, который принимают различные функции обработки JSON; подробности в Таблице 9.49.
`@`	Переменная, представляющая результат вычисления пути в выражениях фильтров.

Таблица 8.25. Операторы обращения в `jsonpath`

Оператор обращения	Описание
`.`*`key`* `."$`*`имя_переменной`*`"`	Оператор обращения к члену объекта, выбираемому по заданному ключу. Если имя ключа совпадает с именем какой-либо переменной, начинающимся с `$`, или не соответствует действующим в JavaScript требованиям к идентификаторам, оно должно заключаться в двойные кавычки и таким образом представляться как строковая константа.
`.*`	Оператор обращения по звёздочке, который возвращает значения всех членов, находящихся на верхнем уровне объекта.
`.**`	Рекурсивный оператор обращения по звёздочке, который проходит по всем уровням иерархии JSON текущего объекта и возвращает все значения членов, вне зависимости от их уровня вложенности. Это реализованное в Postgres Pro расширение стандарта SQL/JSON.
`.**{`*`уровень`*`}` `.**{`*`начальный_уровень`*` to `*`конечный_уровень`*`}`	Этот оператор подобен `.**`, но выбирает только указанные уровни иерархии JSON. Уровни вложенности задаются целыми числами, при этом нулевой уровень соответствует текущему объекту. Для обращения к самому нижнему уровню вложенности можно использовать ключевое слово `last`. Это реализованное в Postgres Pro расширение стандарта SQL/JSON.
`[`*`селектор`*`, ...]`	Оператор обращения к элементу массива. *`Селектор`* может задаваться в двух формах: *`индекс`* или *`начальный_индекс`*` to `*`конечный_индекс`*. Первая форма выбирает единственный элемент по индексу. Вторая форма выбирает срез массива по двум индексам, включающий крайние элементы, соответствущие значениям *`начальный_индекс`* и *`конечный_индекс`*. Задаваемый *`индекс`* может быть целочисленным значением или выражением, возвращающим единственное число, которое автоматически приводится к целому. Индекс 0 соответствует первому элементу массива. Также в качестве индекса принимается ключевое слово `last`, обозначающее индекс последнего элемента массива, что полезно при обработке массивов неизвестной длины.
`[*]`	Оператор обращения к элементам массива по звёздочке, возвращающий все элементы массива.

8.15. Массивы

Postgres Pro позволяет определять столбцы таблицы как многомерные массивы переменной длины. Элементами массивов могут быть любые встроенные или определённые пользователями базовые типы, перечисления, составные типы, типы-диапазоны или домены.

8.15.1. Объявления типов массивов

Чтобы проиллюстрировать использование массивов, мы создадим такую таблицу:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

Как показано, для объявления типа массива к названию типа элементов добавляются квадратные скобки (`[]`). Показанная выше команда создаст таблицу `sal_emp` со столбцами типов `text` (`name`), одномерный массив с элементами `integer` (`pay_by_quarter`), представляющий квартальную зарплату работников, и двухмерный массив с элементами `text` (`schedule`), представляющий недельный график работника.

Команда `CREATE TABLE` позволяет также указать точный размер массивов, например так:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

Однако текущая реализация игнорирует все указанные размеры, т. е. фактически размер массива остаётся неопределённым.

Текущая реализация также не ограничивает число размерностей. Все элементы массивов считаются одного типа, вне зависимости от его размера и числа размерностей. Поэтому явно указывать число элементов или размерностей в команде `CREATE TABLE` имеет смысл только для документирования, на механизм работы с массивом это не влияет.

Для объявления одномерных массивов можно применять альтернативную запись с ключевым словом `ARRAY`, соответствующую стандарту SQL. Столбец `pay_by_quarter` можно было бы определить так:

```
    pay_by_quarter  integer ARRAY[4],
```

Или без указания размера массива:

```
    pay_by_quarter  integer ARRAY,
```

Заметьте, что и в этом случае Postgres Pro не накладывает ограничения на фактический размер массива.

8.15.2. Ввод значения массива

Чтобы записать значение массива в виде буквальной константы, заключите значения элементов в фигурные скобки и разделите их запятыми. (Если вам знаком C, вы найдёте, что это похоже на синтаксис инициализации структур в C.) Вы можете заключить значение любого элемента в двойные кавычки, а если он содержит запятые или фигурные скобки, это обязательно нужно сделать. (Подробнее это описано ниже.) Таким образом, общий формат константы массива выглядит так:

```
'{ значение1 разделитель значение2 разделитель ... }'
```

где *`разделитель`* — символ, указанный в качестве разделителя в соответствующей записи в таблице `pg_type`. Для стандартных типов данных, существующих в дистрибутиве Postgres Pro, разделителем является запятая (`,`), за исключением лишь типа `box`, в котором разделитель —точка с запятой (`;`). Каждое *`значение`* здесь — это либо константа типа элемента массива, либо вложенный массив. Например, константа массива может быть такой:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Эта константа определяет двухмерный массив 3x3, состоящий из трёх вложенных массивов целых чисел.

Чтобы присвоить элементу массива значение NULL, достаточно просто написать `NULL` (регистр символов при этом не имеет значения). Если же требуется добавить в массив строку, содержащую «NULL», это слово нужно заключить в двойные кавычки.

(Такого рода константы массивов на самом деле представляют собой всего лишь частный случай констант, описанных в Подразделе 4.1.2.7. Константа изначально воспринимается как строка и передаётся процедуре преобразования вводимого массива. При этом может потребоваться явно указать целевой тип.)

Теперь мы можем показать несколько операторов `INSERT`:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

Результат двух предыдущих команд:

```
SELECT * FROM sal_emp;
name |     pay_by_quarter      |                 schedule
-----+-------------------------+--------------------------------------
Bill |{10000,10000,10000,10000}|{{meeting,lunch},{training,presentation}}
Carol|{20000,25000,25000,25000}|{{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

В многомерных массивов число элементов в каждой размерности должно быть одинаковым; в противном случае возникает ошибка. Например:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  malformed array literal: "{{"meeting", "lunch"}, {"meeting"}}"
DETAIL:  Multidimensional arrays must have sub-arrays with matching dimensions.
```

Также можно использовать синтаксис конструктора `ARRAY`:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

Заметьте, что элементы массива здесь — это простые SQL-константы или выражения; и поэтому, например строки будут заключаться в одинарные апострофы, а не в двойные, как в буквальной константе массива. Более подробно конструктор `ARRAY` обсуждается в Подразделе 4.2.12.

8.15.3. Обращение к массивам

Добавив данные в таблицу, мы можем перейти к выборкам. Сначала мы покажем, как получить один элемент массива. Этот запрос получает имена сотрудников, зарплата которых изменилась во втором квартале:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];
```

```
 name
-------
 Carol
(1 row)
```

Индексы элементов массива записываются в квадратных скобках. По умолчанию в Postgres Pro действует соглашение о нумерации элементов массива с 1, то есть в массиве из *n* элементов первым считается `array[1]`, а последним — `array[n]`.

Этот запрос выдаёт зарплату всех сотрудников в третьем квартале:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

Мы также можем получать обычные прямоугольные срезы массива, то есть подмассивы. Срез массива обозначается как *`нижняя-граница`*`:`*`верхняя-граница`* для одной или нескольких размерностей. Например, этот запрос получает первые пункты в графике Билла в первые два дня недели:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

Если одна из размерностей записана в виде среза, то есть содержит двоеточие, тогда срез распространяется на все размерности. Если при этом для размерности указывается только одно число (без двоеточия), в срез войдут элемент от 1 до заданного номера. Например, в этом примере `[2]` будет равнозначно `[1:2]`:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                 schedule
-------------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

Во избежание путаницы с обращением к одному элементу, срезы лучше всегда записывать явно для всех измерений, например `[1:2][1:1]` вместо `[2][1:1]`.

Значения *`нижняя-граница`* и/или *`верхняя-граница`* в указании среза можно опустить; опущенная граница заменяется нижним или верхним пределом индексов массива. Например:

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{lunch},{presentation}}
(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

Выражение обращения к элементу массива возвратит NULL, если сам массив или одно из выражений индексов элемента равны NULL. Значение NULL также возвращается, если индекс выходит за границы массива (это не считается ошибкой). Например, если `schedule` в настоящее время имеет размерности `[1:3][1:2]`, результатом обращения к `schedule[3][3]` будет NULL. Подобным образом, при обращении к элементу массива с неправильным числом индексов возвращается NULL, а не ошибка.

Аналогично, NULL возвращается при обращении к срезу массива, если сам массив или одно из выражений, определяющих индексы элементов, равны NULL. Однако в других случаях, например, когда границы среза выходят за рамки массива, возвращается не NULL, а пустой массив (с размерностью 0). (Так сложилось исторически, что в этом срезы отличаются от обращений к обычным элементам.) Если запрошенный срез пересекает границы массива, тогда возвращается не NULL, а срез, сокращённый до области пересечения.

Текущие размеры значения массива можно получить с помощью функции `array_dims`:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

`array_dims` выдаёт результат типа `text`, что удобно скорее для людей, чем для программ. Размеры массива также можно получить с помощью функций `array_upper` и `array_lower`, которые возвращают соответственно верхнюю и нижнюю границу для указанной размерности:

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

`array_length` возвращает число элементов в указанной размерности массива:

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

`cardinality` возвращает общее число элементов массива по всем измерениям. Фактически это число строк, которое вернёт функция `unnest`:

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

 cardinality
-------------
           4
(1 row)
```

8.15.4. Изменение массивов

Значение массива можно заменить полностью так:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

или используя синтаксис `ARRAY`:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

Также можно изменить один элемент массива:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

или срез:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

При этом в указании среза может быть опущена *`нижняя-граница`* и/или *`верхняя-граница`*, но только для массива, отличного от NULL, и имеющего ненулевую размерность (иначе неизвестно, какие граничные значения должны подставляться вместо опущенных).

Сохранённый массив можно расширить, определив значения ранее отсутствовавших в нём элементов. При этом все элементы, располагающиеся между существовавшими ранее и новыми, принимают значения NULL. Например, если массив `myarray` содержит 4 элемента, после присваивания значения элементу `myarray[6]` его длина будет равна 6, а `myarray[5]` будет содержать NULL. В настоящее время подобное расширение поддерживается только для одномерных, но не многомерных массивов.

Определяя элементы по индексам, можно создавать массивы, в которых нумерация элементов может начинаться не с 1. Например, можно присвоить значение выражению `myarray[-2:7]` и таким образом создать массив, в котором будут элементы с индексами от -2 до 7.

Значения массива также можно сконструировать с помощью оператора конкатенации, `||`:

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

Оператор конкатенации позволяет вставить один элемент в начало или в конец одномерного массива. Он также может принять два *N*-мерных массива или массивы размерностей *N* и *N+1*.

Когда в начало или конец одномерного массива вставляется один элемент, в образованном в результате массиве будет та же нижняя граница, что и в массиве-операнде. Например:

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

Когда складываются два массива одинаковых размерностей, в результате сохраняется нижняя граница внешней размерности левого операнда. Выходной массив включает все элементы левого операнда, после которых добавляются все элементы правого. Например:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
```

```
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

Когда к массиву размерности *N+1* спереди или сзади добавляется *N*-мерный массив, он вставляется аналогично тому, как в массив вставляется элемент (это было описано выше). Любой *N*-мерный массив по сути является элементом во внешней размерности массива, имеющего размерность *N +1*. Например:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

Массив также можно сконструировать с помощью функций `array_prepend`, `array_append` и `array_cat`. Первые две функции поддерживают только одномерные массивы, а `array_cat` поддерживает и многомерные. Несколько примеров:

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

В простых случаях описанный выше оператор конкатенации предпочтительнее непосредственного вызова этих функций. Однако так как оператор конкатенации перегружен для решения всех трёх задач, возможны ситуации, когда лучше применить одну из этих функций во избежание неоднозначности. Например, рассмотрите:

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- нетипизированная строка воспринимается как массив
```

```
 ?column?
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';                 -- как и эта
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL;                -- как и буквальный NULL
 ?column?
----------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);    -- это могло иметься в виду на самом деле
 array_append
--------------
 {1,2,NULL}
```

В показанных примерах анализатор запроса видит целочисленный массив с одной стороны оператора конкатенации и константу неопределённого типа с другой. Согласно своим правилам разрешения типа констант, он полагает, что она имеет тот же тип, что и другой операнд — в данном случае целочисленный массив. Поэтому предполагается, что оператор конкатенации здесь представляет функцию `array_cat`, а не `array_append`. Если это решение оказывается неверным, его можно скорректировать, приведя константу к типу элемента массива; однако может быть лучше явно использовать функцию `array_append`.

8.15.5. Поиск значений в массивах

Чтобы найти значение в массиве, необходимо проверить все его элементы. Это можно сделать вручную, если вы знаете размер массива. Например:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

Однако с большим массивами этот метод становится утомительным, и к тому же он не работает, когда размер массива неизвестен. Альтернативный подход описан в Разделе 9.25. Показанный выше запрос можно было переписать так:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

А так можно найти в таблице строки, в которых массивы содержат только значения, равные 10000:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Кроме того, для обращения к элементам массива можно использовать функцию `generate_subscripts`. Например так:

```
SELECT * FROM
   (SELECT pay_by_quarter,
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

Эта функция описана в Таблице 9.68.

Также искать в массиве значения можно, используя оператор `&&`, который проверяет, перекрывается ли левый операнд с правым. Например:

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

Этот и другие операторы для работы с массивами описаны в Разделе 9.19. Он может быть ускорен с помощью подходящего индекса, как описано в Разделе 11.2.

Вы также можете искать определённые значения в массиве, используя функции `array_position` и `array_positions`. Первая функция возвращает позицию первого вхождения значения в массив, а вторая — массив позиций всех его вхождений. Например:

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
 array_position
----------------
              2
(1 row)

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
(1 row)
```

Подсказка

Массивы — это не множества; необходимость поиска определённых элементов в массиве может быть признаком неудачно сконструированной базы данных. Возможно, вместо массива лучше использовать отдельную таблицу, строки которой будут содержать данные элементов массива. Это может быть лучше и для поиска, и для работы с большим количеством элементов.

8.15.6. Синтаксис вводимых и выводимых значений массива

Внешнее текстовое представление значения массива состоит из записи элементов, интерпретируемых по правилам ввода-вывода для типа элемента массива, и оформления структуры массива. Оформление состоит из фигурных скобок (`{` и `}`), окружающих значение массива, и знаков-разделителей между его элементами. В качестве знака-разделителя обычно используется запятая (`,`), но это может быть и другой символ; он определяется параметром `typdelim` для типа элемента массива. Для стандартных типов данных, существующих в дистрибутиве Postgres Pro, разделителем является запятая (`,`), за исключением лишь типа `box`, в котором разделитель — точка с запятой (`;`). В многомерном массиве у каждой размерности (ряд, плоскость, куб и т. д.) есть свой уровень фигурных скобок, а соседние значения в фигурных скобках на одном уровне должны отделяться разделителями.

Функция вывода массива заключает значение элемента в кавычки, если это пустая строка или оно содержит фигурные скобки, знаки-разделители, кавычки, обратную косую черту, пробельный символ или это текст `NULL`. Кавычки и обратная косая черта, включённые в такие значения, преобразуются в спецпоследовательность с обратной косой чертой. Для числовых типов данных можно рассчитывать на то, что значения никогда не будут выводиться в кавычках, но для текстовых типов следует быть готовым к тому, что выводимое значение массива может содержать кавычки.

По умолчанию нижняя граница всех размерностей массива равна одному. Чтобы представить массивы с другими нижними границами, перед содержимым массива можно указать диапазоны индексов. Такое оформление массива будет содержать квадратные скобки (`[]`) вокруг нижней и верхней границ каждой размерности с двоеточием (`:`) между ними. За таким указанием размерности следует знак равно (`=`). Например:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

Процедура вывода массива включает в результат явное указание размерностей, только если нижняя граница в одной или нескольких размерностях отличается от 1.

Если в качестве значения элемента задаётся `NULL` (в любом регистре), этот элемент считается равным непосредственно NULL. Если же оно включает кавычки или обратную косую черту, элементу присваивается текстовая строка «NULL». Кроме того, для обратной совместимости с версиями PostgreSQL до 8.2, параметр конфигурации array_nulls можно выключить (присвоив ему `off`), чтобы строки `NULL` не воспринимались как значения NULL.

Как было показано ранее, записывая значение массива, любой его элемент можно заключить в кавычки. Это *нужно* делать, если при разборе значения массива без кавычек возможна неоднозначность. Например, в кавычки необходимо заключать элементы, содержащие фигурные скобки, запятую (или разделитель, определённый для данного типа), кавычки, обратную косую черту, а также пробельные символы в начале или конце строки. Пустые строки и строки, содержащие одно слово `NULL`, также нужно заключать в кавычки. Чтобы включить кавычки или обратную косую черту в значение, заключённое в кавычки, добавьте обратную косую черту перед таким символом. С другой стороны, чтобы обойтись без кавычек, таким экранированием можно защитить все символы в данных, которые могут быть восприняты как часть синтаксиса массива.

Перед открывающей и после закрывающей скобки можно добавлять пробельные символы. Пробелы также могут окружать каждую отдельную строку значения. Во всех случаях такие пробельные символы игнорируются. Однако все пробелы в строках, заключённых в кавычки, или окружённые не пробельными символами, напротив, учитываются.

Подсказка

Записывать значения массивов в командах SQL часто бывает удобнее с помощью конструктора `ARRAY` (см. Подраздел 4.2.12). В `ARRAY` отдельные значения элементов записываются так же, как если бы они не были членами массива.

8.16. Составные типы

Составной тип представляет структуру табличной строки или записи; по сути это просто список имён полей и соответствующих типов данных. Postgres Pro позволяет использовать составные типы во многом так же, как и простые типы. Например, в определении таблицы можно объявить столбец составного типа.

8.16.1. Объявление составных типов

Ниже приведены два простых примера определения составных типов:

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

Синтаксис очень похож на `CREATE TABLE`, за исключением того, что он допускает только названия полей и их типы, какие-либо ограничения (такие как `NOT NULL`) в настоящее время не поддерживаются. Заметьте, что ключевое слово `AS` здесь имеет значение; без него система будет считать, что подразумевается другой тип команды `CREATE TYPE`, и выдаст неожиданную синтаксическую ошибку.

Определив такие типы, мы можем использовать их в таблицах:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

или функциях:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

Всякий раз, когда создаётся таблица, вместе с ней автоматически создаётся составной тип. Этот тип представляет тип строки таблицы, и его именем становится имя таблицы. Например, при выполнении команды:

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

в качестве побочного эффекта будет создан составной тип `inventory_item`, в точности соответствующий тому, что был показан выше, и использовать его можно так же. Однако заметьте, что в текущей реализации есть один недостаток: так как с составным типом не могут быть связаны ограничения, то описанные в определении таблицы ограничения *не применяются* к значениям составного типа вне таблицы. (Чтобы обойти этот недостаток, создайте *домен* поверх составного типа и добавьте желаемые ограничения в виде ограничений `CHECK` для данного домена.)

8.16.2. Конструирование составных значений

Чтобы записать значение составного типа в виде текстовой константы, его поля нужно заключить в круглые скобки и разделить их запятыми. Значение любого поля можно заключить в кавычки, а если оно содержит запятые или скобки, это делать обязательно. (Подробнее об этом говорится ниже.) Таким образом, в общем виде константа составного типа записывается так:

```
'( значение1 , значение2 , ... )'
```

Например, эта запись:

```
'("fuzzy dice",42,1.99)'
```

будет допустимой для описанного выше типа `inventory_item`. Чтобы присвоить NULL одному из полей, в соответствующем месте в списке нужно оставить пустое место. Например, эта константа задаёт значение NULL для третьего поля:

```
'("fuzzy dice",42,)'
```

Если же вместо NULL требуется вставить пустую строку, нужно записать пару кавычек:

```
'("",42,)'
```

Здесь в первом поле окажется пустая строка, а в третьем — NULL.

(Такого рода константы массивов на самом деле представляют собой всего лишь частный случай констант, описанных в Подразделе 4.1.2.7. Константа изначально воспринимается как строка и передаётся процедуре преобразования составного типа. При этом может потребоваться явно указать тип, к которому будет приведена константа.)

Значения составных типов также можно конструировать, используя синтаксис выражения `ROW`. В большинстве случаев это значительно проще, чем записывать значения в строке, так как при этом не нужно беспокоиться о вложенности кавычек. Мы уже обсуждали этот метод ранее:

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Ключевое слово ROW на самом деле может быть необязательным, если в выражении определяются несколько полей, так что эту запись можно упростить до:

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

Синтаксис выражения `ROW` более подробно рассматривается в Подразделе 4.2.13.

8.16.3. Обращение к составным типам

Чтобы обратиться к полю столбца составного типа, после имени столбца нужно добавить точку и имя поля, подобно тому, как указывается столбец после имени таблицы. На самом деле, эти обращения неотличимы, так что часто бывает необходимо использовать скобки, чтобы команда была разобрана правильно. Например, можно попытаться выбрать поле столбца из тестовой таблицы `on_hand` таким образом:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

Но это не будет работать, так как согласно правилам SQL имя `item` здесь воспринимается как имя таблицы, а не столбца в таблице `on_hand`. Поэтому этот запрос нужно переписать так:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

либо указать также и имя таблицы (например, в запросе с многими таблицами), примерно так:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

В результате объект в скобках будет правильно интерпретирован как ссылка на столбец `item`, из которого выбирается поле.

При выборке поля из значения составного типа также возможны подобные синтаксические казусы. Например, чтобы выбрать одно поле из результата функции, возвращающей составное значение, потребуется написать что-то подобное:

```
SELECT (my_func(...)).field FROM ...
```

Без дополнительных скобок в этом запросе произойдёт синтаксическая ошибка.

Специальное имя поля `*` означает «все поля»; подробнее об этом рассказывается в Подразделе 8.16.5.

8.16.4. Изменение составных типов

Ниже приведены примеры правильных команд добавления и изменения значений составных столбцов. Первые команды иллюстрируют добавление или изменение всего столбца:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

В первом примере опущено ключевое слово `ROW`, а во втором оно есть; присутствовать или отсутствовать оно может в обоих случаях.

Мы можем изменить также отдельное поле составного столбца:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

Заметьте, что при этом не нужно (и на самом деле даже нельзя) заключать в скобки имя столбца, следующее сразу за предложением `SET`, но в ссылке на тот же столбец в выражении, находящемся по правую сторону знака равенства, скобки обязательны.

И мы также можем указать поля в качестве цели команды `INSERT`:

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

Если при этом мы не укажем значения для всех полей столбца, оставшиеся поля будут заполнены значениями NULL.

8.16.5. Использование составных типов в запросах

С составными типами в запросах связаны особые правила синтаксиса и поведение. Эти правила образуют полезные конструкции, но они могут быть неочевидными, если не понимать стоящую за ними логику.

В Postgres Pro ссылка на имя таблицы (или её псевдоним) в запросе по сути является ссылкой на составное значение текущей строки в этой таблице. Например, имея таблицу `inventory_item`, показанную выше, мы можем написать:

```
SELECT c FROM inventory_item c;
```

Этот запрос выдаёт один столбец с составным значением, и его результат может быть таким:

```
           c
------------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

Заметьте, однако, что простые имена сопоставляются сначала с именами столбцов, и только потом с именами таблиц, так что такой результат получается только потому, что в таблицах запроса не оказалось столбца с именем `c`.

Обычную запись полного имени столбца вида *`имя_таблицы`*`.`*`имя_столбца`* можно понимать как применение выбора поля к составному значению текущей строки таблицы. (Из соображений эффективности на самом деле это реализовано по-другому.)

Когда мы пишем

```
SELECT c.* FROM inventory_item c;
```

то, согласно стандарту SQL, мы должны получить содержимое таблицы, развёрнутое в отдельные столбцы:

```
    name    | supplier_id | price
------------+-------------+-------
 fuzzy dice |          42 |  1.99
(1 row)
```

как с запросом

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

Postgres Pro применяет такое развёртывание для любых выражений с составными значениями, хотя как показано выше, необходимо заключить в скобки значение, к которому применяется `.*`, если только это не простое имя таблицы. Например, если `myfunc()` — функция, возвращающая составной тип со столбцами `a`, `b` и `c`, то эти два запроса выдадут одинаковый результат:

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

Подсказка

Postgres Pro осуществляет развёртывание столбцов фактически переводя первую форму во вторую. Таким образом, в данном примере `myfunc()` будет вызываться три раза для каждой строки и с одним, и с другим синтаксисом. Если это дорогостоящая функция, и вы хотите избежать лишних вызовов, можно использовать такой запрос:

```
SELECT m.* FROM some_table, LATERAL myfunc(x) AS m;
```

> Размещение вызова функции в элементе `FROM LATERAL` гарантирует, что она будет вызываться для строки не более одного раза. Конструкция `m.*` так же разворачивается в `m.a, m.b, m.c`, но теперь эти переменные просто ссылаются на выходные значения `FROM`. (Ключевое слово `LATERAL` здесь является необязательным, но мы добавили его, чтобы подчеркнуть, что функция получает `x` из `some_table`.)

Запись *`составное_значение`*`.*` приводит к такому развёртыванию столбцов, когда она фигурирует на верхнем уровне выходного списка `SELECT`, в списке `RETURNING` команд `INSERT`/`UPDATE`/`DELETE`/`MERGE`, в предложении `VALUES` или в конструкторе строки. Во всех других контекстах (включая вложенные в одну из этих конструкций), добавление `.*` к составному значению не меняет это значение, так как это воспринимается как «все столбцы» и поэтому выдаётся то же составное значение. Например, если функция `somefunc()` принимает в качестве аргумента составное значение, эти запросы равносильны:

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

В обоих случаях текущая строка таблицы `inventory_item` передаётся функции как один аргумент с составным значением. И хотя дополнение `.*` в этих случаях не играет роли, использовать его считается хорошим стилем, так как это ясно указывает на использование составного значения. В частности анализатор запроса воспримет `c` в записи `c.*` как ссылку на имя или псевдоним таблицы, а не имя столбца, что избавляет от неоднозначности; тогда как без `.*` неясно, означает ли `c` имя таблицы или имя столбца, и на самом деле при наличии столбца с именем `c` будет выбрано второе прочтение.

Эту концепцию демонстрирует и следующий пример, все запросы в котором действуют одинаково:

```
SELECT * FROM inventory_item c ORDER BY c;
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

Все эти предложения `ORDER BY` обращаются к составному значению строки, вследствие чего строки сортируются по правилам, описанным в Подразделе 9.25.6. Однако если в `inventory_item` содержится столбец с именем `c`, первый запрос будет отличаться от других, так как в нём выполнится сортировка только по данному столбцу. С показанными выше именами столбцов предыдущим запросам также равнозначны следующие:

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);
```

(В последнем случае используется конструктор строки, в котором опущено ключевое слово `ROW`.)

Другая особенность синтаксиса, связанная с составными значениями, состоит в том, что мы можем использовать *функциональную запись* для извлечения поля составного значения. Это легко можно объяснить тем, что записи *`поле`*`(`*`таблица`*`)` и *`таблица`*`.`*`поле`* взаимозаменяемы. Например, следующие запросы равнозначны:

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

Более того, если у нас есть функция, принимающая один аргумент составного типа, мы можем вызвать её в любой записи. Все эти запросы равносильны:

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

Эта равнозначность записи с полем и функциональной записи позволяет использовать с составными типами функции, реализующие «вычисляемые поля». При этом приложению, использующему последний из предыдущих запросов, не нужно знать, что фактически `somefunc` — не настоящий столбец таблицы.

Подсказка

Учитывая такое поведение, будет неразумно давать функции, принимающей один аргумент составного типа, то же имя, что и одному из полей данного составного типа. В случае неоднозначности прочтение имени поля будет выбрано при использовании синтаксиса обращения к полю, а прочтение имени функции — если используется синтаксис вызова функции. Однако в Postgres Pro до 11 версии всегда выбиралось прочтение имени поля, если только синтаксис вызова не подталкивал к прочтению имени функции. Чтобы принудительно выбрать прочтение имени функции, в предыдущих версиях надо было дополнить это имя схемой, то есть написать `*схема*.*функция*(*составное_значение*)`.

8.16.6. Синтаксис вводимых и выводимых значений составного типа

Внешнее текстовое представление составного значения состоит из записи элементов, интерпретируемых по правилам ввода-вывода для соответствующих типов полей, и оформления структуры составного типа. Оформление состоит из круглых скобок (`(` и `)`) окружающих всё значение, и запятых (`,`) между его элементами. Пробельные символы вне скобок игнорируются, но внутри они считаются частью соответствующего элемента и могут учитываться или не учитываться в зависимости от правил преобразования вводимых данных для типа этого элемента. Например, в записи:

```
'(  42)'
```

пробелы будут игнорироваться, если соответствующее поле имеет целочисленный тип, но не текстовый.

Как было показано ранее, записывая составное значение, любой его элемент можно заключить в кавычки. Это *нужно* делать, если при разборе этого значения без кавычек возможна неоднозначность. Например, в кавычки нужно заключать элементы, содержащие скобки, кавычки, запятую или обратную косую черту. Чтобы включить в поле составного значения, заключённое в кавычки, такие символы, как кавычки или обратная косая черта, перед ними нужно добавить обратную косую черту. (Кроме того, продублированные кавычки в значении поля, заключённого в кавычки, воспринимаются как одинарные, подобно апострофам в строках SQL.) С другой стороны, можно обойтись без кавычек, защитив все символы в данных, которые могут быть восприняты как часть синтаксиса составного значения, с помощью спецпоследовательностей.

Значение NULL в этой записи представляется пустым местом (когда между запятыми или скобками нет никаких символов). Чтобы ввести именно пустую строку, а не NULL, нужно написать `""`.

Функция вывода составного значения заключает значения полей в кавычки, если они представляют собой пустые строки, либо содержат скобки, запятые, кавычки или обратную косую черту, либо состоят из одних пробелов. (В последнем случае можно обойтись без кавычек, но они добавляются для удобочитаемости.) Кавычки и обратная косая черта, заключённые в значения полей, при выводе дублируются.

Примечание

Помните, что написанная SQL-команда прежде всего интерпретируется как текстовая строка, а затем как составное значение. Вследствие этого число символов обратной косой черты удваивается (если используются спецпоследовательности). Например, чтобы ввести в поле составного столбца значение типа `text` с обратной косой чертой и кавычками, команду нужно будет записать так:

```
INSERT ... VALUES ('("\"\\")');
```

Сначала обработчик спецпоследовательностей удаляет один уровень обратной косой черты, так что анализатор составного значения получает на вход `("\"\\")`. В свою очередь, он передаёт эту строку процедуре ввода значения типа `text`, где она преобразуются в `"\`. (Если бы мы работали с типом данных, процедура ввода которого также интерпретирует обратную косую черту особым образом, например `bytea`, нам могло бы понадобиться уже восемь таких символов, чтобы сохранить этот символ в поле составного значения.) Во избежание такого дублирования спецсимволов строки можно заключать в доллары (см. Подраздел 4.1.2.4).

Подсказка

Записывать составные значения в командах SQL часто бывает удобнее с помощью конструктора `ROW`. В `ROW` отдельные значения элементов записываются так же, как если бы они не были членами составного выражения.

8.17. Диапазонные типы

Диапазонные типы представляют диапазоны значений некоторого типа данных (он также называется *подтипом* диапазона). Например, диапазон типа `timestamp` может представлять временной интервал, когда зарезервирован зал заседаний. В данном случае типом данных будет `tsrange` (сокращение от «timestamp range»), а подтипом — `timestamp`. Подтип должен быть полностью упорядочиваемым, чтобы можно было однозначно определить, где находится значение по отношению к диапазону: внутри, до или после него.

Диапазонные типы полезны тем, что позволяют представить множество возможных значений в одной структуре данных и чётко выразить такие понятия, как пересечение диапазонов. Наиболее очевидный вариант их использования — применять диапазоны даты и времени для составления расписания, но также полезными могут оказаться диапазоны цен, интервалы измерений и т. д.

Для каждого диапазонного типа есть соответствующий мультидиапазонный тип. Мультидиапазон представляет собой упорядоченный список несмежных, непустых и отличных от NULL диапазонов. Большинство диапазонных операторов работают и с мультидиапазонами. Кроме того, есть функции для работы именно с мультидиапазонными типами.

8.17.1. Встроенные диапазонные и мультидиапазонные типы

Postgres Pro имеет следующие встроенные диапазонные типы:

- `int4range` — диапазон подтипа `integer`, `int4multirange` — соответствующий мультидиапазон
- `int8range` — диапазон подтипа `bigint`, `int8multirange` — соответствующий мультидиапазон
- `numrange` — диапазон подтипа `numeric`, `nummultirange` — соответствующий мультидиапазон
- `tsrange` — диапазон подтипа `timestamp without time zone`, `tsmultirange` — соответствующий мультидиапазон
- `tstzrange` — диапазон подтипа `timestamp with time zone`, `tstzmultirange` — соответствующий мультидиапазон
- `daterange` — диапазон подтипа `date`, `datemultirange` — соответствующий мультидиапазон

Помимо этого, вы можете определять собственные типы; подробнее это описано в CREATE TYPE.

8.17.2. Примеры

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Вхождение
SELECT int4range(10, 20) @> 3;
```

```
-- Перекрытие
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Получение верхней границы
SELECT upper(int8range(15, 25));

-- Вычисление пересечения
SELECT int4range(10, 20) * int4range(15, 25);

-- Является ли диапазон пустым?
SELECT isempty(numrange(1, 5));
```

Полный список операторов и функций, предназначенных для диапазонных типов, приведён в Таблице 9.56 и Таблице 9.58.

8.17.3. Включение и исключение границ

Любой непустой диапазон имеет две границы, верхнюю и нижнюю, и включает все точки между этими значениями. В него также может входить точка, лежащая на границе, если диапазон включает эту границу. И наоборот, если диапазон не включает границу, считается, что точка, лежащая на этой границе, в него не входит.

В текстовой записи диапазона включение нижней границы обозначается символом «`[`», а исключением — символом «`(`». Для верхней границы включение обозначается аналогично, символом «`]`», а исключение — символом «`)`». (Подробнее это описано в Подразделе 8.17.5.)

Для проверки, включается ли нижняя или верхняя граница в диапазон, предназначены функции `lower_inc` и `upper_inc`, соответственно.

8.17.4. Неограниченные (бесконечные) диапазоны

Нижнюю границу диапазона можно опустить и определить тем самым диапазон, включающий все значения, лежащие ниже верхней границы, например: `(,3]`. Подобным образом, если не определить верхнюю границу, в диапазон войдут все значения, лежащие выше нижней границы. Если же опущена и нижняя, и верхняя границы, такой диапазон будет включать все возможные значения своего подтипа. Указание отсутствующей границы как включаемой в диапазон автоматически преобразуется в исключающее; например, `[,]` преобразуется в `(,)`. Можно воспринимать отсутствующие значения как плюс/минус бесконечность, но всё же это особые значения диапазонного типа, которые охватывают и возможные для подтипа значения плюс/минус бесконечность.

Для подтипов, в которых есть понятие «бесконечность», `infinity` может использоваться в качестве явного значения границы. При этом, например, в диапазон `[today,infinity)` с подтипом `timestamp` не будет входить специальное значение `infinity` данного подтипа, однако это значение будет входить в диапазон `[today,infinity]`, как и в диапазоны `[today,)` и `[today,]`.

Проверить, определена ли верхняя или нижняя граница, можно с помощью функций `lower_inf` и `upper_inf`, соответственно.

8.17.5. Ввод-вывод диапазонов

Вводимое значение диапазона должно записываться в одной из следующих форм:

```
(нижняя-граница,верхняя-граница)
(нижняя-граница,верхняя-граница]
[нижняя-граница,верхняя-граница)
[нижняя-граница,верхняя-граница]
empty
```

Тип скобок (квадратные или круглые) определяет, включаются ли в диапазон соответствующие границы, как описано выше. Заметьте, что последняя форма содержит только слово `empty` и определяет пустой диапазон (диапазон, не содержащий точек).

Здесь *нижняя-граница* может быть строкой с допустимым значением подтипа или быть пустой (тогда диапазон будет без нижней границы). Аналогично, *верхняя-граница* может задаваться одним из значений подтипа или быть неопределённой (пустой).

Любое значение границы диапазона можно заключить в кавычки ("). А если значение содержит круглые или квадратные скобки, запятые, кавычки или обратную косую черту, использовать кавычки необходимо, чтобы эти символы не рассматривались как часть синтаксиса диапазона. Чтобы включить в значение границы диапазона, заключённое в кавычки, такие символы, как кавычки или обратная косая черта, перед ними нужно добавить обратную косую черту. (Кроме того, продублированные кавычки в значении диапазона, заключённого в кавычки, воспринимаются как одинарные, подобно апострофам в строках SQL.) С другой стороны, можно обойтись без кавычек, защитив все символы в данных, которые могут быть восприняты как часть синтаксиса диапазона, с помощью спецпоследовательностей. Чтобы задать в качестве границы пустую строку, нужно ввести `""`, так как пустая строка без кавычек будет означать отсутствие границы.

Пробельные символы до и после определения диапазона игнорируются, но когда они присутствуют внутри скобок, они воспринимаются как часть значения верхней или нижней границы. (Хотя они могут также игнорироваться в зависимости от подтипа диапазона.)

Примечание

Эти правила очень похожи на правила записи значений для полей составных типов. Дополнительные замечания приведены в Подразделе 8.16.6.

Примеры:

```
-- в диапазон включается 3, не включается 7 и включаются все точки между ними
SELECT '[3,7)'::int4range;

-- в диапазон не включаются 3 и 7, но включаются все точки между ними
SELECT '(3,7)'::int4range;

-- в диапазон включается только одно значение 4
SELECT '[4,4]'::int4range;

-- диапазон не включает никаких точек (нормализация заменит его определение на 'empty')
SELECT '[4,4)'::int4range;
```

Вводимое значение мультидиапазона заключается в фигурные скобки (`{` и `}`) и содержит ноль или более диапазонов, разделённых запятыми. До и после скобок и запятых допускаются пробельные символы. Синтаксис сделан похожим на синтаксис массивов, но мультидиапазоны намного проще: они имеют только одну размерность, а их содержимое не нужно заключать в кавычки. (Как показано выше, границы диапазонов могут заключаться в кавычки.)

Примеры:

```
SELECT '{}'::int4multirange;
SELECT '{[3,7)}'::int4multirange;
SELECT '{[3,7), [8,9)}'::int4multirange;
```

8.17.6. Конструирование диапазонов и мультидиапазонов

Для каждого диапазонного типа определена функция конструктора, имеющая то же имя, что и данный тип. Использовать этот конструктор обычно удобнее, чем записывать текстовую константу диапазона, так как это избавляет от потребности в дополнительных кавычках. Функция конструктора может принимать два или три параметра. Вариант с двумя параметрами создаёт диапазон в стандартной форме (нижняя граница включается, верхняя исключается), тогда как для варианта с тремя параметрами включение границ определяется третьим параметром. Третий параметр должен содержать одну из строк: «`()`», «`(]`», «`[)`» или «`[]`». Например:

```
-- Полная форма: нижняя граница, верхняя граница и текстовая строка, определяющая
-- включение/исключение границ.
SELECT numrange(1.0, 14.0, '(]');

-- Если третий аргумент опущен, подразумевается '[)'.
SELECT numrange(1.0, 14.0);

-- Хотя здесь указывается '(]', при выводе значение будет приведено к
-- каноническому виду, так как int8range — тип дискретного диапазона (см. ниже).
SELECT int8range(1, 14, '(]');

-- Когда вместо любой границы указывается NULL, соответствующей границы у диапазона не
 будет.
SELECT numrange(NULL, 2.2);
```

Для каждого диапазонного типа также есть конструктор мультидиапазона с тем же именем, что и у мультидиапазонного типа. Функция-конструктор принимает ноль или более аргументов, представляющих собой диапазоны соответствующего типа. Например:

```
SELECT nummultirange();
SELECT nummultirange(numrange(1.0, 14.0));
SELECT nummultirange(numrange(1.0, 14.0), numrange(20.0, 25.0));
```

8.17.7. Типы дискретных диапазонов

Дискретным диапазоном считается диапазон, для подтипа которого однозначно определён «шаг», как например для типов `integer` и `date`. Значения этих двух типов можно назвать соседними, когда между ними нет никаких других значений. В непрерывных диапазонах, напротив, всегда (или почти всегда) можно найти ещё одно значение между двумя данными. Например, непрерывным диапазоном будет диапазон с подтипами `numeric` и `timestamp`. (Хотя `timestamp` имеет ограниченную точность, то есть теоретически он является дискретным, но всё же лучше считать его непрерывным, так как шаг его обычно не определён.)

Можно также считать дискретным подтип диапазона, в котором чётко определены понятия «следующего» и «предыдущего» элемента для каждого значения. Такие определения позволяют преобразовывать границы диапазона из включаемых в исключаемые, выбирая следующий или предыдущий элемент вместо заданного значения. Например, диапазоны целочисленного типа `[4,8]` и `(3,9)` описывают одно и то же множество значений; но для диапазона подтипа numeric это не так.

Для типа дискретного диапазона определяется функция *канонизации*, учитывающая размер шага для данного подтипа. Задача этой функции — преобразовать равнозначные диапазоны к единственному представлению, в частности нормализовать включаемые и исключаемые границы. Если функция канонизации не определена, диапазоны с различным определением будут всегда считаться разными, даже когда они на самом деле представляют одно множество значений.

Для встроенных типов `int4range`, `int8range` и `daterange` каноническое представление включает нижнюю границу и не включает верхнюю; то есть диапазон приводится к виду `[)`. Однако для нестандартных типов можно использовать и другие соглашения.

8.17.8. Определение новых диапазонных типов

Пользователи могут определять собственные диапазонные типы. Это может быть полезно, когда нужно использовать диапазоны с подтипами, для которых нет встроенных диапазонных типов. Например, можно определить новый тип диапазона для подтипа `float8`:

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);
```

```
SELECT '[1.234, 5.678]'::floatrange;
```

Так как для `float8` осмысленное значение «шага» не определено, функция канонизации в данном примере не задаётся.

Когда вы определяете собственный диапазон, система автоматически создаёт мультидиапазонный тип.

Определяя собственный диапазонный тип, вы также можете выбрать другие правила сортировки или класс оператора B-дерева для его подтипа, что позволит изменить порядок значений, от которого зависит, какие значения попадают в заданный диапазон.

Если подтип можно рассматривать как дискретный, а не непрерывный, в команде `CREATE TYPE` следует также задать функцию `канонизации`. Этой функции будет передаваться значение диапазона, а она должна вернуть равнозначное значение, но, возможно, с другими границами и форматированием. Для двух диапазонов, представляющих одно множество значений, например, целочисленные диапазоны `[1, 7]` и `[1, 8)`, функция канонизации должна выдавать один результат. Какое именно представление будет считаться каноническим, не имеет значения — главное, чтобы два равнозначных диапазона, отформатированных по-разному, всегда преобразовывались в одно значение с одинаковым форматированием. Помимо исправления формата включаемых/исключаемых границ, функция канонизации может округлять значения границ, если размер шага превышает точность хранения подтипа. Например, в типе диапазона для подтипа `timestamp` можно определить размер шага, равный часу, тогда функция канонизации должна будет округлить границы, заданные, например с точностью до минут, либо вместо этого выдать ошибку.

Помимо этого, для любого диапазонного типа, ориентированного на использование с индексами GiST или SP-GiST, должна быть определена разница значений подтипов, функция `subtype_diff`. (Индекс сможет работать и без `subtype_diff`, но в большинстве случаев это будет не так эффективно.) Эта функция принимает на вход два значения подтипа и возвращает их разницу (т. е. *X* минус *Y*) в значении типа `float8`. В показанном выше примере может использоваться функция `float8mi`, определяющая нижележащую реализацию обычного оператора «минус» для типа `float8`, но для другого подтипа могут потребоваться дополнительные преобразования. Иногда для представления разницы в числовом виде требуется ещё и творческий подход. Функция `subtype_diff`, насколько это возможно, должна быть согласована с порядком сортировки, вытекающим из выбранных правил сортировки и класса оператора; то есть, её результат должен быть положительным, если согласно порядку сортировки первый её аргумент больше второго.

Ещё один, не столь тривиальный пример функции `subtype_diff`:

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

Дополнительные сведения о создании диапазонных типов можно найти в описании CREATE TYPE.

8.17.9. Индексация

Для столбцов, имеющих диапазонный тип, можно создать индексы GiST и SP-GiST. Также индексы GiST можно создавать для столбцов мультидиапазонных типов. Например, так создаётся индекс GiST:

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

Индекс GiST или SP-GiST для диапазонов помогает ускорить запросы со следующими операторами: `=`, `&&`, `<@`, `@>`, `<<`, `>>`, `-|-`, `&<` и `&>` (дополнительно о них можно узнать в Таблице 9.56. Индекс GiST для мультидиапазонов может ускорить запросы, задействующие один набор мультидиапазонных операторов. Индекс GiST для диапазонов и индекс GiST для мультидиапазонов могут также соответственно ускорить запросы, задействующие следующие межтиповые операторы диапазон-мультидиапазон и мультидиапазон-диапазон: `&&`, `<@`, `@>`, `<<`, `>>`, `-|-`, `&<` и `&>`. Дополнительные сведения можно найти в Таблице 9.56.

Кроме того, для таких столбцов можно создать индексы на основе хеша и B-деревьев. Для индексов таких типов полезен по сути только один оператор диапазона — равно. Порядок сортировки B-дерева определяется для значений диапазона соответствующими операторами < и >, но этот порядок может быть произвольным и он не очень важен в реальном мире. Поддержка B-деревьев и хешей диапазонными типами нужна в основном для сортировки и хеширования при выполнении запросов, но не для создания самих индексов.

8.17.10. Ограничения для диапазонов

Тогда как для скалярных значений естественным ограничением является `UNIQUE`, оно обычно не подходит для диапазонных типов. Вместо этого чаще оказываются полезнее ограничения-исключения (см. CREATE TABLE ... CONSTRAINT ... EXCLUDE). Такие ограничения позволяют, например определить условие «непересечения» диапазонов. Например:

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

Это ограничение не позволит одновременно сохранить в таблице несколько диапазонов, которые накладываются друг на друга:

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ОШИБКА:  конфликтующее значение ключа нарушает ограничение-исключение
 "reservation_during_excl"
ПОДРОБНОСТИ:  Ключ (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00"))
конфликтует с существующим ключом (during)=(["2010-01-01 11:30:00","2010-01-01
 15:00:00")).
```

Для максимальной гибкости в ограничении-исключении можно сочетать простые скалярные типы данных с диапазонами, используя расширение `btree_gist`. Например, если `btree_gist` установлено, следующее ограничение не будет допускать пересекающиеся диапазоны, только если совпадают также и номера комнат:

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ОШИБКА:  конфликтующее значение ключа нарушает ограничение-исключение
 "room_reservation_room_during_excl"
```

```
ПОДРОБНОСТИ:  Ключ (room, during)=(123A, [ 2010-01-01 14:30:00, 2010-01-01 15:30:00 ))
 конфликтует
с существующим ключом (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01
 15:00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

8.18. Типы доменов

Домен — пользовательский тип данных, основанный на другом *нижележащем типе*. Он может быть определён с условиями, ограничивающими множество допустимых значений подмножеством значений нижележащего типа. В остальном он ведёт себя как нижележащий тип — например, с доменными типом будут работать любые операторы или функции, применимые к нижележащему типу. Нижележащим типом может быть любой встроенный или пользовательский базовый тип, тип-перечисление, массив, составной тип, диапазон или другой домен.

Например, мы можем создать домен поверх целых чисел, принимающий только положительные числа:

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);   -- работает
INSERT INTO mytable VALUES(-1);  -- ошибка
```

Когда к значению доменного типа применяются операторы или функции, предназначенные для нижележащего типа, домен автоматически приводится к нижележащему типу. Так, например, результат операции `mytable.id - 1` будет считаться имеющим тип `integer`, а не `posint`. Мы могли бы записать `(mytable.id - 1)::posint`, чтобы снова привести результат к типу `posint`, что повлечёт перепроверку ограничений домена. В этом случае, если данное выражение будет применено к `id`, равному 1, произойдёт ошибка. Значение нижележащего типа можно присвоить полю или переменной доменного типа, не записывая приведение явно, но и в этом случае ограничения домена будут проверяться.

За дополнительными сведениями обратитесь к описанию CREATE DOMAIN.

8.19. Идентификаторы объектов

Идентификатор объекта (Object Identifier, OID) используется внутри Postgres Pro в качестве первичного ключа различных системных таблиц. Идентификатор объекта представляется в типе `oid`. Также существуют различные типы-псевдонимы для `oid`, с именами `reg`*`сущность`*. Обзор этих типов приведён в Таблице 8.26.

В настоящее время тип `oid` реализован как четырёхбайтное целое число без знака. Таким образом, значение этого типа может быть недостаточно большим для обеспечения уникальности в базе данных или даже в отдельных больших таблицах.

Для самого типа `oid` помимо сравнения определены всего несколько операторов. Однако его можно привести к целому и затем задействовать в обычных целочисленных вычислениях. (При этом следует опасаться путаницы со знаковыми/беззнаковыми значениями.)

Типы-псевдонимы OID сами по себе не вводят новых операций и отличаются только специализированными функциями ввода-вывода. Эти функции могут принимать и выводить не просто числовые значения, как тип `oid`, а символические имена системных объектов. Эти типы позволяют упростить поиск объектов по значениям OID. Например, чтобы выбрать из `pg_attribute` строки, относящиеся к таблице `mytable`, можно написать:

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

вместо:

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

Хотя второй вариант выглядит не таким уж плохим, но это лишь очень простой запрос. Если же потребуется выбрать правильный OID, когда таблица `mytable` есть в нескольких схемах, вложенный подзапрос будет гораздо сложнее. Преобразователь вводимого значения типа `regclass` находит таблицу согласно заданному пути поиска схем, так что он делает «всё правильно» автоматически. Аналогично, приведя идентификатор таблицы к типу `regclass`, можно получить символическое представление числового кода.

Таблица 8.26. Идентификаторы объектов

Имя	Ссылки	Описание	Пример значения
`oid`	any	числовой идентификатор объекта	`564182`
`regclass`	`pg_class`	имя отношения	`pg_type`
`regcollation`	`pg_collation`	имя правила сортировки	`"POSIX"`
`regconfig`	`pg_ts_config`	конфигурация текстового поиска	`english`
`regdictionary`	`pg_ts_dict`	словарь текстового поиска	`simple`
`regnamespace`	`pg_namespace`	пространство имён	`pg_catalog`
`regoper`	`pg_operator`	имя оператора	`+`
`regoperator`	`pg_operator`	оператор с типами аргументов	`*(integer,integer)` или `-(NONE, integer)`
`regproc`	`pg_proc`	имя функции	`sum`
`regprocedure`	`pg_proc`	функция с типами аргументов	`sum(int4)`
`regprofile`	`pg_profile`	имя профиля	`default`
`regrole`	`pg_authid`	имя роли	`smithee`
`regtype`	`pg_type`	имя типа данных	`integer`

Все типы псевдонимов OID для объектов, сгруппированных в пространство имён, принимают имена, дополненные именем схемы, и выводят имена со схемой, если данный объект нельзя будет найти в текущем пути поиска без имени схемы. Например, `myschema.mytable` является приемлемым входным значением для `regclass` (если существует такая таблица). Это значение может выводиться как `myschema.mytable` или просто `mytable`, в зависимости от текущего пути поиска. Типы `regproc` и `regoper` принимают только уникальные вводимые имена (не перегруженные), что ограничивает их применимость; в большинстве случаев лучше использовать `regprocedure` или `regoperator`. Для типа `regoperator` в записи унарного оператора неиспользуемый операнд заменяется словом `NONE`.

Функции ввода для данных типов допускают пробелы между компонентами и приводят буквы верхнего регистра к нижнему, за исключением строки в двойных кавычках; это сделано для того, чтобы правила записи были похожи на принятые для записи имён объектов в SQL. И наоборот, функции вывода будут добавлять двойные кавычки, если это необходимо, чтобы выводимая строка была допустимым идентификатором SQL. Например, OID функции с именем `Foo` (с `F` в верхнем регистре), принимающей два целочисленных аргумента, можно ввести как `' "Foo" ( int, integer ) '::regprocedure`. Результат будет выглядеть как `"Foo"(integer,integer)`. И имя функции, и имена типов аргументов также могут быть дополнены схемой.

Многие встроенные функции Postgres Pro принимают OID таблицы или другого типа объекта БД и для удобства объявляются как принимающие `regclass` (или соответствующий тип-псевдоним OID). Это означает, что вам не нужно искать OID объекта вручную, а можно просто ввести его имя в виде строки. Например, функция `nextval (regclass)` принимает OID отношения последовательности, поэтому её можно вызвать так:

```
nextval('foo')              обращается к последовательности foo
nextval('FOO')              то же самое
nextval('"Foo"')            обращается к последовательности Foo
nextval('myschema.foo')     обращается к myschema.foo
nextval('"myschema".foo')   то же самое
nextval('foo')              ищет foo в пути поиска
```

Примечание

Когда аргумент такой функции записывается как текстовая строка в чистом виде, она становится константой типа `regclass`. Так как фактически это будет просто значение OID, оно будет привязано к изначально идентифицированной последовательности, несмотря на то, что она может быть переименована, перенесена в другую схему и т. д. Такое «раннее связывание» обычно желательно для ссылок на последовательности в значениях столбцов по умолчанию и представлениях. Но иногда возникает необходимость в «позднем связывании», когда ссылки на последовательности распознаются в процессе выполнения. Чтобы получить такое поведение, нужно принудительно изменить тип константы с `regclass` на `text`:

```
nextval('foo'::text)      foo распознаётся во время выполнения
```

Для поиска во время выполнения также может использоваться функция `to_regclass()` и подобные. См. Таблицу 9.74.

Другой практический пример использования `regclass` — поиск OID таблицы, отображённой в представлениях `information_schema`, которые не предоставляют такие OID напрямую. Например, можно вызвать функцию `pg_relation_size()`, для которой требуется OID таблицы. С учётом указанных выше правил, далее представлен корректный способ вызова данной функции

```
SELECT table_schema, table_name,
       pg_relation_size((quote_ident(table_schema) || '.' ||
                         quote_ident(table_name))::regclass)
FROM information_schema.tables
WHERE ...
```

. Функция `quote_ident()` заключит идентификатор в двойные кавычки, когда это необходимо. Более простым способом кажется

```
SELECT pg_relation_size(table_name)
FROM information_schema.tables
WHERE ...
```

но использовать его *не рекомендуется*, потому что он не сработает для таблиц, которые не входят в заданный путь поиска или имена которых нужно заключать в кавычки.

Дополнительным свойством большинства типов псевдонимов OID является образование зависимостей. Когда в сохранённом выражении фигурирует константа одного из этих типов (например, в представлении или в значении столбца по умолчанию), это создаёт зависимость от целевого объекта. Например, если значение по умолчанию определяется выражением `nextval('my_seq'::regclass)`, Postgres Pro понимает, что это выражение зависит от последовательности `my_seq`, и не позволит удалить последовательность раньше, чем будет удалено это выражение. Альтернативная запись `nextval('my_seq'::text)` не создаёт зависимость. (Исключениями являются типы `regprofile` и `regrole`. Константы этих типов в таких выражениях не допускаются.)

Есть ещё один тип системных идентификаторов, `xid`, представляющий идентификатор транзакции (сокращённо xact). Этот тип имеют системные столбцы `xmin` и `xmax`. Идентификаторы транзакций определяются 32-битными числами. В некоторых контекстах используется 64-битный вариант `xid8`. В отличие от `xid`, значения `xid8` увеличиваются строго монотонно и никогда не повторяются на протяжении всего существования кластера баз данных. За подробностями обратитесь к Разделу 64.1.

Третий тип идентификаторов, используемых в системе, — `cid`, идентификатор команды (command identifier). Этот тип данных имеют системные столбцы `cmin` и `cmax`. Идентификаторы команд — это тоже 32-битные числа.

И наконец, последний тип системных идентификаторов — `tid`, идентификатор строки/кортежа (tuple identifier). Этот тип данных имеет системный столбец `ctid`. Идентификатор кортежа представляет собой пару (из номера блока и индекса кортежа в блоке), идентифицирующую физическое расположение строки в таблице.

(Подробнее о системных столбцах рассказывается в Разделе 5.6.)

8.20. Тип `pg_lsn`

Тип данных `pg_lsn` может применяться для хранения значения LSN (последовательный номер в журнале, Log Sequence Number), которое представляет собой указатель на позицию в журнале WAL. Этот тип содержит `XLogRecPtr` и является внутренним системным типом Postgres Pro.

Технически LSN — это 64-битное целое число, представляющее байтовое смещение в потоке журнала предзаписи. Он выводится в виде двух шестнадцатеричных чисел до 8 цифр каждое, через косую черту, например: `16/B374D848`. Тип `pg_lsn` поддерживает стандартные операторы сравнения, такие как `=` и `>`. Можно также вычесть один LSN из другого с помощью оператора `-`; результатом будет число байт между этими двумя позициями в журнале предзаписи. Также число байт можно добавлять и вычитать из LSN с помощью операторов `+(pg_lsn, numeric)` и `-(pg_lsn, numeric)` соответственно. Заметьте, что вычисленный LSN должен находиться в допустимом для типа `pg_lsn` диапазоне, то есть между `0/0` и `FFFFFFFF/FFFFFFFF`.

8.21. Псевдотипы

В систему типов Postgres Pro включены несколько специальных элементов, которые в совокупности называются *псевдотипами*. Псевдотип нельзя использовать в качестве типа данных столбца, но можно объявить функцию с аргументом или результатом такого типа. Каждый из существующих псевдотипов полезен в ситуациях, когда характер функции не позволяет просто получить или вернуть определённый тип данных SQL. Все существующие псевдотипы перечислены в Таблице 8.27.

Таблица 8.27. Псевдотипы

Имя	Описание
`any`	Указывает, что функция принимает любой вводимый тип данных.
`anyelement`	Указывает, что функция принимает любой тип данных (см. Подраздел 36.2.5).
`anyarray`	Указывает, что функция принимает любой тип массива (см. Подраздел 36.2.5).
`anynonarray`	Указывает, что функция принимает любой тип данных, кроме массивов (см. Подраздел 36.2.5).
`anyenum`	Указывает, что функция принимает любое перечисление (см. Подраздел 36.2.5 и Раздел 8.7).
`anyrange`	Указывает, что функция принимает любой диапазонный тип данных (см. Подраздел 36.2.5 и Раздел 8.17).
`anymultirange`	Указывает, что функция принимает любой мультидиапазонный тип данных (см. Подраздел 36.2.5 и Раздел 8.17).

Имя	**Описание**
`anycompatible`	Указывает, что функция принимает любой тип данных и может автоматически приводить различные аргументы к общему типу данных (см. Подраздел 36.2.5).
`anycompatiblearray`	Указывает, что функция принимает любой тип массива и может автоматически приводить различные аргументы к общему типу данных (см. Подраздел 36.2.5).
`anycompatiblenonarray`	Указывает, что функция принимает любой тип, отличный от массива, и может автоматически приводить различные аргументы к общему типу данных (см. Подраздел 36.2.5).
`anycompatiblerange`	Указывает, что функция принимает любой тип диапазонный данных и может автоматически приводить различные аргументы к общему типу данных (см. Подраздел 36.2.5 и Раздел 8.17).
`anycompatiblemultirange`	Указывает, что функция принимает любой мультидиапазонный тип данных и может автоматически приводить различные аргументы к общему типу данных (см. Подраздел 36.2.5 и Раздел 8.17).
`cstring`	Указывает, что функция принимает или возвращает строку в стиле C.
`internal`	Указывает, что функция принимает или возвращает внутренний серверный тип данных.
`language_handler`	Обработчик процедурного языка объявляется как возвращающий тип `language_handler` .
`fdw_handler`	Обработчик обёртки сторонних данных объявляется как возвращающий тип `fdw_handler` .
`table_am_handler`	Обработчик табличного метода доступа объявляется как возвращающий тип `table_am_handler` .
`index_am_handler`	Обработчик метода доступа индекса объявляется как возвращающий тип `index_am_handler` .
`tsm_handler`	Обработчик метода выборки из таблицы объявляется как возвращающий тип `tsm_handler` .
`record`	Указывает, что функция принимает или возвращает неопределённый тип строки.
`trigger`	Триггерная функция объявляется как возвращающая тип `trigger`.
`event_trigger`	Функция событийного триггера объявляется как возвращающая тип `event_trigger` .
`pg_ddl_command`	Обозначает представление команд DDL, доступное событийным триггерам.
`void`	Указывает, что функция не возвращает значение.
`unknown`	Обозначает ещё не распознанный тип, например простую строковую константу.

Функции, написанные на языке C (встроенные или динамически загружаемые), могут быть объявлены с параметрами или результатами любого из этих псевдотипов. Ответственность за безопасное поведение функции с аргументами таких типов ложится на разработчика функции.

Функции, написанные на процедурных языках, могут использовать псевдотипы, только если это позволяет соответствующий язык. В настоящее время большинство процедурных языков запрещают использовать псевдотипы в качестве типа аргумента и позволяют использовать для результатов только типы `void` и `record` (и `trigger` или `event_trigger`, когда функция реализует триггер или событийный триггер). Некоторые языки также поддерживают полиморфные функции с полиморфными псевдотипами, подробно рассмотренными в Подразделе 36.2.5.

Псевдотип `internal` используется в объявлениях функций, предназначенных только для внутреннего использования в СУБД, но не для прямого вызова в запросах SQL. Если у функции есть как хотя бы один аргумент типа `internal`, её нельзя будет вызывать из SQL. Чтобы сохранить типобезопасность при таком ограничении, следуйте важному правилу: не создавайте функцию, возвращающую результат типа `internal`, если у неё нет ни одного аргумента `internal`.

Глава 9. Функции и операторы

Postgres Pro предоставляет огромное количество функций и операторов для встроенных типов данных. В этой главе описаны основные из них, тогда как некоторые специальные функции описываются в других разделах документации. Кроме того, пользователи могут определять свои функции и операторы, как описано в Части V. Просмотреть все существующие функции и операторы можно в psql с помощью команд `\df` и `\do`, соответственно.

В этой главе типы аргументов и результата функции обозначаются так:

`repeat ( text, integer )` → `text`

В данном случае она говорит, что функция `repeat` принимает один текстовый и один целочисленный аргумент и возвращает результат текстового типа. Стрелка вправо также указывает на результат в примере использования, например:

`repeat('Pg', 4)` → `PgPgPgPg`

Если для вас важна переносимость, учтите, что практически все функции и операторы, описанные в этой главе, за исключением простейших арифметических и операторов сравнения, а также явно отмеченных функций, не описаны в стандарте SQL. Тем не менее частично эта расширенная функциональность присутствует и в других СУБД SQL и во многих случаях различные реализации одинаковых функций оказываются аналогичными и совместимыми.

9.1. Логические операторы

Набор логических операторов включает обычные:

`boolean AND boolean` → `boolean`

`boolean OR boolean` → `boolean`

`NOT boolean` → `boolean`

В SQL работает логическая система с тремя состояниями: true (истина), false (ложь) и `NULL`, «неопределённое» состояние. Рассмотрите следующие таблицы истинности:

a	*b*	*a* **AND** *b*	*a* **OR** *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT** *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

Операторы `AND` и `OR` коммутативны, то есть от перемены мест операндов результат не меняется. (Однако левый операнд не обязательно будет вычисляться перед правым. Более подробно порядок вычисления подвыражений описывается в Подразделе 4.2.14.)

9.2. Функции и операторы сравнения

Набор операторов сравнения включает обычные операторы, перечисленные в Таблице 9.1.

Таблица 9.1. Операторы сравнения

Оператор	Описание
тип_данных < *тип_данных* → boolean	Меньше
тип_данных > *тип_данных* → boolean	Больше
тип_данных <= *тип_данных* → boolean	Меньше или равно
тип_данных >= *тип_данных* → boolean	Больше или равно
тип_данных = *тип_данных* → boolean	Равно
тип_данных <> *тип_данных* → boolean	Не равно
тип_данных != *тип_данных* → boolean	Не равно

Примечание

В стандарте SQL для условия «не равно» принята запись `<>`. Синонимичная ей запись `!=` преобразуется в `<>` на самой ранней стадии разбора запроса. Как следствие, реализовать операторы `!=` и `<>` так, чтобы они работали по-разному, невозможно.

Эти операторы сравнения имеются для всех встроенных типов данных, значения которых сортируются естественным образом, включая числовые, строковые типы, а также типы даты/времени. Кроме того, сравниваться могут массивы, составные типы и диапазоны, если типы данных их компонентов являются сравниваемыми.

Обычно можно сравнивать также значения связанных типов данных; например, возможно сравнение `integer` > `bigint`. Некоторые подобные операции реализуются непосредственно «межтиповыми» операторами сравнения, но если такого оператора нет, анализатор запроса попытается привести частные типы к более общим и применить подходящий для них оператор сравнения.

Как показано выше, все операторы сравнения являются бинарными и возвращают значения типа `boolean`. Таким образом, выражения вида `1 < 2 < 3` недопустимы (так как не существует оператора `<`, который бы сравнивал булево значение с `3`). Для проверки нахождения значения в интервале, воспользуйтесь предикатом `BETWEEN`, описанным ниже.

Существует также несколько предикатов сравнения; они приведены в Таблице 9.2. Они работают подобно операторам, но имеют особый синтаксис, установленный стандартом SQL.

Таблица 9.2. Предикаты сравнения

Предикат Описание Пример(ы)
тип_данных BETWEEN *тип_данных* AND *тип_данных* → boolean Между (включая границы интервала). 2 BETWEEN 1 AND 3 → t 2 BETWEEN 3 AND 1 → f
тип_данных NOT BETWEEN *тип_данных* AND *тип_данных* → boolean Не между (обратное к `BETWEEN`). 2 NOT BETWEEN 1 AND 3 → f
тип_данных BETWEEN SYMMETRIC *тип_данных* AND *тип_данных* → boolean Между, после сортировки граничных значений. 2 BETWEEN SYMMETRIC 3 AND 1 → t
тип_данных NOT BETWEEN SYMMETRIC *тип_данных* AND *тип_данных* → boolean

Предикат **Описание** **Пример(ы)**
Не между, после сортировки граничных значений. `2 NOT BETWEEN SYMMETRIC 3 AND 1` → `f`
тип_данных `IS DISTINCT FROM` *тип_данных* → `boolean` Не равно, при этом NULL воспринимается как обычное значение. `1 IS DISTINCT FROM NULL` → `t` (а не `NULL`) `NULL IS DISTINCT FROM NULL` → `f` (а не `NULL`)
тип_данных `IS NOT DISTINCT FROM` *тип_данных* → `boolean` Равно, при этом NULL воспринимается как обычное значение. `1 IS NOT DISTINCT FROM NULL` → `f` (а не `NULL`) `NULL IS NOT DISTINCT FROM NULL` → `t` (а не `NULL`)
тип_данных `IS NULL` → `boolean` Проверяет, является ли значение эквивалентным NULL. `1.5 IS NULL` → `f`
тип_данных `IS NOT NULL` → `boolean` Проверяет, отличается ли значение от NULL. `'null' IS NOT NULL` → `t`
тип_данных `ISNULL` → `boolean` Проверяет, является ли значение эквивалентным NULL (нестандартный синтаксис).
тип_данных `NOTNULL` → `boolean` Проверяет, отличается ли значение от NULL (нестандартный синтаксис).

`boolean IS TRUE` → `boolean`
Проверяет, является ли результат логического выражения значением true.
`true IS TRUE` → `t`
`NULL::boolean IS TRUE` → `f` (а не `NULL`)

`boolean IS NOT TRUE` → `boolean`
Проверяет, является ли результат логического выражения значением false или неизвестным.
`true IS NOT TRUE` → `f`
`NULL::boolean IS NOT TRUE` → `t` (а не `NULL`)

`boolean IS FALSE` → `boolean`
Проверяет, является ли результат логического выражения значением false.
`true IS FALSE` → `f`
`NULL::boolean IS FALSE` → `f` (а не `NULL`)

`boolean IS NOT FALSE` → `boolean`
Проверяет, является ли результат логического выражения значением true или неизвестным.
`true IS NOT FALSE` → `t`
`NULL::boolean IS NOT FALSE` → `t` (а не `NULL`)

`boolean IS UNKNOWN` → `boolean`
Проверяет, является ли результат логического выражения неизвестным значением.
`true IS UNKNOWN` → `f`
`NULL::boolean IS UNKNOWN` → `t` (а не `NULL`)

Предикат **Описание** **Пример(ы)**

`boolean IS NOT UNKNOWN` → `boolean`
Проверяет, является ли результат логического выражения значением true или false.
`true IS NOT UNKNOWN` → `t`
`NULL::boolean IS NOT UNKNOWN` → `f` (а не `NULL`)

Предикат `BETWEEN` упрощает проверки интервала:

```
a BETWEEN x AND y
```

равнозначно

```
a >= x AND a <= y
```

Заметьте, что `BETWEEN` считает, что границы интервала включаются в интервал. Предикат `BETWEEN SYMMETRIC` аналогичен `BETWEEN`, за исключением того, что аргумент слева от `AND` не обязательно должен быть меньше или равен аргументу справа. Если это не так, аргументы автоматически меняются местами, так что всегда подразумевается непустой интервал.

Различные варианты `BETWEEN` реализуются посредством обычных операторов сравнения, и поэтому они будут работать с любыми типами данных, которые можно сравнивать.

> ### Примечание
>
> Использование `AND` в конструкции `BETWEEN` создаёт неоднозначность с использованием `AND` в качестве логического оператора. Для её устранения в качестве второго аргумента предложения `BETWEEN` принимается только ограниченный набор типов выражений. Если вам нужно записать более сложное подвыражение в `BETWEEN`, заключите это подвыражение в скобки.

Обычные операторы сравнения выдают NULL (что означает «неопределённость»), а не true или false, когда любое из сравниваемых значений NULL. Например, `7 = NULL` выдаёт NULL, так же, как и `7 <> NULL`. Когда это поведение нежелательно, можно использовать предикаты `IS [ NOT ] DISTINCT FROM`:

```
a IS DISTINCT FROM b
a IS NOT DISTINCT FROM b
```

Для значений не NULL условие `IS DISTINCT FROM` работает так же, как оператор `<>`. Однако если оба сравниваемых значения NULL, результат будет false, и только если одно из значений NULL, возвращается true. Аналогично, условие `IS NOT DISTINCT FROM` равносильно `=` для значений не NULL, но возвращает true, если оба сравниваемых значения NULL, и false в противном случае. Таким образом, эти предикаты по сути работают с NULL, как с обычным значением, а не с «неопределённостью».

Для проверки, содержит ли значение NULL или нет, используются предикаты:

```
выражение IS NULL
выражение IS NOT NULL
```

или равнозначные (но нестандартные) предикаты:

```
выражение ISNULL
выражение NOTNULL
```

Заметьте, что проверка `выражение = NULL` *не* будет работать, так как `NULL` считается не «равным» `NULL`. (Значение NULL представляет неопределённость, и равны ли две неопределённости, тоже не определено.)

Подсказка

Некоторые приложения могут ожидать, что *`выражение`* `= NULL` вернёт true, если результатом *`выражения`* является NULL. Такие приложения настоятельно рекомендуется исправить и привести в соответствие со стандартом SQL. Однако в случаях, когда это невозможно, это поведение можно изменить с помощью параметра конфигурации transform_null_equals. Когда этот параметр включён, Postgres Pro преобразует условие `x = NULL` в `x IS NULL`.

Если *`выражение`* возвращает табличную строку, тогда `IS NULL` будет истинным, когда само выражение — NULL или все поля строки — NULL, а `IS NOT NULL` будет истинным, когда само выражение не NULL, и все поля строки так же не NULL. Вследствие такого определения, `IS NULL` и `IS NOT NULL` не всегда будут возвращать взаимодополняющие результаты для таких выражений; в частности такие выражения со строками, одни поля которых NULL, а другие не NULL, будут ложными одновременно. Например:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- выбор всех строк, содержащих только NULL

SELECT ROW(table.*) IS NOT NULL FROM table;  -- выбор всех строк, не содержащих NULL

SELECT NOT(ROW(table.*) IS NOT NULL) FROM TABLE;  -- выбор всех строк, содержащих хотя
 бы одно значение NULL
```

В некоторых случаях имеет смысл написать *`строка`* `IS DISTINCT FROM NULL` или *`строка`* `IS NOT DISTINCT FROM NULL`, чтобы просто проверить, равно ли NULL всё значение строки, без каких-либо дополнительных проверок полей строки.

Логические значения можно также проверить с помощью предикатов

```
логическое_выражение IS TRUE
логическое_выражение IS NOT TRUE
логическое_выражение IS FALSE
логическое_выражение IS NOT FALSE
логическое_выражение IS UNKNOWN
логическое_выражение IS NOT UNKNOWN
```

Они всегда возвращают true или false и никогда NULL, даже если какой-любо операнд — NULL. Они интерпретируют значение NULL как «неопределённость». Заметьте, что `IS UNKNOWN` и `IS NOT UNKNOWN` по сути равнозначны `IS NULL` и `IS NOT NULL`, соответственно, за исключением того, что выражение может быть только булевого типа.

Также имеется несколько связанных со сравнениями функций; они перечислены в Таблице 9.3.

Таблица 9.3. Функции сравнения

Функция Описание Пример(ы)

`num_nonnulls ( VARIADIC "any" )` → `integer`
Возвращает число аргументов, отличных от NULL.
`num_nonnulls(1, NULL, 2)` → `2`

`num_nulls ( VARIADIC "any" )` → `integer`
Возвращает число аргументов NULL.
`num_nulls(1, NULL, 2)` → `1`

9.3. Математические функции и операторы

Математические операторы определены для множества типов Postgres Pro. Как работают эти операции с типами, для которых нет стандартных соглашений о математических действиях (например, с типами даты/времени), мы опишем в последующих разделах.

В Таблице 9.4 показаны математические операторы, реализованные для стандартных числовых типов. Если не отмечено обратное, операторы, принимающие *`числовой_тип`*, работают с типами `smallint`, `integer`, `bigint`, `numeric`, `real` и `double precision`. Операторы, принимающие *`целочисленный_тип`*, работают с типами `smallint`, `integer` и `bigint`. За исключением явно отмеченных случаев, все разновидности операторов возвращают такой же тип данных, который имеет их аргумент (или их аргументы). Вызовы с неодинаковыми типами аргументов, например `integer + numeric`, разрешаются в пользу типа, оказавшегося последним в определённых выше списках.

Таблица 9.4. Математические операторы

Оператор	Описание	Пример(ы)
`числовой_тип` `+` *`числовой_тип`* → *`числовой_тип`*	Сложение	`2 + 3` → `5`
`+` *`числовой_тип`* → *`числовой_тип`*	Унарный плюс (нет операции)	`+ 3.5` → `3.5`
`числовой_тип` `-` *`числовой_тип`* → *`числовой_тип`*	Вычитание	`2 - 3` → `-1`
`-` *`числовой_тип`* → *`числовой_тип`*	Смена знака	`- (-4)` → `4`
`числовой_тип` `*` *`числовой_тип`* → *`числовой_тип`*	Умножение	`2 * 3` → `6`

`числовой_тип` `/` *`числовой_тип`* → *`числовой_тип`*	Деление (при делении целочисленных типов результат округляется в направлении нуля)	`5.0 / 2` → `2.5000000000000000`
		`5 / 2` → `2`
		`(-5) / 2` → `-2`

`числовой_тип` `%` *`числовой_тип`* → *`числовой_тип`*	Остаток от деления; имеется для типов `smallint`, `integer`, `bigint` и `numeric`	`5 % 4` → `1`

`numeric ^ numeric` → `numeric`
`double precision ^ double precision` → `double precision`
	Возведение в степень	`2 ^ 3` → `8`
		В отличие от принятого в математике порядка, `^` по умолчанию вычисляется слева направо
		`2 ^ 3 ^ 3` → `512`
		`2 ^ (3 ^ 3)` → `134217728`

Оператор Описание Пример(ы)

`|/ double precision` → `double precision`
Квадратный корень
`|/ 25.0` → `5`

`||/ double precision` → `double precision`
Кубический корень
`||/ 64.0` → `4`

`@` *числовой_тип* → *числовой_тип* Абсолютное значение `@ -5.0` → `5.0`
целочисленный_тип `&` *целочисленный_тип* → *целочисленный_тип* Битовое И `91 & 15` → `11`
целочисленный_тип `|` *целочисленный_тип* → *целочисленный_тип* Битовое ИЛИ `32 | 3` → `35`
целочисленный_тип `#` *целочисленный_тип* → *целочисленный_тип* Битовое исключающее ИЛИ `17 # 5` → `20`
`~` *целочисленный_тип* → *целочисленный_тип* Битовое НЕ `~1` → `-2`
целочисленный_тип `<< integer` → *целочисленный_тип* Битовый сдвиг влево `1 << 4` → `16`
целочисленный_тип `>> integer` → *целочисленный_тип* Битовый сдвиг вправо `8 >> 2` → `2`

В Таблице 9.5 перечислены все существующие математические функции. Многие из этих функций имеют несколько форм с разными типами аргументов. За исключением случаев, где это указано явно, все разновидности функций возвращают тот же тип данных, который имеет их аргумент (или аргументы); вызовы с неодинаковыми типами разрешаются по тому же принципу, что был описан выше для операторов. Функции, работающие с данными `double precision`, в массе своей используют реализации из системных библиотек сервера, поэтому точность и поведение в граничных случаях может зависеть от системы сервера.

Таблица 9.5. Математические функции

Функция Описание Пример(ы)
`abs` (*числовой_тип*) → *числовой_тип* Абсолютное значение `abs(-17.4)` → `17.4`
`cbrt` (`double precision`) → `double precision` Кубический корень

Функция **Описание** **Пример(ы)**
`cbrt(64.0)` → `4`

`ceil` (`numeric`) → `numeric`
`ceil` (`double precision`) → `double precision`
Ближайшее целое, большее или равное аргументу
`ceil(42.2)` → `43`
`ceil(-42.8)` → `-42`

`ceiling` (`numeric`) → `numeric`
`ceiling` (`double precision`) → `double precision`
Ближайшее целое, большее или равное аргументу (равнозначно `ceil`)
`ceiling(95.3)` → `96`

`degrees` (`double precision`) → `double precision`
Преобразование радианов в градусы
`degrees(0.5)` → `28.64788975654116`

`div` (*y* `numeric`, *x* `numeric`) → `numeric`
Целочисленный результат *y*/*x* (округлённый в направлении нуля)
`div(9, 4)` → `2`

`erf` (`double precision`) → `double precision`
Функция ошибок
`erf(1.0)` → `0.8427007929497149`

`erfc` (`double precision`) → `double precision`
Дополнительная функция ошибок (`1 - erf(x)` , без потери точности при передаче больших входных данных)
`erfc(1.0)` → `0.15729920705028513`

`exp` (`numeric`) → `numeric`
`exp` (`double precision`) → `double precision`
Экспонента (`e` возводится в заданную степень)
`exp(1.0)` → `2.7182818284590452`

`factorial` (`bigint`) → `numeric`
Факториал
`factorial(5)` → `120`

`floor` (`numeric`) → `numeric`
`floor` (`double precision`) → `double precision`
Ближайшее целое, меньшее или равное аргументу
`floor(42.8)` → `42`
`floor(-42.8)` → `-43`

`gcd` (*числовой_тип* , *числовой_тип*) → *числовой_тип* Наибольший общий делитель (наибольшее положительное целое, на которое оба аргумента делятся без остатка); `0`, если оба аргумента нулевые; имеется для типов `integer`, `bigint` и `numeric` `gcd(1071, 462)` → `21`
`lcm` (*числовой_тип* , *числовой_тип*) → *числовой_тип*

Функция **Описание** **Пример(ы)**
Наименьшее общее кратное (наименьшее строго положительное число, которое делится нацело на оба аргумента); 0, если оба аргумента нулевые; имеется для типов integer, bigint и numeric lcm(1071, 462) → 23562
ln (numeric) → numeric ln (double precision) → double precision Натуральный логарифм ln(2.0) → 0.6931471805599453
log (numeric) → numeric log (double precision) → double precision Логарифм по основанию 10 log(100) → 2
log10 (numeric) → numeric log10 (double precision) → double precision Логарифм по основанию 10 (то же, что и log) log10(1000) → 3
log (*b* numeric, *x* numeric) → numeric Логарифм *x* по основанию *b* log(2.0, 64.0) → 6.0000000000000000
min_scale (numeric) → integer Минимальный масштаб (число цифр в дробной части), необходимый для точного представления заданного значения min_scale(8.4100) → 2
mod (*y числовой_тип* , *x числовой_тип*) → *числовой_тип* Остаток от деления *y*/*x*; имеется для типов smallint, integer, bigint и numeric mod(9, 4) → 1
pi ( ) → double precision Приближённое значение π pi() → 3.141592653589793
power (*a* numeric, *b* numeric) → numeric power (*a* double precision, *b* double precision) → double precision *a* возводится в степень *b* power(9, 3) → 729
radians (double precision) → double precision Преобразование градусов в радианы radians(45.0) → 0.7853981633974483
round (numeric) → numeric round (double precision) → double precision Округляет до ближайшего целого числа. Для numeric половина (.5) округляется до одного по модулю. Для double precision результат округления половины зависит от платформы (обычно выполняется «округление до ближайшего чётного»). round(42.4) → 42
round (*v* numeric, *s* integer) → numeric

Функция **Описание** **Пример(ы)**

Округление *v* до *s* десятичных знаков. Половина (.5) округляется до 1 по модулю.
`round(42.4382, 2)` → `42.44`
`round(1234.56, -1)` → `1230`

`scale` (`numeric`) → `integer`
Масштаб аргумента (число десятичных цифр в дробной части)
`scale(8.4100)` → `4`

`sign` (`numeric`) → `numeric`
`sign` (`double precision`) → `double precision`
Знак аргумента (-1, 0 или +1)
`sign(-8.4)` → `-1`

`sqrt` (`numeric`) → `numeric`
`sqrt` (`double precision`) → `double precision`
Квадратный корень
`sqrt(2)` → `1.4142135623730951`

`trim_scale` (`numeric`) → `numeric`
Сокращает масштаб значения (число цифр в дробной части), убирая конечные нули
`trim_scale(8.4100)` → `8.41`

`trunc` (`numeric`) → `numeric`
`trunc` (`double precision`) → `double precision`
Усечение до целого (в направлении нуля)
`trunc(42.8)` → `42`
`trunc(-42.8)` → `-42`

`trunc` (*v* `numeric`, *s* `integer`) → `numeric`
Усечение *v* до *s* десятичных знаков
`trunc(42.4382, 2)` → `42.43`

`width_bucket` (*operand* `numeric`, *low* `numeric`, *high* `numeric`, *count* `integer`) → `integer`
`width_bucket` (*operand* `double precision`, *low* `double precision`, *high* `double precision`, *count* `integer`) → `integer`
Возвращает номер группы, в которую попадёт *operand* в гистограмме с числом групп *count* равного размера, в диапазоне от *low* до *high*. Группы определяются нижними границами (включая значения на границе) и верхними границами (без значений на границе). Результатом будет `0`, если операнд лежит левее значения *low*, и *count*+1, если значение больше значения *high* или равно ему. При сценарии *low* > *high* ситуация будет обратная, и группа `1` теперь будет ниже значения *low*, и включаться будут пограничные значения начиная с верхних.
`width_bucket(5.35, 0.024, 10.06, 5)` → `3`
`width_bucket(9, 10, 0, 10)` → `2`

`width_bucket` (*operand* `anycompatible`, *thresholds* `anycompatiblearray`) → `integer` Возвращает номер группы, в которую попадёт *operand* (группы определяются нижними границами, включающими пограничные значения, передаваемыми в *thresholds*). Результатом будет 0, если операнд оказывается левее нижней границы. Аргумент *operand* и элементы массива могут быть любого типа, для которого определены стандартные операторы сравнения. Массив *thresholds* *должен быть отсортирован* по возрастанию, иначе будут получены неожиданные результаты.

Функция **Описание** **Пример(ы)**
`width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])` → `2`

В Таблице 9.6 перечислены все функции для генерации случайных чисел.

Таблица 9.6. Случайные функции

Функция **Описание** **Пример(ы)**

`random` ( ) → `double precision`
Возвращает случайное число в диапазоне 0.0 <= x < 1.0
`random()` → `0.897124072839091`

`random` (*min* `integer`, *max* `integer`) → `integer`
`random` (*min* `bigint`, *max* `bigint`) → `bigint`
`random` (*min* `numeric`, *max* `numeric`) → `numeric`
Возвращает случайное число в диапазоне *min* <= x <= *max*. Для типа `numeric` результат будет иметь столько же цифр в дробной части, что и *min* или *max*, в зависимости от того, у какого из них больше цифр в дробной части.
`random(1, 10)` → `7`
`random(-0.499, 0.499)` → `0.347`

`random_normal` ([*mean* `double precision` [, *stddev* `double precision`]]) → `double precision`
Возвращает случайное значение нормального распределения с заданными параметрами. По умолчанию для *mean* установлено значение 0.0, а для *stddev* — 1.0.
`random_normal(0.0, 1.0)` → `0.051285419`

`setseed` (`double precision`) → `void`
Задаёт отправную точку для последующих вызовов `random()` и `random_normal()` ; аргументом может быть число от -1.0 до 1.0, включая границы
`setseed(0.12345)`

Функции `random()` и `random_normal()`, указанные в таблице Таблица 9.6, используют детерминированный генератор псевдослучайных чисел. Он работает быстро, но не подходит для криптографических приложений; более безопасная альтернатива имеется в модуле pgcrypto. Если воспользоваться функцией `setseed()` и вызывать её с одним и тем же аргументом, в текущем сеансе можно получать повторяющиеся последовательности результатов `random()` или `random_normal()`. Если функция `setseed()` до этого не вызывалась в текущем сеансе, при первом вызове одной из этих функций затравка будет получена из платформозависимого источника случайных чисел.

В Таблице 9.7 перечислены все имеющиеся тригонометрические функции. У каждой функции имеются две вариации: одна измеряет углы в радианах, а вторая — в градусах.

Таблица 9.7. Тригонометрические функции

Функция **Описание** **Пример(ы)**

`acos` (`double precision`) → `double precision`
Арккосинус в радианах
`acos(1)` → `0`

`acosd` (`double precision`) → `double precision` Арккосинус в градусах

Функция **Описание** **Пример(ы)**
`acosd(0.5)` → `60`

`asin` (`double precision`) → `double precision`
Арксинус в радианах
`asin(1)` → `1.5707963267948966`

`asind` (`double precision`) → `double precision`
Арксинус в градусах
`asind(0.5)` → `30`

`atan` (`double precision`) → `double precision`
Арктангенс в радианах
`atan(1)` → `0.7853981633974483`

`atand` (`double precision`) → `double precision`
Арктангенс в градусах
`atand(1)` → `45`

`atan2` (*y* `double precision`, *x* `double precision`) → `double precision`
Арктангенс отношения *y*/*x* в радианах
`atan2(1, 0)` → `1.5707963267948966`

`atan2d` (*y* `double precision`, *x* `double precision`) → `double precision`
Арктангенс отношения *y*/*x* в градусах
`atan2d(1, 0)` → `90`

`cos` (`double precision`) → `double precision`
Косинус угла в радианах
`cos(0)` → `1`

`cosd` (`double precision`) → `double precision`
Косинус угла в градусах
`cosd(60)` → `0.5`

`cot` (`double precision`) → `double precision`
Котангенс угла в радианах
`cot(0.5)` → `1.830487721712452`

`cotd` (`double precision`) → `double precision`
Котангенс угла в градусах
`cotd(45)` → `1`

`sin` (`double precision`) → `double precision`
Синус угла в радианах
`sin(1)` → `0.8414709848078965`

`sind` (`double precision`) → `double precision`
Синус угла в градусах
`sind(30)` → `0.5`

`tan` (`double precision`) → `double precision`
Тангенс угла в радианах
`tan(1)` → `1.5574077246549023`

`tand` (`double precision`) → `double precision` Тангенс угла в градусах

Функция **Описание** **Пример(ы)**
`tand(45)` → `1`

> ### Примечание
>
> Также можно работать с углами в градусах, применяя вышеупомянутые функции преобразования единиц `radians()` и `degrees()`. Однако предпочтительнее использовать тригонометрические функции с градусами, так как это позволяет избежать ошибок округления в особых случаях, например, при вычислении `sind(30)`.

В Таблице 9.8 перечислены все имеющиеся гиперболические операторы.

Таблица 9.8. Гиперболические функции

Функция **Описание** **Пример(ы)**

`sinh ( double precision ) → double precision`
Гиперболический синус
`sinh(1)` → `1.1752011936438014`

`cosh ( double precision ) → double precision`
Гиперболический косинус
`cosh(0)` → `1`

`tanh ( double precision ) → double precision`
Гиперболический тангенс
`tanh(1)` → `0.7615941559557649`

`asinh ( double precision ) → double precision`
Обратный гиперболический синус
`asinh(1)` → `0.881373587019543`

`acosh ( double precision ) → double precision`
Обратный гиперболический косинус
`acosh(1)` → `0`

`atanh ( double precision ) → double precision`
Обратный гиперболический тангенс
`atanh(0.5)` → `0.5493061443340548`

9.4. Строковые функции и операторы

В этом разделе описаны функции и операторы для работы с текстовыми строками. Под строками в данном контексте подразумеваются значения типов `character`, `character varying` и `text`. Если не отмечено обратное, эти функции и операторы принимают и возвращают тип `text`. С тем же успехом в аргументах может передаваться тип `character varying`. Аргументы же типа `character` до вызова оператора или функции приводятся к типу `text`, вследствие чего завершающие пробелы в значении `character` будут обрезаться.

В SQL определены несколько строковых функций, в которых аргументы разделяются не запятыми, а ключевыми словами. Они перечислены в Таблице 9.9. Postgres Pro также предоставляет варианты этих функций с синтаксисом, обычным для функций (см. Таблицу 9.10).

Примечание

Оператор конкатенации строк (`||`) сможет принять нестроковый аргумент, если хотя бы один аргумент имеет строковый тип, как показано в Таблице 9.9. В других случаях можно использовать явное преобразование в `text`, чтобы оператор принял нестроковый аргумент.

Таблица 9.9. Строковые функции и операторы языка SQL

Функция/оператор **Описание** **Пример(ы)**

`text || text → text`
Соединяет две строки.
`'Post' || 'greSQL' → PostgreSQL`

`text || anynonarray → text`
`anynonarray || text → text`
Преобразует нестроковый аргумент в текст, а затем соединяет две строки. (Нестроковый аргумент не должен быть массивом, иначе возникает неоднозначность с операторами массивов `||`. Если вы хотите соединить строку с текстовой формой массива, явно приведите его к типу `text`.)
`'Value: ' || 42 → Value: 42`

`btrim (` *`string`* `text [,` *`characters`* `text ] ) → text`
Удаляет наибольшую подстроку, содержащую только символы *`characters`* (по умолчанию пробел), с начала и с конца строки *`string`*.
`btrim('xyxtrimyyx', 'xyz') → trim`

`text IS [NOT] [`*`form`*`] NORMALIZED → boolean`
Проверяет, соответствует ли строка определённой форме нормализации Юникода. Форма указывается в необязательном ключевом слове *`form`*: `NFC` (по умолчанию), `NFD`, `NFKC` или `NFKD`. Это выражение можно использовать, только если кодировка сервера — `UTF8`. Заметьте, что проверить нормализацию с помощью этого выражения, как правило, будет быстрее, чем нормализовать уже, возможно, нормализованные строки.
`U&'\0061\0308bc' IS NFD NORMALIZED → t`

`bit_length ( text ) → integer`
Возвращает число бит в строке (это число в 8 раз больше `octet_length`).
`bit_length('jose') → 32`

`char_length ( text ) → integer`
`character_length ( text ) → integer`
Возвращает число символов в строке.
`char_length('josé') → 4`

`lower ( text ) → text`
Переводит символы строки в нижний регистр в соответствии с правилами локали базы данных.
`lower('TOM') → tom`

`lpad (` *`string`* `text,` *`length`* `integer [,` *`fill`* `text ] ) → text`
Дополняет строку *`string`* слева до длины *`length`* символами *`fill`* (по умолчанию пробелами). Если длина строки уже больше заданной, она обрезается справа.
`lpad('hi', 5, 'xy') → xyxhi`

`ltrim (` *`string`* `text [,` *`characters`* `text ] ) → text`

Функция/оператор **Описание** **Пример(ы)**
Удаляет наибольшую подстроку, содержащую только символы *`characters`* (по умолчанию пробелы), с начала строки *`string`*. `ltrim('zzzytest', 'xyz')` → `test`

`normalize` (`text` [, *`form`*]) → `text`
Переводит строку в заданную форму нормализации Unicode. Форма указывается в необязательном ключевом слове *`form`*: `NFC` (по умолчанию), `NFD`, `NFKC` или `NFKD`. Эту функцию можно использовать, только если кодировка сервера — `UTF8`.
`normalize(U&'\0061\0308bc', NFC)` → `U&'\00E4bc'`

`octet_length` (`text`) → `integer` Возвращает число байт в строке. `octet_length('josé')` → `5` (если серверная кодировка — UTF8)

`octet_length` (`character`) → `integer`
Возвращает число байт в строке. Так как эта вариация функции принимает непосредственно тип `character`, завершающие строку пробелы не обрезаются.
`octet_length('abc '::character(4))` → `4`

`overlay` (*`string`* `text` `PLACING` *`newsubstring`* `text` `FROM` *`start`* `integer` [`FOR` *`count`* `integer`]) → `text`
Заменяет подстроку в *`string`*, начиная с символа с номером *`start`*, длиной *`count`* символов, на подстроку *`newsubstring`*. В отсутствие параметра *`count`* количество заменяемых символов определяется длиной *`newsubstring`*.
`overlay('Txxxxas' placing 'hom' from 2 for 4)` → `Thomas`

`position` (*`substring`* `text` `IN` *`string`* `text`) → `integer`
Возвращает начальную позицию первого вхождения *`substring`* в строке *`string`* либо 0, если такого вхождения нет.
`position('om' in 'Thomas')` → `3`

`rpad` (*`string`* `text`, *`length`* `integer` [, *`fill`* `text`]) → `text`
Дополняет строку *`string`* справа до длины *`length`* символами *`fill`* (по умолчанию пробелами). Если длина строки уже больше заданной, она обрезается.
`rpad('hi', 5, 'xy')` → `hixyx`

`rtrim` (*`string`* `text` [, *`characters`* `text`]) → `text`
Удаляет наибольшую подстроку, содержащую только символы *`characters`* (по умолчанию пробелы), с конца строки *`string`*.
`rtrim('testxxzx', 'xyz')` → `test`

`substring` (*`string`* `text` [`FROM` *`start`* `integer`] [`FOR` *`count`* `integer`]) → `text`
Извлекает из *`string`* подстроку, начиная с позиции *`start`* (если она указана), длиной до *`count`* символов (если она указана). Параметры *`start`* и *`count`* могут опускаться, но не оба сразу.
`substring('Thomas' from 2 for 3)` → `hom`
`substring('Thomas' from 3)` → `omas`
`substring('Thomas' for 2)` → `Th`

`substring` (*`string`* `text` `FROM` *`pattern`* `text`) → `text`
Извлекает первую подстроку, соответствующую регулярному выражению в стиле POSIX; см. Подраздел 9.7.3.
`substring('Thomas' from '...$')` → `mas`

Функция/оператор **Описание** **Пример(ы)**

`substring` (*`string`* `text` `SIMILAR` *`pattern`* `text` `ESCAPE` *`escape`* `text`) → `text`
`substring` (*`string`* `text` `FROM` *`pattern`* `text` `FOR` *`escape`* `text`) → `text`
Извлекает первую подстроку, соответствующую регулярному выражению в стиле SQL; см. Подраздел 9.7.2. Первая форма определена в стандарте, начиная с SQL:2003, а вторая существовала только в стандарте SQL:1999 и её следует считать устаревшей.
`substring('Thomas' similar '%#"o_a#"_' escape '#')` → `oma`

`trim` ([`LEADING` | `TRAILING` | `BOTH`] [*`characters`* `text`] `FROM` *`string`* `text`) → `text`
Удаляет наибольшую подстроку, содержащую только символы *`characters`* (по умолчанию пробелы), с начала, с конца или с обеих сторон (`BOTH`, по умолчанию) строки *`string`*.
`trim(both 'xyz' from 'yxTomxx')` → `Tom`

`trim` ([`LEADING` | `TRAILING` | `BOTH`] [`FROM`] *`string`* `text` [, *`characters`* `text`]) → `text`
Это нестандартный синтаксис вызова `trim()`.
`trim(both from 'yxTomxx', 'xyz')` → `Tom`

`unicode_assigned` (`text`) → `boolean` Возвращает `true`, если всем символам в строке присвоены коды Unicode, в противном случае — `false`. Эту функцию можно использовать, только если кодировка сервера — `UTF8`.

`upper` (`text`) → `text`
Переводит символы строки в верхний регистр, в соответствии с правилами локали базы данных.
`upper('tom')` → `TOM`

Кроме этого, в PostgreSQL есть и другие функции для работы со строками, перечисленные в Таблице 9.10. (Некоторые из них используются в качестве внутренней реализации стандартных строковых функций SQL, приведённых в Таблице 9.9.) Также имеются операторы сопоставления шаблонов, описанные в Разделе 9.7, и операторы полнотекстового поиска, описанные в Главе 12.

Таблица 9.10. Другие строковые функции и операторы

Функция/оператор **Описание** **Пример(ы)**

`text` `^@` `text` → `boolean`
Возвращает true, если первая строка начинается со второй строки (равнозначно функции `starts_with()`).
`'alphabet' ^@ 'alph'` → `t`

`ascii` (`text`) → `integer`
Возвращает числовой код первого символа аргумента. Для UTF8 возвращает код символа в Unicode. Для других многобайтных кодировок аргумент должен быть ASCII-символом.
`ascii('x')` → `120`

`chr` (`integer`) → `text`
Возвращает символ с данным кодом. Для UTF8 аргумент воспринимается как код символа Unicode, а для других кодировок он должен указывать на ASCII-символ. Эта функция не может выдать `chr(0)`, так как данный символ нельзя сохранить в текстовых типах данных.
`chr(65)` → `A`

`concat` (*`val1`* `"any"` [, *`val2`* `"any"` [, ...]]) → `text`

Функция/оператор **Описание** **Пример(ы)**
Соединяет текстовые представления всех аргументов, игнорируя NULL. `concat('abcde', 2, NULL, 22)` → `abcde222`

`concat_ws` (*sep* `text`, *val1* `"any"` [, *val2* `"any"` [, ...]]) → `text`
Соединяет вместе все аргументы, кроме первого, через разделитель. Разделитель задаётся в первом аргументе и должен быть отличен от NULL. В других аргументах значение NULL игнорируется.
`concat_ws(',', 'abcde', 2, NULL, 22)` → `abcde,2,22`

`format` (*formatstr* `text` [, *formatarg* `"any"` [, ...]]) → `text`
Форматирует аргументы в соответствии со строкой формата; см. Подраздел 9.4.1. Эта функция работает подобно `sprintf` в языке C.
`format('Hello %s, %1$s', 'World')` → `Hello World, World`

`initcap` (`text`) → `text`
Переводит первую букву каждого слова в строке в верхний регистр, а остальные — в нижний. Словами считаются последовательности алфавитно-цифровых символов, разделённые любыми другими символами.
`initcap('hi THOMAS')` → `Hi Thomas`

`left` (*string* `text`, *n* `integer`) → `text`
Возвращает первые *n* символов в строке. Когда *n* меньше нуля, возвращаются все символы слева, кроме последних |*n*|.
`left('abcde', 2)` → `ab`

`length` (`text`) → `integer`
Возвращает число символов в строке.
`length('jose')` → `4`

`md5` (`text`) → `text`
Вычисляет MD5-хеш аргумента и выдаёт результат в шестнадцатеричном виде.
`md5('abc')` → `900150983cd24fb0d6963f7d28e17f72`

`parse_ident` (*qualified_identifier* `text` [, *strict_mode* `boolean` `DEFAULT` `true`]) → `text[]`
Раскладывает полный идентификатор, задаваемый параметром *qualified_identifier* , на массив идентификаторов, удаляя кавычки, обрамляющие отдельные идентификаторы. По умолчанию лишние символы после последнего идентификатора вызывают ошибку, но если отключить строгий режим (передать во втором параметре `false`), такие символы игнорируются. (Это поведение полезно для разбора имён таких объектов, как функции.) Заметьте, что эта функция не усекает чрезмерно длинные идентификаторы. Если вы хотите получить усечённые имена, можно привести результат к `name[]`.
`parse_ident('"SomeSchema".someTable')` → `{SomeSchema,sometable}`

`pg_client_encoding` ( ) → `name`
Возвращает имя текущей клиентской кодировки.
`pg_client_encoding()` → `UTF8`

`quote_ident` (`text`) → `text`
Преобразует аргумент в строку, подходящую для использования в качестве идентификатора в SQL-операторе. При необходимости идентификатор заключается в кавычки (например, если он содержит символы, недопустимые в открытом виде, или буквы в разных регистрах). Если переданная строка содержит кавычки, они дублируются. См. также Пример 41.1.
`quote_ident('Foo bar')` → `"Foo bar"`

Функция/оператор **Описание** **Пример(ы)**
quote_literal (text) → text Преобразует аргумент в строку, подходящую для использования в качестве текстовой константы в SQL-операторе. Внутренние символы апостроф и обратная косая черта при этом дублируются. Заметьте, что quote_literal возвращает NULL, когда на вход ей передаётся строка NULL; если же нужно получить представление и такого аргумента, лучше использовать quote_nullable. См. также Пример 41.1. quote_literal(E'O\'Reilly') → 'O''Reilly'
quote_literal (anyelement) → text Переводит данное значение в текстовый вид и заключает в апострофы как текстовую строку. Символы апостроф и обратная косая черта при этом дублируются. quote_literal(42.5) → '42.5'
quote_nullable (text) → text Преобразует аргумент в строку, подходящую для использования в качестве текстовой константы в SQL-операторе; при этом для аргумента NULL возвращается строка NULL. Символы апостроф и обратная косая черта дублируются должным образом. См. также Пример 41.1. quote_nullable(NULL) → NULL
quote_nullable (anyelement) → text Переводит данное значение в текстовый вид и заключает в апострофы как текстовую строку, при этом для аргумента NULL возвращается строка NULL. Символы апостроф и обратная косая черта дублируются должным образом. quote_nullable(42.5) → '42.5'
regexp_count (*string* text, *pattern* text [, *start* integer [, *flags* text]]) → integer Возвращает количество вхождений регулярного выражения POSIX *pattern* в строке *string*; см. Подраздел 9.7.3. regexp_count('123456789012', '\d\d\d', 2) → 3
regexp_instr (*string* text, *pattern* text [, *start* integer [, *N* integer [, *endoption* integer [, *flags* text [, *subexpr* integer]]]]]) → integer Возвращает позицию в строке *string* *N*-го вхождения шаблона *pattern*, содержащего регулярное выражение POSIX, или ноль, если такого вхождения нет; см. Подраздел 9.7.3. regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i') → 3 regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i', 2) → 5
regexp_like (*string* text, *pattern* text [, *flags* text]) → boolean Проверяет, есть ли в строке *string* вхождение регулярного выражения POSIX *pattern*; см. Подраздел 9.7.3. regexp_like('Hello World', 'world$', 'i') → t
regexp_match (*string* text, *pattern* text [, *flags* text]) → text[] Возвращает подстроки из первого вхождения регулярного выражения POSIX *pattern* в строке *string*; см. Подраздел 9.7.3. regexp_match('foobarbequebaz', '(bar)(beque)') → {bar,beque}
regexp_matches (*string* text, *pattern* text [, *flags* text]) → setof text[] Возвращает подстроки из первого вхождения регулярного выражения POSIX *pattern* в строке *string* или подстроки из всех таких вхождений, если используется флаг g; см. Подраздел 9.7.3. regexp_matches('foobarbequebaz', 'ba.', 'g') →

Функция/оператор **Описание** **Пример(ы)**
`{bar}` `{baz}`

`regexp_replace` (*string* `text`, *pattern* `text`, *replacement* `text` [, *start* `integer`] [, *flags* `text`]) → `text`
Заменяет подстроку первого вхождения регулярного выражения POSIX *pattern* или подстроки всех таких вхождений, если использован флаг *g*; см. Подраздел 9.7.3.
`regexp_replace('Thomas', '.[mN]a.', 'M')` → `ThM`

`regexp_replace` (*string* `text`, *pattern* `text`, *replacement* `text`, *start* `integer`, *N* `integer` [, *flags* `text`]) → `text`
Заменяет подстроку *N*-го вхождения регулярного выражения POSIX *pattern* в строке *string* или подстроки всех таких вхождений, если *N* равен нулю; см. Подраздел 9.7.3.
`regexp_replace('Thomas', '.', 'X', 3, 2)` → `ThoXas`

`regexp_split_to_array` (*string* `text`, *pattern* `text` [, *flags* `text`]) → `text[]`
Разделяет содержимое *string* на элементы, используя в качестве разделителя регулярное выражение POSIX, и выдаёт массив результатов; см. Подраздел 9.7.3.
`regexp_split_to_array('hello world', '\s+')` → `{hello,world}`

`regexp_split_to_table` (*string* `text`, *pattern* `text` [, *flags* `text`]) → `setof text`
Разделяет содержимое *string* на элементы, используя в качестве разделителя регулярное выражение POSIX, и выдаёт набор результатов; см. Подраздел 9.7.3.
`regexp_split_to_table('hello world', '\s+')` →
`hello`
`world`

`regexp_substr` (*string* `text`, *pattern* `text` [, *start* `integer` [, *N* `integer` [, *flags* `text` [, *subexpr* `integer`]]]]) → `text`
Возвращает подстроку *N*-го вхождения регулярного выражения POSIX *pattern* в строке *string* или `NULL`, если таких вхождений не нашлось; см. Подраздел 9.7.3.
`regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i')` → `CDEF`
`regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i', 2)` → `EF`

`repeat` (*string* `text`, *number* `integer`) → `text`
Повторяет содержимое *string* указанное число (*number*) раз.
`repeat('Pg', 4)` → `PgPgPgPg`

`replace` (*string* `text`, *from* `text`, *to* `text`) → `text`
Заменяет все вхождения в *string* подстроки *from* подстрокой *to*.
`replace('abcdefabcdef', 'cd', 'XX')` → `abXXefabXXef`

`reverse` (`text`) → `text`
Переставляет символы в строке в обратном порядке.
`reverse('abcde')` → `edcba`

`right` (*string* `text`, *n* `integer`) → `text`
Возвращает последние *n* символов в строке. Когда *n* меньше нуля, возвращаются все символы справа, кроме первых |*n*|.
`right('abcde', 2)` → `de`

`split_part` (*string* `text`, *delimiter* `text`, *n* `integer`) → `text` Разделяет строку *string* по символу *delimiter* и выдаёт *n*-ый элемент (считая с 1) или *n*-ый с конца, если *n* отрицательный.

Функция/оператор **Описание** **Пример(ы)**
`split_part('abc~@~def~@~ghi', '~@~', 2)` → `def` `split_part('abc,def,ghi,jkl', ',', -2)` → `ghi`

`starts_with` (*`string`* `text`, *`prefix`* `text`) → `boolean`
Возвращает `true`, если строка *`string`* начинается с подстроки *`prefix`*.
`starts_with('alphabet', 'alph')` → `t`

`string_to_array` (*`string`* `text`, *`delimiter`* `text` [, *`null_string`* `text`]) → `text[]`
Разделяет заданную параметром *`string`* строку на поля по разделителю *`delimiter`* и формирует из полученных полей массив значений `text`. Если в качестве *`delimiter`* передаётся `NULL`, каждый символ *`string`* становится отдельным элементом массива. Если в *`delimiter`* передаётся пустая строка, вся строка *`string`* воспринимается как одно поле и помещается в один элемент массива. Если в аргументе *`null_string`* передана отличная от `NULL` строка, поля, совпадающие с этой строкой, заменяются значениями `NULL`. См. также `array_to_string`.
`string_to_array('xx~~yy~~zz', '~~', 'yy')` → `{xx,NULL,zz}`

`string_to_table` (*`string`* `text`, *`delimiter`* `text` [, *`null_string`* `text`]) → `setof text`
Разделяет заданную параметром *`string`* строку на поля по разделителю *`delimiter`* и возвращает полученные поля как набор строк типа `text`. Если в качестве *`delimiter`* передаётся `NULL`, каждый символ *`string`* становится отдельной строкой результата. Если в *`delimiter`* передаётся пустая строка, вся строка *`string`* воспринимается как одно поле. Если в аргументе *`null_string`* передана отличная от `NULL` строка, совпадающие с ней поля заменяются значениями `NULL`.
`string_to_table('xx~^~yy~^~zz', '~^~', 'yy')` →
`xx`
`NULL`
`zz`

`strpos` (*`string`* `text`, *`substring`* `text`) → `integer`
Возвращает начальную позицию первого вхождения *`substring`* в строке *`string`* либо 0, если такого вхождения нет. (Ей подобна функция `position(` *`substring`* `in` *`string`*`)`, но обратите внимание на другой порядок аргументов.)
`strpos('high', 'ig')` → `2`

`substr` (*`string`* `text`, *`start`* `integer` [, *`count`* `integer`]) → `text`
Извлекает из *`string`* подстроку, начиная с позиции *`start`*, длиной до *`count`* символов (если это значение указано). (Ей равнозначна функция `substring(` *`string`* `from` *`start`* `for` *`count`*`)`.)
`substr('alphabet', 3)` → `phabet`
`substr('alphabet', 3, 2)` → `ph`

`to_ascii` (*`string`* `text`) → `text`
`to_ascii` (*`string`* `text`, *`encoding`* `name`) → `text`
`to_ascii` (*`string`* `text`, *`encoding`* `integer`) → `text`
Преобразует *`string`* в ASCII из другой кодировки, задаваемой по имени или номеру. В отсутствие указания *`encoding`* подразумевается кодировка базы данных (и на практике это единственный полезный вариант использования). В основном суть преобразования сводится к отбрасыванию диакритических знаков. Это преобразование поддерживается только для кодировок `LATIN1`, `LATIN2`, `LATIN9` и `WIN1250`. (Другое, более гибкое решение реализовано в модуле unaccent).
`to_ascii('Karél')` → `Karel`

Функция/оператор **Описание** **Пример(ы)**

`to_bin ( integer ) → text`
`to_bin ( bigint ) → text`
Преобразует число в двоичное представление в дополнительном коде.
`to_bin(2147483647)` → `1111111111111111111111111111111`
`to_bin(-1234)` → `11111111111111111111101100101110`

`to_hex ( integer ) → text`
`to_hex ( bigint ) → text`
Преобразует число в шестнадцатеричное представление в дополнительном коде.
`to_hex(2147483647)` → `7fffffff`
`to_hex(-1234)` → `fffffb2e`

`to_oct ( integer ) → text`
`to_oct ( bigint ) → text`
Преобразует число в восьмеричное представление в дополнительном коде.
`to_oct(2147483647)` → `17777777777`
`to_oct(-1234)` → `37777775456`

`translate (` *`string`* `text,` *`from`* `text,` *`to`* `text ) → text`
Заменяет каждый символ в *`string`*, входящий в множество *`from`*, на соответствующий символ в множестве *`to`*. Если строка *`from`* длиннее *`to`*, найденные в исходной строке лишние символы *`from`* удаляются.
`translate('12345', '143', 'ax')` → `a2x5`

`unistr ( text ) → text`
Преобразует спецпоследовательности Unicode в аргументе. Символы Unicode могут быть заданы как `\XXXX` (4 шестнадцатеричных цифры), `\+XXXXXX` (6 шестнадцатеричных цифр), `\uXXXX` (4 шестнадцатеричных цифры) или `\UXXXXXXXX` (8 шестнадцатеричных цифр). Чтобы указать обратную косую черту, напишите две обратные косые черты. Все остальные символы берутся буквально.
Когда кодировка сервера — не UTF-8, символ с кодом, заданным этой спецпоследовательностью, преобразуется в фактическую кодировку сервера; если такое преобразование невозможно, выдаётся ошибка.
Эта функция предоставляет (нестандартную) альтернативу строковым константам со спецпоследовательностями Unicode (см. Подраздел 4.1.2.3).
`unistr('d\0061t\+000061')` → `data`
`unistr('d\u0061t\U00000061')` → `data`

Функции `concat`, `concat_ws` и `format` принимают переменное число аргументов, так что им для объединения или форматирования можно передавать значения в виде массива, помеченного ключевым словом `VARIADIC` (см. Подраздел 36.5.6). Элементы такого массива обрабатываются, как если бы они были обычными аргументами функции. Если вместо массива в соответствующем аргументе передаётся NULL, функции `concat` и `concat_ws` возвращают NULL, а `format` воспринимает NULL как массив нулевого размера.

Также обратите внимание на агрегатную функцию `string_agg` в Разделе 9.21 и функции для преобразования текста в `bytea` и наоборот в Таблице 9.13.

9.4.1. `format`

Функция `format` выдаёт текст, отформатированный в соответствии со строкой формата, подобно функции `sprintf` в C.

```
format(formatstr text [, formatarg "any" [, ...] ])
```

`formatstr` — строка, определяющая, как будет форматироваться результат. Обычный текст в строке формата непосредственно копируется в результат, за исключением *спецификаторов формата*. Спецификаторы формата представляют собой местозаполнители, определяющие, как должны форматироваться и выводиться в результате аргументы функции. Каждый аргумент *`formatarg`* преобразуется в текст по правилам вывода своего типа данных, а затем форматируется и вставляется в результирующую строку согласно спецификаторам формата.

Спецификаторы формата предваряются символом `%` и имеют форму

```
%[position][flags][width]type
```

Здесь:

`position` (необязательный)

Строка вида *`n$`*, где *`n`* — индекс выводимого аргумента. Индекс, равный 1, выбирает первый аргумент после *`formatstr`*. Если *`position`* опускается, по умолчанию используется следующий аргумент по порядку.

`flags` (необязательный)

Дополнительные флаги, управляющие форматированием данного спецификатора. В настоящее время поддерживается только знак минус (`-`), который выравнивает результат спецификатора по левому краю. Этот флаг работает, только если также определено поле *`width`*.

`width` (необязательный)

Задаёт *минимальное* число символов, которое будет занимать результат данного спецификатора. Выводимое значение выравнивается по правой или левой стороне (в зависимости от флага `-`) с дополнением необходимым числом пробелов. Если ширина слишком мала, она просто игнорируется, т. е. результат не усекается. Ширину можно обозначить положительным целым, звёздочкой (`*`), тогда ширина будет получена из следующего аргумента функции, или строкой вида `*`*`n$`*, тогда ширина будет задаваться в *`n`*-ом аргументе функции.

Если ширина передаётся в аргументе функции, этот аргумент выбирается до аргумента, используемого для спецификатора. Если аргумент ширины отрицательный, результат выравнивается по левой стороне (как если бы был указан флаг `-`) в рамках поля длины `abs(`*`width`*`)`.

`type` (обязательный)

Тип спецификатора определяет преобразование соответствующего выводимого значения. Поддерживаются следующие типы:

- `s` форматирует значение аргумента как простую строку. Значение NULL представляется пустой строкой.
- `I` обрабатывает значение аргумента как SQL-идентификатор, при необходимости заключая его в кавычки. Значение NULL для такого преобразования считается ошибочным (так же, как и для `quote_ident`).
- `L` заключает значение аргумента в апострофы, как строку SQL. Значение NULL выводится буквально, как `NULL`, без кавычек (так же, как и с `quote_nullable`).

В дополнение к спецификаторам, описанным выше, можно использовать спецпоследовательность `%%`, которая просто выведет символ `%`.

Несколько примеров простых преобразований формата:

```
SELECT format('Hello %s', 'World');
Результат: Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
```

```
Результат: Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
Результат: INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
Результат: INSERT INTO locations VALUES('C:\Program Files')
```

Следующие примеры иллюстрируют использование поля *width* и флага `-`:

```
SELECT format('|%10s|', 'foo');
Результат: |       foo|

SELECT format('|%-10s|', 'foo');
Результат: |foo       |

SELECT format('|%*s|', 10, 'foo');
Результат: |       foo|

SELECT format('|%*s|', -10, 'foo');
Результат: |foo       |

SELECT format('|%-*s|', 10, 'foo');
Результат: |foo       |

SELECT format('|%-*s|', -10, 'foo');
Результат: |foo       |
```

Эти примеры показывают применение полей *position*:

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
Результат: Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
Результат: |       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
Результат: |       foo|
```

В отличие от стандартной функции C `sprintf`, функция `format` в Postgres Pro позволяет комбинировать в одной строке спецификаторы с полями *position* и без них. Спецификатор формата без поля *position* всегда использует следующий аргумент после последнего выбранного. Кроме того, функция `format` не требует, чтобы в строке формата использовались все аргументы функции. Пример этого поведения:

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
Результат: Testing three, two, three
```

Спецификаторы формата `%I` и `%L` особенно полезны для безопасного составления динамических операторов SQL. См. Пример 41.1.

9.5. Функции и операторы двоичных строк

В этом разделе описываются функции и операторы для работы с двоичными строками, то есть со значениями типа `bytea`. Многие из них идентичны по функциональности и синтаксису описанным в предыдущем разделе функциям, предназначенным для текстовых строк.

В SQL определены несколько строковых функций, в которых аргументы разделяются не запятыми, а ключевыми словами. Подробнее это описано в Таблице 9.11. Postgres Pro также предоставляет варианты этих функций с синтаксисом, обычным для функций (см. Таблицу 9.12).

Таблица 9.11. SQL-функции и операторы для работы с двоичными строками

Функция/оператор **Описание** **Пример(ы)**
bytea || bytea → bytea Соединяет две двоичные строки. '\x123456'::bytea || '\x789a00bcde'::bytea → \x123456789a00bcde
bit_length (bytea) → integer Возвращает число бит в двоичной строке (это число в 8 раз больше octet_length). bit_length('\x123456'::bytea) → 24
btrim (*bytes* bytea, *bytesremoved* bytea) → bytea Удаляет наибольшую строку, содержащую только байты, заданные в параметре *bytesremoved*, с начала и с конца строки *bytes*. btrim('\x1234567890'::bytea, '\x9012'::bytea) → \x345678
ltrim (*bytes* bytea, *bytesremoved* bytea) → bytea Удаляет наибольшую строку, содержащую только байты, заданные в параметре *bytesremoved*, с начала строки *bytes*. ltrim('\x1234567890'::bytea, '\x9012'::bytea) → \x34567890
octet_length (bytea) → integer Возвращает число байт в двоичной строке. octet_length('\x123456'::bytea) → 3
overlay (*bytes* bytea PLACING *newsubstring* bytea FROM *start* integer [FOR *count* integer]) → bytea Заменяет подстроку в *bytes*, начиная с байта с номером *start*, длиной *count* байт, на подстроку *newsubstring*. В отсутствие параметра *count* количество заменяемых байтов определяется длиной *newsubstring*. overlay('\x1234567890'::bytea placing '\002\003'::bytea from 2 for 3) → \x12020390
position (*substring* bytea IN *bytes* bytea) → integer Возвращает начальную позицию первого вхождения *substring* в *bytes* либо 0, если такого вхождения нет. position('\x5678'::bytea in '\x1234567890'::bytea) → 3
rtrim (*bytes* bytea, *bytesremoved* bytea) → bytea Удаляет наибольшую строку, содержащую только байты, заданные в параметре *bytesremoved*, с конца строки *bytes*. rtrim('\x1234567890'::bytea, '\x9012'::bytea) → \x12345678
substring (*bytes* bytea [FROM *start* integer] [FOR *count* integer]) → bytea Извлекает из *bytes* подстроку, начиная с позиции *start* (если она указана), длиной до *count* символов (если она указана). Параметры *start* и *count* могут опускаться, но не оба сразу. substring('\x1234567890'::bytea from 3 for 2) → \x5678
trim ([LEADING | TRAILING | BOTH] *bytesremoved* bytea FROM *bytes* bytea) → bytea Удаляет наибольшую строку, содержащую только байты, заданные в параметре *bytesremoved*, с начала, с конца или с обеих сторон (по умолчанию подразумевается указание BOTH) строки *bytes*. trim('\x9012'::bytea from '\x1234567890'::bytea) → \x345678
trim ([LEADING | TRAILING | BOTH] [FROM] *bytes* bytea, *bytesremoved* bytea) → bytea

Функция/оператор **Описание** **Пример(ы)**
Это нестандартный синтаксис вызова `trim()` . `trim(both from '\x1234567890'::bytea, '\x9012'::bytea)` → `\x345678`

В PostgreSQL есть и другие функции для работы с двоичными строками, перечисленные в Таблице 9.12. Некоторые из них используются в качестве внутренней реализации стандартных функций SQL, приведённых в Таблице 9.11.

Таблица 9.12. Другие функции для работы с двоичными строками

Функция **Описание** **Пример(ы)**

`bit_count` (*bytes* `bytea`) → `bigint`
Возвращает число бит в двоичной строке (эту операцию также называют «popcount»).
`bit_count('\x1234567890'::bytea)` → `15`

`get_bit` (*bytes* `bytea`, *n* `bigint`) → `integer`
Извлекает из двоичной строки бит с номером n.
`get_bit('\x1234567890'::bytea, 30)` → `1`

`get_byte` (*bytes* `bytea`, *n* `integer`) → `integer`
Извлекает из двоичной строки байт с номером n.
`get_byte('\x1234567890'::bytea, 4)` → `144`

`length` (`bytea`) → `integer`
Выдаёт число байт в двоичной строке.
`length('\x1234567890'::bytea)` → `5`

`length` (*bytes* `bytea`, *encoding* `name`) → `integer`
Выдаёт число символов в двоичной строке, в предположении, что она содержит текст в кодировке *encoding*.
`length('jose'::bytea, 'UTF8')` → `4`

`md5` (`bytea`) → `text`
Вычисляет MD5-хеш двоичной строки и выдаёт результат в шестнадцатеричном виде.
`md5('Th\000omas'::bytea)` → `8ab2d3c9689aaf18b4958c334c82d8b1`

`set_bit` (*bytes* `bytea`, *n* `bigint`, *newvalue* `integer`) → `bytea`
Устанавливает в двоичной строке для бита с номером n значение *newvalue*.
`set_bit('\x1234567890'::bytea, 30, 0)` → `\x1234563890`

`set_byte` (*bytes* `bytea`, *n* `integer`, *newvalue* `integer`) → `bytea`
Устанавливает в двоичной строке для байта с номером n значение *newvalue*.
`set_byte('\x1234567890'::bytea, 4, 64)` → `\x1234567840`

`sha224` (`bytea`) → `bytea`
Вычисляет хеш SHA-224 для двоичной строки.
`sha224('abc'::bytea)` → `\x23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7`

`sha256` (`bytea`) → `bytea`
Вычисляет хеш SHA-256 для двоичной строки.
`sha256('abc'::bytea)` → `\xba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`

`sha384` (`bytea`) → `bytea`

Функция **Описание** **Пример(ы)**
Вычисляет хеш SHA-384 для двоичной строки. `sha384('abc'::bytea)` → `\xcb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7`

`sha512 ( bytea ) → bytea`
Вычисляет хеш SHA-512 для двоичной строки.
`sha512('abc'::bytea)` → `\xddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f`

`substr ( `*bytes*` bytea, `*start*` integer [, `*count*` integer] ) → bytea`
Извлекает из *bytes* подстроку, начиная с позиции *start*, длиной до *count* байт (если это значение указано). (Ей равнозначна функция `substring(` *bytes* `from` *start* `for` *count*`)`.)
`substr('\x1234567890'::bytea, 3, 2)` → `\x5678`

Для функций `get_byte` и `set_byte` байты нумеруется с 0. Функции `get_bit` и `set_bit` нумеруют биты справа налево; например, бит 0 будет меньшим значащим битом первого байта, а бит 15 — большим значащим битом второго байта.

По историческим причинам функция `md5` возвращает значение в шестнадцатеричном виде в типе `text`, тогда как функции SHA-2 возвращают тип `bytea`. Для преобразования значения из одного представления в другое используйте функции `encode` и `decode`. Например, вызвав `encode(sha256('abc'), 'hex')`, вы получите значение в шестнадцатеричном виде в текстовой строке, а `decode(md5('abc'), 'hex')` выдаст значение `bytea`.

В Таблице 9.13 показаны функции для перекодирования текста из одного набора символов (кодировки) в другой и для представления произвольных двоичных данных в текстовом виде. Для всех этих функций аргумент или результат типа `text` содержит текст в текущей кодировке базы данных, тогда как аргументы или результаты типа `bytea` содержат текст в кодировке, заданной соответствующим аргументом.

Таблица 9.13. Функции для преобразования текстовых/двоичных строк

Функция **Описание** **Пример(ы)**

`convert ( `*bytes*` bytea, `*src_encoding*` name, `*dest_encoding*` name ) → bytea`
Преобразует двоичную строку, содержащую текст в кодировке *src_encoding*, в двоичную строку с текстом в кодировке *dest_encoding* (возможные варианты преобразований описаны в Подразделе 22.3.4).
`convert('text_in_utf8', 'UTF8', 'LATIN1')` → `\x746578745f696e5f75746638`

`convert_from ( `*bytes*` bytea, `*src_encoding*` name ) → text`
Преобразует двоичную строку, содержащую текст в кодировке *src_encoding*, в строку типа `text` в кодировке базы данных (возможные варианты преобразований описаны в Подразделе 22.3.4).
`convert_from('text_in_utf8', 'UTF8')` → `text_in_utf8`

`convert_to ( `*string*` text, `*dest_encoding*` name ) → bytea`
Преобразует строку типа `text` в кодировке базы данных в двоичную строку с текстом в кодировке *dest_encoding* (возможные варианты преобразований описаны в Подразделе 22.3.4).
`convert_to('некоторый_текст', 'UTF8')` → `\x736f6d655f74657874`

Функция **Описание** **Пример(ы)**

`encode` (*bytes* `bytea`, *format* `text`) → `text`
Переводит двоичные данные в текстовое представление; поддерживаются следующие значения *format*: `base64`, `escape`, `hex`.
`encode('123\000\001', 'base64')` → `MTIzAAE=`

`decode` (*string* `text`, *format* `text`) → `bytea`
Переводит двоичные данные из текстового представления; поддерживает те же значения *format*, что и функция `encode`.
`decode('MTIzAAE=', 'base64')` → `\x3132330001`

Функции `encode` и `decode` поддерживают следующие текстовые форматы:

base64

Формат `base64` описан в *RFC 2045, Разделе 6.8*. Согласно этому RFC, закодированные строки разбиваются по 76 символов. Однако завершаются строки не символами CRLF (как требуется в соответствии с MIME), а одним символом конца строки. Функция `decode`, с другой стороны, игнорирует символы перевода каретки, новой строки, пробелы и табуляции. Если на вход `decode` поступают некорректные данные base64, возникает ошибка — в том числе, если оказывается некорректным завершающее выравнивание.

escape

В формате `escape` нулевые байты и байты с установленным старшим битом переводятся в восьмеричные спецпоследовательности (`\`*nnn*), а обратная косая черта дублируется. Другие байтовые значения представляются в буквальном виде. Функция `decode` выдаст ошибку, встретив обратную косую черту, за которой не следует ещё одна обратная косая или три восьмеричных цифры; другие значения байта она принимает без изменений.

hex

В формате `hex` каждые 4 бита данных представляются одной шестнадцатеричной цифрой, от `0` до `f`, при этом первой идёт цифра, представляющая старшие биты. Шестнадцатеричные цифры `a`-`f` функция `encode` выводит в нижнем регистре. Так как наименьшая единица данных — байт (8 бит), функция `encode` всегда возвращает чётное количество символов. Функция `decode`, с другой стороны, принимает символы `a`-`f` в любом регистре. Если на вход функции `decode` поступают некорректные данные, возникает ошибка — в том числе, если число символов оказывается нечётным.

См. также агрегатную функцию `string_agg` в Разделе 9.21 и функции для работы с большими объектами в Разделе 33.4.

9.6. Функции и операторы для работы с битовыми строками

В этом разделе описываются функции и операторы, предназначенные для работы с битовыми строками, то есть с данными типов `bit` и `bit varying`. (Хотя в этих таблицах упоминается только тип `bit`, с ним полностью взаимозаменяем тип `bit varying`.) Для битовых строк поддерживаются обычные операторы сравнения, показанные в Таблице 9.1, а также операторы, приведённые в Таблице 9.14.

Таблица 9.14. Операторы для работы с битовыми строками

Оператор **Описание** **Пример(ы)**
`bit` `||` `bit` → `bit`

Оператор Описание Пример(ы)
Конкатенация `B'10001' || B'011'` → `10001011`

`bit & bit` → `bit`
Битовое И (операнды должны быть одинаковой длины)
`B'10001' & B'01101'` → `00001`

`bit | bit` → `bit`
Битовое ИЛИ (операнды должны быть одинаковой длины)
`B'10001' | B'01101'` → `11101`

`bit # bit` → `bit`
Битовое исключающее ИЛИ (операнды должны быть одинаковой длины)
`B'10001' # B'01101'` → `11100`

`~ bit` → `bit`
Битовое НЕ
`~ B'10001'` → `01110`

`bit << integer` → `bit`
Битовый сдвиг влево (с сохранением длины строки)
`B'10001' << 3` → `01000`

`bit >> integer` → `bit`
Битовый сдвиг вправо (с сохранением длины строки)
`B'10001' >> 2` → `00100`

Некоторые функции, работающие с двоичными строками, также работают и с битовыми строками, как показано в Таблице 9.15.

Таблица 9.15. Функции для работы с битовыми строками

Функция Описание Пример(ы)

`bit_count ( bit )` → `bigint`
Возвращает число бит в битовой строке (эту операцию так же называют «popcount»).
`bit_count(B'10111')` → `4`

`bit_length ( bit )` → `integer`
Возвращает число бит в битовой строке.
`bit_length(B'10111')` → `5`

`length ( bit )` → `integer`
Возвращает число бит в битовой строке.
`length(B'10111')` → `5`

`octet_length ( bit )` → `integer`
Возвращает число байт в битовой строке.
`octet_length(B'1011111011')` → `2`

`overlay ( bits bit PLACING newsubstring bit FROM start integer [FOR count integer] )` → `bit` Заменяет подстроку в *bits*, начиная с бита с номером *start*, длиной *count* бит, на подстроку *newsubstring*. В отсутствие параметра *count* количество заменяемых бит определяется длиной *newsubstring*.

Функция **Описание** **Пример(ы)**
`overlay(B'01010101010101010' placing B'11111' from 2 for 3)` → `0111110101010101010`

`position` (*`substring`* `bit` `IN` *`bits`* `bit`) → `integer`
Возвращает начальную позицию первого вхождения *`substring`* в *`bits`* либо 0, если такого вхождения нет.
`position(B'010' in B'000001101011')` → `8`

`substring` (*`bits`* `bit` [`FROM` *`start`* `integer`] [`FOR` *`count`* `integer`]) → `bit`
Извлекает из *`bits`* подстроку, начиная с позиции *`start`* (если она указана), длиной до *`count`* бит (если она указана). Параметры *`start`* и *`count`* могут опускаться, но не оба сразу.
`substring(B'110010111111' from 3 for 2)` → `00`

`get_bit` (*`bits`* `bit`, *`n`* `integer`) → `integer`
Извлекает из битовой строки бит с номером *`n`*; первый (самый левый) бит имеет номер 0.
`get_bit(B'101010101010101010', 6)` → `1`

`set_bit` (*`bits`* `bit`, *`n`* `integer`, *`newvalue`* `integer`) → `bit`
Устанавливает для бита с номером *`n`* в битовой строке значение *`newvalue`*; первый (самый левый) бит имеет номер 0.
`set_bit(B'101010101010101010', 6, 0)` → `101010001010101010`

Кроме того, целые значения можно преобразовать в тип `bit` и обратно. Приведение целого к типу `bit(n)` заключается в копировании в это значение правых `n` бит. Когда целое приводится к битовой строке, имеющей длину больше, чем размер этого целого, результат дополняется слева знаком. Некоторые примеры:

```
44::bit(10)                    0000101100
44::bit(3)                     100
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

Заметьте, что приведение к типу «bit» без длины будет означать приведение к `bit(1)` и в результате будет получен только один самый младший бит числа.

9.7. Поиск по шаблону

Postgres Pro предлагает три разных способа поиска текста по шаблону: традиционный оператор `LIKE` языка SQL, более современный `SIMILAR TO` (добавленный в SQL:1999) и регулярные выражения в стиле POSIX. Помимо простых операторов, отвечающих на вопрос «соответствует ли строка этому шаблону?», в Postgres Pro есть функции для извлечения или замены соответствующих подстрок и для разделения строки по заданному шаблону.

Подсказка

Если этих встроенных возможностей оказывается недостаточно, вы можете написать собственные функции на языке Perl или Tcl.

Внимание

Хотя чаще всего поиск по регулярному выражению бывает очень быстрым, регулярные выражения бывают и настолько сложными, что их обработка может занять приличное время и объём памяти. Поэтому опасайтесь шаблонов регулярных выражений, поступающих из недоверенных источников. Если у вас нет другого выхода, рекомендуется ввести тайм-аут для операторов.

Поиск с шаблонами `SIMILAR TO` несёт те же риски безопасности, так как конструкция `SIMILAR TO` предоставляет во многом те же возможности, что и регулярные выражения в стиле POSIX.

Поиск с `LIKE` гораздо проще, чем два другие варианта, поэтому его безопаснее использовать с недоверенными источниками шаблонов поиска.

Все три вида операторов поиска по шаблону не поддерживают недетерминированные правила сортировки. В случае необходимости это ограничение можно обойти, применив к выражению другое правило сортировки.

9.7.1. LIKE

```
строка LIKE шаблон [ESCAPE спецсимвол]
строка NOT LIKE шаблон [ESCAPE спецсимвол]
```

Выражение `LIKE` возвращает true, если *`строка`* соответствует заданному *`шаблону`*. (Как можно было ожидать, выражение `NOT LIKE` возвращает false, когда `LIKE` возвращает true, и наоборот. Этому выражению равносильно выражение `NOT (строка LIKE шаблон)`.)

Если *`шаблон`* не содержит знаков процента и подчёркиваний, тогда шаблон представляет в точности строку и `LIKE` работает как оператор сравнения. Подчёркивание (`_`) в *`шаблоне`* подменяет (вместо него подходит) любой символ; а знак процента (`%`) подменяет любую (в том числе и пустую) последовательность символов.

Несколько примеров:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

При проверке по шаблону `LIKE` всегда рассматривается вся строка. Поэтому, если нужно найти последовательность символов где-то в середине строки, шаблон должен начинаться и заканчиваться знаками процента.

Чтобы найти в строке буквальное вхождение знака процента или подчёркивания, перед соответствующим символом в *`шаблоне`* нужно добавить спецсимвол. По умолчанию в качестве спецсимвола выбрана обратная косая черта, но с помощью предложения `ESCAPE` можно выбрать и другой. Чтобы включить спецсимвол в шаблон поиска, продублируйте его.

Примечание

Если параметр standard_conforming_strings выключен, каждый символ обратной косой черты, записываемый в текстовой константе, нужно дублировать. Подробнее это описано в Подразделе 4.1.2.1.

Также можно отказаться от спецсимвола, написав `ESCAPE ''`. При этом механизм спецпоследовательностей фактически отключается и использовать знаки процента и подчёркивания буквально в шаблоне нельзя.

Согласно стандарту SQL, отсутствие указания `ESCAPE` означает, что спецсимвол не определён (то есть спецсимволом не будет обратная косая черта), а пустое значение в `ESCAPE` не допускается. Таким образом, в этом поведение Postgres Pro несколько отличается от оговорённого в стандарте.

Вместо `LIKE` можно использовать ключевое слово `ILIKE`, чтобы поиск был регистр-независимым с учётом текущей языковой среды. Этот оператор не описан в стандарте SQL; это расширение Postgres Pro.

Кроме того, в Postgres Pro есть оператор `~~`, равнозначный `LIKE`, и `~~*`, соответствующий `ILIKE`. Есть также два оператора `!~~` и `!~~*`, представляющие `NOT LIKE` и `NOT ILIKE`, соответственно. Все эти операторы относятся к особенностям Postgres Pro. Вы можете увидеть их, например, в выводе команды `EXPLAIN`, так как при разборе запроса проверка `LIKE` и подобные заменяются ими.

Фразы `LIKE`, `ILIKE`, `NOT LIKE` и `NOT ILIKE` в синтаксисе Postgres Pro обычно обрабатываются как операторы; например, их можно использовать в конструкциях *`выражение оператор`* ANY (*`подвыражение`*), хотя предложение `ESCAPE` здесь добавить нельзя. В некоторых особых случаях всё же может потребоваться использовать вместо них нижележащие операторы.

Также обратите внимание на оператор проверки начала строки `^@` и соответствующую функцию `starts_with()`, которые полезны в случаях, когда нужно произвести сопоставление только с началом строки.

9.7.2. Регулярные выражения `SIMILAR TO`

```
строка SIMILAR TO шаблон [ESCAPE спецсимвол]
строка NOT SIMILAR TO шаблон [ESCAPE спецсимвол]
```

Оператор `SIMILAR TO` возвращает true или false в зависимости от того, соответствует ли данная строка шаблону или нет. Он работает подобно оператору `LIKE`, только его шаблоны соответствуют определению регулярных выражений в стандарте SQL. Регулярные выражения SQL представляют собой любопытный гибрид синтаксиса `LIKE` с синтаксисом обычных регулярных выражений (POSIX).

Как и `LIKE`, условие `SIMILAR TO` истинно, только если шаблон соответствует всей строке; это отличается от условий с регулярными выражениями, в которых шаблон может соответствовать любой части строки. Также подобно `LIKE`, `SIMILAR TO` воспринимает символы `_` и `%` как знаки подстановки, подменяющие любой один символ или любую подстроку, соответственно (в регулярных выражениях POSIX им аналогичны символы `.` и `.*`).

Помимо средств описания шаблонов, позаимствованных от `LIKE`, `SIMILAR TO` поддерживает следующие метасимволы, унаследованные от регулярных выражений POSIX:

- `|` означает выбор (одного из двух вариантов).
- `*` означает повторение предыдущего элемента 0 и более раз.
- `+` означает повторение предыдущего элемента 1 и более раз.
- `?` означает вхождение предыдущего элемента 0 или 1 раз.
- `{`*`m`*`}` означает повторяет предыдущего элемента ровно *`m`* раз.
- `{`*`m`*`,}` означает повторение предыдущего элемента *`m`* или более раз.
- `{`*`m`*`,`*`n`*`}` означает повторение предыдущего элемента не менее чем *`m`* и не более чем *`n`* раз.
- Скобки `()` объединяют несколько элементов в одну логическую группу.
- Квадратные скобки `[...]` обозначают класс символов так же, как и в регулярных выражениях POSIX.

Обратите внимание, точка (`.`) не является метасимволом для оператора `SIMILAR TO`.

Как и с `LIKE`, обратная косая черта отменяет специальное значение любого из этих метасимволов. Предложение `ESCAPE` позволяет выбрать другой спецсимвол, а запись `ESCAPE ''` — отказаться от использования спецсимвола.

Согласно стандарту SQL, отсутствие указания `ESCAPE` означает, что спецсимвол не определён (то есть спецсимволом не будет обратная косая черта), а пустое значение в `ESCAPE` не допускается. Таким образом, в этом поведение Postgres Pro несколько отличается от оговорённого в стандарте.

Ещё одно отклонение от стандарта состоит в том, что следующая за спецсимволом буква или цифра открывает возможности спецпоследовательностей, определённых для регулярных выражений POSIX; см. Таблицу 9.20, Таблицу 9.21 и Таблицу 9.22.

Несколько примеров:

```
'abc' SIMILAR TO 'abc'          true
'abc' SIMILAR TO 'a'            false
'abc' SIMILAR TO '%(b|d)%'      true
'abc' SIMILAR TO '(b|c)%'       false
'-abc-' SIMILAR TO '%\mabc\M%'  true
'xabcy' SIMILAR TO '%\mabc\M%'  false
```

Функция `substring` с тремя параметрами производит извлечение подстроки, соответствующей шаблону регулярного выражения SQL. Эту функцию можно вызвать в стандартном синтаксисе SQL:

```
substring(строка similar шаблон escape спецсимвол)
```

или используя синтаксис уже устаревшего стандарта SQL:1999:

```
substring(строка from шаблон for спецсимвол)
```

или в виде обычной функции с тремя аргументами:

```
substring(строка, шаблон, спецсимвол)
```

Как и с `SIMILAR TO`, указанному шаблону должна соответствовать вся строка; в противном случае функция не найдёт ничего и вернёт NULL. Для выделения в шаблоне границ подстроки, которая представляет интерес в соответствующей этому шаблону входной строке, шаблон может содержать два спецсимвола с кавычками (`"`) после каждого. В случае успешного обнаружения шаблона эта функция возвращает часть строки, заключённую между этими разделителями.

Разделители «спецсимвол+кавычки» фактически разделяют шаблон `substring` на три независимых регулярных выражения, так что, например, вертикальная черта (`|`) в любой из этих трёх частей действует только в рамках этой части. Кроме того, в случае неоднозначности в выборе подстрок, соответствующих этим частям, первое и третье регулярные выражения считаются захватывающими наименьшие возможные подстроки. (На языке POSIX можно сказать, что первое и третье регулярные выражения — «нежадные».)

В качестве расширения стандарта SQL в Postgres Pro допускается указание только одного разделителя, в этом случае третье регулярное выражение считается пустым; также разделители могут отсутствовать вовсе, в этом случае пустыми считаются первое и третье регулярные выражения.

Несколько примеров с маркерами `#"`, выделяющими возвращаемую строку:

```
substring('foobar' similar '%#"o_b#"%' escape '#')   oob
substring('foobar' similar '#"o_b#"%' escape '#')    NULL
```

9.7.3. Регулярные выражения POSIX

В Таблице 9.16 перечислены все существующие операторы для проверки строк регулярными выражениями POSIX.

Таблица 9.16. Операторы регулярных выражений

Оператор Описание Пример(ы)

`text ~ text → boolean`
Проверка соответствия строки регулярному выражению с учётом регистра
`'thomas' ~ 't.*ma' → t`

`text ~* text → boolean`
Проверка соответствия строки регулярному выражению без учёта регистра
`'thomas' ~* 'T.*ma' → t`

Оператор **Описание** **Пример(ы)**

`text !~ text` → `boolean`
Проверка несоответствия строки регулярному выражению с учётом регистра
`'thomas' !~ 't.*max'` → `t`

`text !~* text` → `boolean`
Проверка несоответствия строки регулярному выражению без учёта регистра
`'thomas' !~* 'T.*ma'` → `f`

Регулярные выражения POSIX предоставляют более мощные средства поиска по шаблонам, чем операторы `LIKE` и `SIMILAR TO`. Во многих командах Unix, таких как `egrep`, `sed` и `awk` используется язык шаблонов, похожий на описанный здесь.

Регулярное выражение — это последовательность символов, представляющая собой краткое определение набора строк (*регулярное множество*). Строка считается соответствующей регулярному выражению, если она является членом регулярного множества, описываемого регулярным выражением. Как и для `LIKE`, символы шаблона непосредственно соответствуют символам строки, за исключением специальных символов языка регулярных выражений. При этом спецсимволы регулярных выражений отличается от спецсимволов `LIKE`. В отличие от шаблонов `LIKE`, регулярное выражение может совпадать с любой частью строки, если только оно не привязано явно к началу и/или концу строки.

Несколько примеров:

```
'abcd' ~ 'bc'     true
'abcd' ~ 'a.c'    true — точке соответствует любой символ
'abcd' ~ 'a.*d'   true — * обозначает повторение предыдущего элемента шаблона
'abcd' ~ '(b|x)'  true — | означает ИЛИ для группы в скобках
'abcd' ~ '^a'     true — ^ привязывает шаблон к началу строки
'abcd' ~ '^(b|c)' false — совпадение не найдено по причине привязки к началу
```

Более подробно язык шаблонов в стиле POSIX описан ниже.

Функция `substring` с двумя параметрами, `substring(`*строка* `from` *шаблон*`)`, извлекает подстроку, соответствующую шаблону регулярного выражения POSIX. Она возвращает первый фрагмент текста, подходящий шаблону, если таковой находится в строке, либо NULL в противном случае. Но если шаблон содержит скобки, она возвращает первое подвыражение, заключённое в скобки (то, которое начинается с самой первой открывающей скобки). Если вы хотите использовать скобки, но не в таком особом режиме, можно просто заключить в них всё выражение. Если же вам нужно включить скобки в шаблон до подвыражения, которое вы хотите извлечь, это можно сделать, используя группы без захвата, которые будут описаны ниже.

Несколько примеров:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

Функция `regexp_count` подсчитывает количество мест, где шаблон регулярного выражения POSIX соответствует строке. Она имеет синтаксис `regexp_count`(*строка*, *шаблон* [, *начало* [, *флаги*]]). *Шаблон* ищется в *строке* обычно с начала строки, но если указан параметр *начало*, то начиная с этого индекса. В параметре *флаги* может передаваться текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Например, буква `i` во *флагах* выбирает регистронезависимый режим поиска. Поддерживаемые флаги описаны в Таблице 9.24.

Несколько примеров:

```
regexp_count('ABCABCAXYaxy', 'A.')          3
regexp_count('ABCABCAXYaxy', 'A.', 1, 'i')  4
```

Функция `regexp_instr` возвращает начальную или конечную позицию *N*-го вхождения шаблона регулярного выражения POSIX в строке или ноль, если такого вхождения нет. Она имеет синтаксис `regexp_instr`(*строка*, *шаблон* [, *начало* [, *N* [, *выбор_конца* [, *флаги* [, *подвыражение*]]]]]). *Шаблон* ищется в *строке* обычно с начала строки, но если указан параметр *начало*, то начиная с этого индекса. Если указано *N*, то ищется *N*-е вхождение шаблона, иначе ищется первое вхождение. Если параметр *выбор_конца* опущен или равен нулю, функция возвращает позицию первого символа вхождения. Иначе *выбор_конца* должен быть равен одному и функция возвращает позицию символа, следующего за вхождением. В параметре *флаги* может передаваться текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Например, буква `i` во *флагах* выбирает регистронезависимый режим поиска. Поддерживаемые флаги описаны в Таблице 9.24. Для шаблона, содержащего подвыражения в скобках, *подвыражение* — целое число, которое выбирает представляющее интерес подвыражение: в результате выдаётся позиция подстроки, соответствующей этому подвыражению. Подвыражения нумеруются по порядку открывающих их скобок. Когда *подвыражение* опущено или равно нулю, в результате выдаётся позиция всего вхождения независимо от подвыражений в скобках.

Несколько примеров:

```
regexp_instr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                   23
regexp_instr('ABCDEFGHI', '(c..)(...)', 1, 1, 0, 'i', 2)
                                   6
```

Функция `regexp_like` проверяет, встречается ли в строке вхождение шаблона регулярного выражения POSIX, возвращая логическое значение true или false. Она имеет синтаксис `regexp_like`(*строка*, *шаблон* [, *флаги*]). В параметре *флаги* передаётся необязательная текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Поддерживаемые флаги описаны в Таблица 9.24. Если флаги не указаны, эта функция выдаёт те же результаты, что и оператор `~`. Если указан только флаг `i`, она выдаёт те же результаты, что и оператор `~*`.

Несколько примеров:

```
regexp_like('Hello World', 'world')       false
regexp_like('Hello World', 'world', 'i')  true
```

Функция `regexp_match` возвращает текстовый массив с совпавшими подстроками из первого вхождения шаблона регулярного выражения POSIX в строке. Она имеет синтаксис `regexp_match`(*строка*, *шаблон* [, *флаги*]). Если вхождение не находится, результатом будет `NULL`. Если вхождение находится и *шаблон* не содержит подвыражений в скобках, результатом будет текстовый массив с одним элементом, содержащим подстроку, которая соответствует всему шаблону. Если вхождение находится и *шаблон* содержит подвыражения в скобках, результатом будет текстовый массив, в котором *n*-м элементом будет *n*-е заключённое в скобки подвыражение *шаблона* (не считая «незахватывающих» скобок; подробнее см. ниже). В параметре *флаги* передаётся необязательная текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Допустимые флаги описаны в Таблице 9.24.

Несколько примеров:

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

Подсказка

В общем случае, когда нужно просто получить всю найденную подстроку или `NULL` при отсутствии вхождений, лучше использовать `regexp_substr()`. Однако функция `regexp_substr()` есть только в Postgres Pro версии 15 и выше. В предыдущих версиях для этого можно извлечь первый элемент результата `regexp_match()`, например:

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
 regexp_match
--------------
 barbeque
(1 row)
```

Функция `regexp_matches` возвращает набор текстовых массивов с совпавшими подстроками из вхождений шаблона регулярного выражения POSIX в строке. Она имеет тот же синтаксис, что и `regexp_match`. Эта функция не возвращает никаких строк, если вхождений нет; возвращает одну строку, если найдено одно вхождение и не передан флаг `g`, или *N* строк, если найдено *N* вхождений и передан флаг `g`. Каждая возвращаемая строка представляет собой текстовый массив, содержащий всю найденную подстроку или подстроки, соответствующие заключённым в скобки подвыражениям *шаблона*, как и описанный выше результат `regexp_match`. Функция `regexp_matches` принимает все флаги, показанные в Таблице 9.24, а также флаг `g`, указывающий ей выдать все вхождения, а не только первое.

Несколько примеров:

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

Подсказка

В большинстве случаев `regexp_matches()` должна применяться с флагом `g`, так как если вас интересует только первое вхождение, проще и эффективнее использовать функцию `regexp_match()`. Однако `regexp_match()` существует только в Postgres Pro версии 10 и выше. В старых версиях обычно помещали вызов `regexp_matches()` во вложенный SELECT, например, так:

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

В результате выдаётся текстовый массив, если вхождение найдено, или `NULL` в противном случае, так же как с `regexp_match()`. Без вложенного SELECT этот запрос не возвращает никакие строки, если соответствие не находится, а это обычно не то, что нужно.

Функция `regexp_replace` подставляет другой текст вместо подстрок, соответствующих шаблонам регулярных выражений POSIX. Она имеет синтаксис `regexp_replace(`*исходная_строка*, *шаблон*, *замена* [, *начало* [, *N*]] [, *флаги*]`)`. (Заметьте, что если не задано *начало*, указывать *N* нельзя, но *флаги* можно указывать в любом случае.) Если *исходная_строка* не содержит фрагмента, подходящего под *шаблон*, она возвращается неизменной. Если же соответствие находится, возвращается *исходная_строка*, в которой вместо соответствующего фрагмента подставляется *замена*. Строка *замена* может содержать `\`*n*, где *n* — число от 1 до 9, указывающее на исходную подстроку, соответствующую *n*-му подвыражению в скобках, и может содержать обозначение `\&`, указывающее, что будет вставлена подстрока, соответствующая всему шаблону. Если же в текст замены нужно включить обратную косую черту буквально, следует написать `\\`. *Шаблон* ищется в *строке* обычно с начала строки, но если указан параметр *начало*, то начиная с этого индекса. По умолчанию заменяется только первое вхождение шаблона. Если *N* указано и больше нуля, заменяется *N*-е вхождение шаблона. Если передан флаг `g` или заданное *N* равно нулю, заменяются все вхождения с позиции *начало* и далее. (Флаг `g` игнорируется, когда задано *N*.) В необязательном параметре *флаги* передаётся текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Допустимые флаги (кроме `g`) описаны в Таблице 9.24.

Несколько примеров:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                   fooXarYXazY
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 0, 'i')
                                   X PXstgrXSQL fXnctXXn
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 3, 'i')
                                   A PostgrXSQL function
```

Функция `regexp_split_to_table` разделяет строку, используя в качестве разделителя шаблон регулярного выражения POSIX. Она имеет синтаксис `regexp_split_to_table`(*строка*, *шаблон* [, *флаги*]). Если *шаблон* не находится в переданной строке, возвращается вся *строка* целиком. Если находится минимум одно вхождение, для каждого такого вхождения возвращается текст от конца предыдущего вхождения (или начала строки) до начала вхождения. После последнего найденного вхождения возвращается фрагмент от его конца до конца строки. В необязательном параметре *флаги* передаётся текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Флаги, которые поддерживает `regexp_split_to_table`, описаны в Таблице 9.24.

Функция `regexp_split_to_array` ведёт себя подобно `regexp_split_to_table`, за исключением того, что `regexp_split_to_array` возвращает результат в массиве элементов типа `text`. Она имеет синтаксис `regexp_split_to_array`(*строка*, *шаблон* [, *флаги*]). Параметры у этой функции те же, что и у `regexp_split_to_table`.

Несколько примеров:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog',
 '\s+') AS foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
```

```
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

Как показывает последний пример, функции разделения по регулярным выражениям игнорируют вхождения нулевой длины, идущие в начале и в конце строки, а также непосредственно за предыдущим вхождением. Это поведение противоречит строгому определению поиска по регулярным выражениям, который реализуют другие функции, но обычно более удобно на практике. Подобное поведение наблюдается и в других программных средах, например в Perl.

Функция `regexp_substr` возвращает подстроку, соответствующую шаблону регулярного выражения POSIX, или `NULL`, если такого соответствия нет. Она имеет синтаксис `regexp_substr(`*`строка`*, *`шаблон`* [, *`начало`* [, *`N`* [, *`флаги`* [, *`подвыражение`*]]]]). *`Шаблон`* ищется в *`строке`* обычно с начала строки, но если указан параметр *`начало`*, то начиная с этого индекса. Если указано *`N`*, возвращается *`N`*-е вхождение шаблона, в противном случае возвращается первое вхождение. В параметре *`флаги`* передаётся необязательная текстовая строка, содержащая ноль или более однобуквенных флагов, меняющих поведение функции. Поддерживаемые флаги описаны в Таблица 9.24. Для шаблона, содержащего подвыражения в скобках, *`подвыражение`* — целое число, которое выбирает представляющее интерес подвыражение: в результате выдаётся подстрока, соответствующая этому подвыражению. Подвыражения нумеруются по порядку открывающих их скобок. Когда *`подвыражение`* опущено или равно нулю, в результате выдаётся всё вхождение независимо от подвыражений в скобках.

Несколько примеров:

```
regexp_substr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                 town zip
regexp_substr('ABCDEFGHI', '(c..)(...)', 1, 1, 'i', 2)
                                FGH
```

9.7.3.1. Подробное описание регулярных выражений

Регулярные выражения в Postgres Pro реализованы с использованием программного пакета, который разработал Генри Спенсер (Henry Spencer). Практически всё следующее описание регулярных выражений дословно скопировано из его руководства.

Регулярное выражение (Regular expression, RE), согласно определению в POSIX 1003.2, может иметь две формы: *расширенное* RE или ERE (грубо говоря, это выражения, которые понимает `egrep`) и *простое* RE или BRE (грубо говоря, это выражения для `ed`). Postgres Pro поддерживает обе формы, а кроме того реализует некоторые расширения, не предусмотренные стандартом POSIX, но широко используемые вследствие их доступности в некоторых языках программирования, например в Perl и Tcl. Регулярные выражения, использующие эти несовместимые с POSIX расширения, здесь называются *усовершенствованными* RE или ARE. ARE практически представляют собой надмножество ERE, тогда как BRE отличаются некоторой несовместимостью в записи (помимо того, что они гораздо более ограничены). Сначала мы опишем формы ARE и ERE, отметив особенности, присущие только ARE, а затем расскажем, чем от них отличаются BRE.

> ## Примечание
>
> Postgres Pro изначально всегда предполагает, что регулярное выражение следует правилам ARE. Однако можно переключиться на более ограниченные правила ERE или BRE, добавив в шаблон RE *встроенный параметр*, как описано в Подразделе 9.7.3.4. Это может быть полезно для совместимости с приложениями, ожидающими от СУБД строгого следования правилам POSIX 1003.2.

Регулярное выражение определяется как одна или более *ветвей*, разделённых символами `|`. Оно считается соответствующим всему, что соответствует одной из этих ветвей.

Ветвь — это ноль или несколько *количественных атомов* или *ограничений*, соединённых вместе. Соответствие ветви в целом образуется из соответствия первой части, за которым следует соответствие второй части и т. д.; пустой ветви соответствует пустая строка.

Количественный атом — это *атом*, за которым может следовать *определитель количества*. Без этого определителя ему соответствует одно вхождение атома. С определителем количества ему может соответствовать некоторое число вхождений этого атома. Все возможные *атомы* перечислены в Таблице 9.17. Варианты определителей количества и их значения перечислены в Таблице 9.18.

Ограничению соответствует пустая строка, но это соответствие возможно только при выполнении определённых условий. Ограничения могут использоваться там же, где и атомы, за исключением того, что их нельзя дополнять определителями количества. Простые ограничения показаны в Таблице 9.19; некоторые дополнительные ограничения описаны ниже.

Таблица 9.17. Атомы регулярных выражений

Атом	**Описание**
`(` *`re`*`)`	(где *`re`* — любое регулярное выражение) описывает соответствие *`re`*, при этом данное соответствие захватывается для последующей обработки
`(?:` *`re`*`)`	подобно предыдущему, но соответствие не захватывается (т. е. это набор скобок «без захвата») (применимо только к ARE)
`.`	соответствует любому символу
`[`*`символы`*`]`	*выражение в квадратных скобках*, соответствует любому из *`символов`* (подробнее это описано в Подразделе 9.7.3.2)
`\`*`k`*	(где *`k`* — не алфавитно-цифровой символ) соответствует обычному символу буквально, т. е. `\\` соответствует обратной косой черте
`\`*`c`*	где *`c`* — алфавитно-цифровой символ (за которым могут следовать другие символы), это *спецсимвол*, см. Подраздел 9.7.3.3 (применим только к ARE; в ERE и BRE этому атому соответствует *`c`*)
`{`	когда за этим символом следует любой символ, кроме цифры, этот атом соответствует левой

Атом	**Описание**
	фигурной скобке (`{`), если же за ним следует цифра, это обозначает начало *границы* (см. ниже)
x	(где *x* — один символ, не имеющий специального значения) соответствует этому символу

Выражение RE не может заканчиваться обратной косой чертой (`\`).

Примечание

Если параметр standard_conforming_strings выключен, каждый символ обратной косой черты, записываемый в текстовой константе, нужно дублировать. Подробнее это описано в Подразделе 4.1.2.1.

Таблица 9.18. Определители количества в регулярных выражениях

Определитель	**Соответствует**
`*`	0 или более вхождений атома
`+`	1 или более вхождений атома
`?`	0 или 1 вхождение атома
`{`*m*`}`	ровно *m* вхождений атома
`{`*m*`,}`	*m* или более вхождений атома
`{`*m*`,` *n*`}`	от *m* до *n* (включая границы) вхождений атома; *m* не может быть больше *n*
`*?`	нежадная версия `*`
`+?`	нежадная версия `+`
`??`	нежадная версия `?`
`{`*m*`}?`	нежадная версия `{`*m*`}`
`{`*m*`,}?`	нежадная версия `{`*m*`,}`
`{`*m*`,` *n*`}?`	нежадная версия `{`*m*`,` *n*`}`

В формах с `{...}` числа *m* и *n* определяют так называемые *границы* количества. Эти числа должны быть беззнаковыми десятичными целыми в диапазоне от 0 до 255 включительно.

Не жадные определители (допустимые только в ARE) описывают те же возможные соответствия, что и аналогичные им обычные («*жадные*»), но предпочитают выбирать наименьшее, а не наибольшее количество вхождений. Подробнее это описано в Подразделе 9.7.3.5.

Примечание

Определители количества не могут следовать один за другим, например запись `**` будет ошибочной. Кроме того, определители не могут стоять в начале выражения или подвыражения и идти сразу после `^` или `|`.

Таблица 9.19. Ограничения в регулярных выражениях

Ограничение	**Описание**
`^`	соответствует началу строки
`$`	соответствует концу строки

Ограничение	**Описание**
`(?=` *`re`*`)`	*позитивный просмотр вперёд* находит соответствие там, где начинается подстрока, соответствующая *`re`* (только для ARE)
`(?!` *`re`*`)`	*негативный просмотр вперёд* находит соответствие там, где не начинается подстрока, соответствующая *`re`* (только для ARE)
`(?<=` *`re`*`)`	*позитивный просмотр назад* находит соответствие там, где заканчивается подстрока, соответствующая *`re`* (только для ARE)
`(?<!` *`re`*`)`	*негативный просмотр назад* находит соответствие там, где не заканчивается подстрока, соответствующая *`re`* (только для ARE)

Ограничения просмотра вперёд и назад не могут содержать *ссылки назад* (см. Подраздел 9.7.3.3), и все скобки в них считаются «скобками без захвата».

9.7.3.2. Выражения в квадратных скобках

Выражение в квадратных скобках содержит список символов, заключённый в `[]`. Обычно ему соответствует любой символ из списка (об исключении написано ниже). Если список начинается с `^`, ему соответствует любой символ, который *не* перечисляется далее в этом списке. Если два символа в списке разделяются знаком `-`, это воспринимается как краткая запись полного интервала символов между двумя заданными (и включая их) в порядке сортировки; например выражению `[0-9]` в ASCII соответствует любая десятичная цифра. Два интервала не могут разделять одну границу, т. е. выражение `a-c-e` недопустимо. Интервалы зависят от порядка сортировки, который может меняться, поэтому в переносимых программах их лучше не использовать.

Чтобы включить в список `]`, этот символ нужно написать первым (сразу за `^`, если он присутствует). Чтобы включить в список символ `-`, его нужно написать первым или последним, либо как вторую границу интервала. Указать `-` в качестве первой границы интервал можно, заключив его между `[.` и `.]`, чтобы он стал элементом сортировки (см. ниже). За исключением этих символов, некоторых комбинаций с `[` (см. следующие абзацы) и спецсимволов (в ARE), все остальные специальные символы в квадратных скобках теряют своё особое значение. В частности, символ `\` по правилам ERE или BRE воспринимается как обычный, хотя в ARE он экранирует символ, следующий за ним.

Выражения в квадратных скобках могут содержать элемент сортировки (символ или последовательность символов или имя такой последовательности), определение которого заключается между `[.` и `.]`. Определяющая его последовательность воспринимается в выражении в скобках как один элемент. Это позволяет включать в такие выражения элементы, соответствующие последовательности нескольких символов. Например, с элементом сортировки `ch` в квадратных скобках регулярному выражению `[[.ch.]]*c` будут соответствовать первые пять символов строки `chchcc`.

> **Примечание**
>
> В настоящее время Postgres Pro не поддерживает элементы сортировки, состоящие из нескольких символов. Эта информация относится к возможному в будущем поведению.

В квадратных скобках могут содержаться элементы сортировки, заключённые между `[=` и `=]`, обозначающие *классы эквивалентности*, т. е. последовательности символов из всех элементов сортировки, эквивалентных указанному, включая его самого. (Если для этого символа нет эквивалентных, он обрабатывается как заключённый между `[.` и `.]`.) Например, если `е` и `ё` — члены одного класса эквивалентности, выражения `[[=е=]]`, `[[=ё=]]` и `[её]` будут равнозначными. Класс эквивалентности нельзя указать в качестве границы интервала.

В квадратных скобках может также задаваться имя класса символов, заключённое между `[:` и `:]`, которое обозначает множество всех символов, принадлежащих этому классу. Класс символов не может представлять границу интервала. В стандарте POSIX определены следующие имена классов: `alnum` (буквы и цифры), `alpha` (буквы), `blank` (пробел и табуляция), `cntrl` (управляющие символы), `digit` (десятичные цифры), `graph` (печатаемые символы, кроме пробела), `lower` (буквы в нижнем регистре), `print` (печатаемые символы, включая пробел), `punct` (знаки пунктуации), `space` (любые пробельные символы), `upper` (буквы в верхнем регистре) и `xdigit` (шестнадцатеричные цифры). На согласованное поведение этих стандартных классов символов на разных платформах можно рассчитывать в рамках 7-битного набора ASCII. Будет ли определённый не ASCII-символ считаться принадлежащим одному из этих классов, зависит от правила сортировки, используемого оператором или функцией регулярных выражений (см. Раздел 22.2). По умолчанию это правило определяется свойством `LC_CTYPE` базы данных (см. Раздел 22.1). Классификация не ASCII-символов может меняться от платформы к платформе даже при выборе локалей с похожими названиями. (Но локаль `C` никогда не относит не ASCII-символы ни к одному из этих классов.) Помимо этих стандартных классов символов, в Postgres Pro определён класс символов `word`, отличающийся от класса `alnum` лишь тем, что включает ещё символ подчёркивания (_), и класс символов `ascii`, который содержит все 7-битные символы ASCII, но не какие-либо другие.

Есть два особых вида выражений в квадратных скобках: выражения `[[:<:]]` и `[[:>:]]`, представляющие собой ограничения, соответствующие пустым строкам в начале и конце слова. Слово в данном контексте определяется как последовательность словосоставляющих символов, перед или после которой нет словосоставляющих символов. Словосоставляющий символ — это любой символ, принадлежащий классу `word`, то есть любая буква, цифра или подчёркивание. Это расширение совместимо со стандартом POSIX 1003.2, но не описано в нём, и поэтому его следует использовать с осторожностью там, где важна совместимость с другими системами. Обычно лучше использовать ограничивающие спецсимволы, описанные ниже; они также не совсем стандартны, но набрать их легче.

9.7.3.3. Спецсимволы регулярных выражений

Спецсимволы — это специальные команды, состоящие из `\` и последующего алфавитно-цифрового символа. Можно выделить следующие категории спецсимволов: обозначения символов, коды классов, ограничения и ссылки назад. Символ `\`, за которым идёт алфавитно-цифровой символ, не образующий допустимый спецсимвол, считается ошибочным в ARE. В ERE спецсимволов нет: вне квадратных скобок пара из `\` и последующего алфавитно-цифрового символа, воспринимается просто как данный символ, а в квадратных скобках и сам символ `\` воспринимается просто как обратная косая черта. (Последнее на самом деле нарушает совместимость между ERE и ARE.)

Спецобозначения символов введены для того, чтобы облегчить ввод в RE непечатаемых и других неудобных символов. Они приведены в Таблице 9.20.

Коды классов представляют собой краткий способ записи имён некоторых распространённых классов символов. Они перечислены в Таблице 9.21.

Спецсимволы ограничений обозначают ограничения, которым при совпадении определённых условий соответствует пустая строка. Они перечислены в Таблице 9.22.

Ссылка назад (`\`*n*) соответствует той же строке, какой соответствовало предыдущее подвыражение в скобках под номером *n* (см. Таблицу 9.23). Например, `([bc])\1` соответствует `bb` или `cc`, но не `bc` или `cb`. Это подвыражение должно полностью предшествовать ссылке назад в RE. Нумеруются подвыражения в порядке следования их открывающих скобок. При этом скобки без захвата исключаются из рассмотрения. Ссылка назад учитывает только строковые символы, совпадающие с указанным подвыражением, но не содержащиеся в нём ограничения. Например, выражению `(^\d)\1` будет соответствовать `22`).

Таблица 9.20. Спецобозначения символов в регулярных выражениях

Спецсимвол	Описание
`\a`	символ звонка, как в C
`\b`	символ «забой», как в C

Спецсимвол	**Описание**
`\B`	синоним для обратной косой черты (`\`), сокращающий потребность в дублировании этого символа
`\c`*`X`*	(где *`X`* — любой символ) символ, младшие 5 бит которого те же, что и у *`X`*, а остальные равны 0
`\e`	символ, определённый в последовательности сортировки с именем `ESC`, либо, если таковой не определён, символ с восьмеричным значением `033`
`\f`	подача формы, как в C
`\n`	новая строка, как в C
`\r`	возврат каретки, как в C
`\t`	горизонтальная табуляция, как в C
`\u`*`wxyz`*	(где *`wxyz`* ровно четыре шестнадцатеричные цифры) символ с шестнадцатеричным кодом `0x`*`wxyz`*
`\U`*`stuvwxyz`*	(где *`stuvwxyz`* ровно восемь шестнадцатеричных цифр) символ с шестнадцатеричным кодом `0x`*`stuvwxyz`*
`\v`	вертикальная табуляция, как в C
`\x`*`hhh`*	(где *`hhh`* — несколько шестнадцатеричных цифр) символ с шестнадцатеричным кодом `0x`*`hhh`* (символ всегда один вне зависимости от числа шестнадцатеричных цифр)
`\0`	символ с кодом `0` (нулевой байт)
`\`*`xy`*	(где *`xy`* — ровно две восьмеричных цифры, не *ссылка назад*) символ с восьмеричным кодом `0`*`xy`*
`\`*`xyz`*	(где *`xyz`* — ровно три восьмеричных цифры, не *ссылка назад*) символ с восьмеричным кодом `0`*`xyz`*

Шестнадцатеричные цифры записываются символами `0`-`9` и `a`-`f` или `A`-`F`. Восьмеричные цифры — цифры от `0` до `7`.

Спецпоследовательности с числовыми кодами, задающими значения вне диапазона ASCII (0–127), воспринимаются по-разному в зависимости от кодировки базы данных. Когда база данных имеет кодировку UTF-8, спецкод равнозначен позиции символа в Unicode, например, `\u1234` обозначает символ `U+1234`. Для других многобайтных кодировок спецпоследовательности обычно просто задают серию байт, определяющих символ. Если в кодировке базы данных отсутствует символ, заданный спецпоследовательностью, ошибки не будет, но и никакие данные не будут ей соответствовать.

Символы, переданные спецобозначением, всегда воспринимаются как обычные символы. Например, `\135` кодирует `]` в ASCII, но спецпоследовательность `\135` не будет закрывать выражение в квадратных скобках.

Таблица 9.21. Спецкоды классов в регулярных выражениях

Спецсимвол	**Описание**
`\d`	соответствует любой цифре; равнозначно `[[:digit:]]`

Спецсимвол	**Описание**
`\s`	соответствует любому пробельному символу; равнозначно `[[:space:]]`
`\w`	соответствует любому буквенному символу; равнозначно `[[:word:]]`
`\D`	соответствует любому нецифровому символу; равнозначно `[^[:digit:]]`
`\S`	соответствует любому непробельному символу; равнозначно `[^[:space:]]`
`\W`	соответствует любому небуквенному символу; равнозначно `[^[:word:]]`

Коды классов также работают в выражениях в квадратных скобках, хотя определения, показанные выше, не совсем синтаксически верны в данном контексте. Например, выражение `[a-c\d]` равнозначно `[a-c[:digit:]]`.

Таблица 9.22. Спецсимволы ограничений в регулярных выражений

Спецсимвол	**Описание**
`\A`	соответствует только началу строки (чем это отличается от `^`, описано в Подразделе 9.7.3.5)
`\m`	соответствует только началу слова
`\M`	соответствует только концу слова
`\y`	соответствует только началу или концу слова
`\Y`	соответствует только положению не в начале и не в конце слова
`\Z`	соответствует только концу строки (чем это отличается от `$`, описано в Подразделе 9.7.3.5)

Определением слова здесь служит то же, что было приведено выше в описании `[[:<:]]` и `[[:>:]]`. В квадратных скобках спецсимволы ограничений не допускаются.

Таблица 9.23. Ссылки назад в регулярных выражениях

Спецсимвол	**Описание**
`\`*`m`*	(где *`m`* — цифра, отличная от 0) — ссылка назад на подвыражение под номером *`m`*
`\`*`mnn`*	(где *`m`* — цифра, отличная от 0, а *`nn`* — ещё несколько цифр с десятичным значением *`mnn`*, не превышающим число закрытых до этого скобок с захватом) ссылка назад на подвыражение под номером *`mnn`*

Примечание

Регулярным выражениям присуща неоднозначность между восьмеричными кодами символов и ссылками назад, которая разрешается следующим образом (это упоминалось выше). Ведущий ноль всегда считается признаком восьмеричной последовательности. Единственная цифра, отличная от 0, за которой не следует ещё одна цифра, всегда воспринимается как ссылка назад. Последовательность из нескольких цифр, которая начинается не с 0, воспринимается как ссылка назад, если она идёт за подходящим подвыражением (т. е. число оказывается в диапазоне, допустимом для ссылки назад), в противном случае она воспринимается как восьмеричное число.

9.7.3.4. Метасинтаксис регулярных выражений

В дополнение к основному синтаксису, описанному выше, можно использовать также несколько особых форм и разнообразные синтаксические удобства.

Регулярное выражение может начинаться с одного из двух специальных префиксов режима. Если RE начинается с `***:`, его продолжение рассматривается как ARE. (В Postgres Pro это обычно не имеет значения, так как регулярные выражения воспринимаются как ARE по умолчанию; но это может быть полезно, когда параметр *флаги* функций regex включает режим ERE или BRE.) Если RE начинается с `***=`, его продолжение воспринимается как обычная текстовая строка, все его символы воспринимаются буквально.

ARE может начинаться со *встроенных параметров*: последовательности `(?`*xyz*`)` (где *xyz* — один или несколько алфавитно-цифровых символов), определяющих параметры остального регулярного выражения. Эти параметры переопределяют любые ранее определённые параметры, в частности они могут переопределить режим чувствительности к регистру, подразумеваемый для оператора regex, или параметр *флаги* функции regex. Допустимые буквы параметров показаны в Таблице 9.24. Заметьте, что те же буквы используются в параметре *флаги* функций regex.

Таблица 9.24. Буквы встроенных параметров ARE

Параметр	Описание
`b`	продолжение регулярного выражения — BRE
`c`	поиск соответствий с учётом регистра (переопределяет тип оператора)
`e`	продолжение RE — ERE
`i`	поиск соответствий без учёта регистра (см. Подраздел 9.7.3.5) (переопределяет тип оператора)
`m`	исторически сложившийся синоним `n`
`n`	поиск соответствий с учётом перевода строк (см. Подраздел 9.7.3.5)
`p`	переводы строк учитываются частично (см. Подраздел 9.7.3.5)
`q`	продолжение регулярного выражения — обычная строка («в кавычках»), содержимое которой воспринимается буквально
`s`	поиск соответствий без учёта перевода строк (по умолчанию)
`t`	компактный синтаксис (по умолчанию; см. ниже)
`w`	переводы строк учитываются частично, но в другом, «странном» режиме (см. Подраздел 9.7.3.5)
`x`	развёрнутый синтаксис (см. ниже)

Внедрённые параметры начинают действовать сразу после скобки `)`, завершающей их последовательность. Они могут находиться только в начале ARE (после указания `***:`, если оно присутствует).

Помимо обычного (*компактного*) синтаксиса RE, в котором имеют значение все символы, поддерживается также *развёрнутый* синтаксис, включить который можно с помощью встроенного параметра `x`. В развёрнутом синтаксисе игнорируются пробельные символы, а также все символы от `#` до конца строки (или конца RE). Это позволяет разделять RE на строки и добавлять в него комментарии. Но есть три исключения:

- пробельный символ или `#`, за которым следует `\`, сохраняется
- пробельный символ или `#` внутри выражения в квадратных скобках сохраняется
- пробельные символы и комментарии не могут присутствовать в составных символах, например, в `(?:`

В данном контексте пробельными символами считаются пробел, табуляция, перевод строки и любой другой символ, относящийся к классу символов *`space`*.

И наконец, в ARE последовательность `(?#`*`ttt`*`)` (где *`ttt`* — любой текст, не содержащий `)`) вне квадратных скобок также считается комментарием и полностью игнорируется. При этом она так же не может находиться внутри составных символов, таких как `(?:`. Эти комментарии в большей степени историческое наследие, чем полезное средство; они считаются устаревшими, а вместо них рекомендуется использовать развёрнутый синтаксис.

Ни одно из этих расширений метасинтаксиса не будет работать, если выражение начинается с префикса `***=`, после которого строка воспринимается буквально, а не как RE.

9.7.3.5. Правила соответствия регулярным выражениям

В случае, когда RE может соответствовать более чем одной подстроке в заданной строке, соответствующей RE считается подстрока, которая начинается в ней первой. Если к данной позиции подобных соответствующих подстрок оказывается несколько, из них выбирается либо самая длинная, либо самая короткая из возможных, в зависимости от того, какой режим выбран в RE: *жадный* или *не жадный*.

Где жадный или не жадный характер RE определяется по следующим правилам:

- Большинство атомов и все ограничения не имеют признака жадности (так как они всё равно не могут соответствовать подстрокам разного состава).
- Скобки, окружающие RE, не влияют на его «жадность».
- Атом с определителем фиксированного количества (`{`*`m`*`}` или `{`*`m`*`}?`) имеет ту же характеристику жадности (или может не иметь её), как и сам атом.
- Атом с другими обычными определителями количества (включая `{`*`m`*`,`*`n`*`}`, где *`m`* равняется *`n`*) считается жадным (предпочитает соответствие максимальной длины).
- Атом с не жадным определителем количества (включая `{`*`m`*`,`*`n`*`}?`, где *`m`* равно *`n`*) считается не жадным (предпочитает соответствие минимальной длины).
- Ветвь (RE без оператора `|` на верхнем уровне) имеет ту же характеристику жадности, что и первый количественный атом в нём, имеющий атрибут жадности.
- RE, образованное из двух или более ветвей, соединённых оператором `|`, всегда считается жадным.

Эти правила связывают характеристики жадности не только с отдельными количественными атомами, но и с ветвями и целыми RE, содержащими количественные атомы. Это означает, что при сопоставлении ветвь или целое RE может соответствовать максимально длинной или короткой подстроке *в целом*. Когда определена длина всего соответствия, часть его, соответствующая конкретному подвыражению, определяется с учётом характеристики жадности для этого подвыражения, при этом подвыражения, начинающиеся в RE раньше, имеют больший приоритет, чем следующие за ними.

Это иллюстрирует следующий пример:

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
```

```
Результат: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Результат: 1
```

В первом случае RE в целом жадное, так как жадным является атом `Y*`. Соответствие ему начинается с буквы `Y` и оно включает подстроку максимальной длины с этого места, т. е. подстроку `Y123`. Результат выражения — её часть, соответствующая подвыражению в скобках, т. е. `123`. Во втором случае, RE в целом наследует не жадный характер от атома `Y*?`. Соответствие ему так же начинается с `Y`, но включает оно подстроку минимальной длины с этого места, т. е. `Y1`. И хотя подвыражение `[0-9]{1,3}` имеет жадный характер, оно не может повлиять на выбор длины соответствия в целом, поэтому ему остаётся только подстрока `1`.

Другими словами, когда RE содержит и жадные, и не жадные подвыражения, всё соответствие будет максимально длинным или коротким в зависимости от характеристики всего RE. Характеристики, связанные с подвыражениями, влияют только на то, какую часть подстроки может «поглотить» одно подвыражение относительно другого.

Чтобы явно придать характеристику «жадности» или «нежадности» подвыражению или всему RE, можно использовать определители `{1,1}` и `{1,1}?`, соответственно. Это полезно, когда вам нужно, чтобы общая характеристика жадности RE отличалась от той, что вытекает из его элементов. Например, предположим, что вы пытаетесь выделить из строки, содержащей несколько цифр, эти цифры и части до и после них. Можно попытаться сделать это так:

```
SELECT regexp_match('abc01234xyz', '(.*)(\d+)(.*)');
Результат: {abc0123,4,xyz}
```

Но это не будет работать: первая группа `.*` — «жадная», она «съест» всё, что сможет, оставляя для соответствия `\d+` только последнюю возможность, то есть последнюю цифру. Можно попытаться сделать запрос «нежадным»:

```
SELECT regexp_match('abc01234xyz', '(.*?)(\d+)(.*)');
Результат: {abc,0,""}
```

И это не будет работать, так теперь весь RE в целом стал нежадным, и все соответствия завершаются как можно раньше. Но мы можем получить нужный результат, явно сделав жадным всё RE:

```
SELECT regexp_match('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Результат: {abc,01234,xyz}
```

Управление общей характеристикой «жадности» RE независимо от «жадности» его компонентов даёт большую гибкость в описании шаблонов переменной длины.

При определении более длинного или более короткого соответствия длины соответствий определяются в символах, а не в элементах сортировки. Пустая строка считается длиннее, чем отсутствие соответствия. Например, выражению `bb*` соответствуют три символа в середине строки `abbbc`, выражению `(week|wee)(night|knights)` — все десять символов `weeknights`; когда выражение `(.*).*` сопоставляется со строкой `abc`, подвыражению в скобках соответствуют все три символа; а когда `(a*)*` сопоставляется со строкой `bc`, то и RE в целом, и подстроке в скобках соответствует пустая строка.

Игнорирование регистра символов даёт практически тот же эффект, как если бы в алфавите исчезли различия прописных и строчных букв. Если буква, существующая и в верхнем, и в нижнем регистре, фигурирует вне квадратных скобок как обычный символ, она по сути преобразуется в выражение в квадратных скобках, содержащее оба варианта, например `x` становится `[xX]`. Если же она фигурирует в выражении в квадратных скобках, в это выражение добавляются все её варианты, например `[x]` становится `[xX]`, а `[^x]` — `[^xX]`.

Когда включён многострочный режим поиска, атом `.` и выражения в квадратных скобках с `^` никогда не будут соответствовать символам конца строки (так что соответствия никогда не будут пересекать границы строк, если в RE нет явных указаний на эти символы), а `^` и `$` будут соответствовать пустой подстроке не только в начале и конце всего текста, но и в начале и конце каждой отдельной его строки. Однако спецсимволы ARE `\A` и `\Z` по-прежнему будут соответствовать *толь-*

ко началу и концу всего текста. Кроме того, коды классов символов `\D` и `\W` будут соответствовать концам строк независимо от этого режима. (До Postgres Pro 14 они не соответствовали концам строк в многострочном режиме. Для использования старого поведения напишите `[^[:digit:]]` или `[^[:word:]]`.)

В режиме, когда переводы строк учитываются частично, особый смысл перевод строк имеет для атома `.` и выражений в квадратных скобках, но не для `^` и `$`.

В обратном частичном режиме, перевод строк имеет особый смысл для `^` и `$`, как и в режиме с учётом перевода строк, но не для `.` и выражений в квадратных скобках. Данный режим не очень полезен, но существует для симметрии.

9.7.3.6. Пределы и совместимость

В текущей реализации отсутствует какой-либо явно заданный предел длины RE. Однако, разрабатывая программы высокой степени переносимости, не следует применять RE длиннее 256 байт, так как другая POSIX-совместимая реализация может отказаться обрабатывать такие регулярные выражения.

Единственная особенность ARE, действительно несовместимая с ERE стандарта POSIX проявляется в том, что в ARE знак `\` не теряет своё специальное значение в квадратных скобках. Все другие расширения ARE используют синтаксические возможности, которые не определены, не допустимы или не поддерживаются в ERE; синтаксис переключения режимов (`***`) также выходит за рамки синтаксиса POSIX как для BRE, так и для ERE.

Многие расширения ARE заимствованы из языка Perl, но некоторые были изменены, оптимизированы, а несколько расширений Perl были исключены. В результате имеют место следующие несовместимости: атомы `\b` и `\B`, отсутствие специальной обработки завершающего перевода строки, добавление исключений в квадратных скобках в число случаев, когда учитывается перевод строк, особые условия для скобок и ссылок назад в ограничениях просмотра вперёд/назад и семантика «наиболее длинное/короткое соответствие» (вместо «первое соответствие»).

9.7.3.7. Простые регулярные выражения

BRE имеют ряд отличий от ERE. В BRE знаки `|`, `+` и `?` теряют специальное значение, а замены им нет. Границы количества окружаются символами `\{` и `\}`, тогда как `{` и `}` рассматриваются как обычные символы. Вложенные подвыражения помещаются между `\(` и `\)`, а `(` и `)` представляют обычные символы. Символ `^` воспринимается как обычный, если только он не находится в начале RE или подвыражения в скобках, `$` — тоже обычный символ, если он находится не в конце RE или в конце подвыражения в скобках, и `*` — обычный символ, когда он находится в начале RE или подвыражения в скобках (возможно, после начального `^`). И, наконец, в BRE работают ссылки назад с одной цифрой, `\<` и `\>` — синонимы для `[[:<:]]` и `[[:>:]]`, соответственно; никакие другие спецсимволы в BRE не поддерживаются.

9.7.3.8. Отличия от XQuery и стандарта SQL

Начиная с SQL:2008, в стандарт SQL входят операторы и функции регулярных выражений, которые выполняют сопоставление с шаблоном в соответствии со стандартом регулярных выражений XQuery:

- `LIKE_REGEX`
- `OCCURRENCES_REGEX`
- `POSITION_REGEX`
- `SUBSTRING_REGEX`
- `TRANSLATE_REGEX`

В Postgres Pro эти операторы и функции на данный момент не реализованы. Но вы можете в каждом случае получить примерно тот же результат, как показано в Таблице 9.25. (В этой таблице опущены различные необязательные предложения с обеих сторон.)

Таблица 9.25. Аналоги функций регулярных выражений

Стандарт SQL	PostgreSQL
строка `LIKE_REGEX` *шаблон*	`regexp_like(` *строка*, *шаблон*`)` или *строка* `~` *шаблон*
`OCCURRENCES_REGEX(` *шаблон* `IN` *строка*`)`	`regexp_count(` *строка*, *шаблон*`)`
`POSITION_REGEX(` *шаблон* `IN` *строка*`)`	`regexp_instr(` *строка*, *шаблон*`)`
`SUBSTRING_REGEX(` *шаблон* `IN` *строка*`)`	`regexp_substr(` *строка*, *шаблон*`)`
`TRANSLATE_REGEX(` *шаблон* `IN` *строка* `WITH` *замена*`)`	`regexp_replace(` *строка*, *шаблон*, *замена*`)`

Функции регулярных выражений, подобные существующим в PostgreSQL, также имеются в ряде других реализаций SQL, в то время как функции стандарта SQL распространены не так широко. Некоторые детали синтаксиса регулярных выражений, скорее всего, будут различаться в каждой реализации.

В функциях и операторах стандарта SQL используются регулярные выражения XQuery, синтаксис которых похож на ARE, описанный выше. Регулярные выражения на базе POSIX в существующей реализации и регулярные выражения XQuery имеют ряд заметных отличий, в том числе:

- Вычитание классов символов XQuery не поддерживается. Например, это вычитание позволяет извлекать только английские согласные так: `[a-z-[aeiou]]`.
- Коды классов символов XQuery `\c`, `\C`, `\i` и `\I` не поддерживаются.
- Элементы классов символов XQuery с обозначением `\p{UnicodeProperty}` или обратным, `\P{UnicodeProperty}`, не поддерживаются.
- В POSIX классы символов, такие как `\w` (см. Таблицу 9.21), интерпретируются согласно текущей локали (и вы можете управлять этим, добавив предложение `COLLATE` к оператору или вызову функции). В XQuery эти классы определяются по свойствам символов Unicode, поэтому одинаковое поведение возможно только с локалями, соответствующими требованиям Unicode.
- Синтаксис стандарта SQL (не сам язык XQuery) стремится воспринять больше вариантов «перевода строки», чем синтаксис POSIX. Для описанных выше вариантов сопоставления с учётом перевода строк переводом строки считается только символ ASCII NL (`\n`), тогда как SQL считает переводом строки также CR (`\r`), CRLF (`\r\n`) (перевод строки в стиле Windows) и некоторые присущие только Unicode символы, например, LINE SEPARATOR (U+2028). Стоит заметить, что согласно SQL коды шаблона `.` и `\s` должны считать последовательность `\r\n` одним символом, а не двумя.
- Из спецкодов, определяющих символы, описанных в Таблице 9.20, XQuery поддерживает только `\n`, `\r` и `\t`.
- XQuery не поддерживает синтаксис `[:`*имя*`:]` для указания класса символов в квадратных скобках.
- В XQuery отсутствуют условия с просмотром вперёд и назад, а также не поддерживаются спецсимволы ограничений, описанные в Таблице 9.22.
- Формы метасинтаксиса, описанные в Подразделе 9.7.3.4, в XQuery не существуют.
- Буквы флагов регулярных выражений, определённые в XQuery, имеют общее с буквами флагов в POSIX (см. Таблицу 9.24), но не равнозначны им. Одинаково ведут себя только флаги `i` и `q`, а все остальные различаются:
  - Флаги XQuery `s` (допускающий сопоставление точки с переводом строки) и `m` (допускающий сопоставление `^` и `$` с переводами строк) позволяют получить то же поведение, что и флаги `n`, `p` и `w` в POSIX, но они *не* равнозначны флагам POSIX `s` и `m`. В частности, заметьте, что по умолчанию точка соответствует переводу строки в POSIX, но не в XQuery.
  - Флаг `x` (игнорировать пробельные символы в шаблоне) в XQuery значительно отличается от флага расширенного режима POSIX. В POSIX флаг `x` дополнительно позволяет написать

комментарий в шаблоне (начиная с символа #), а пробельный символ после обратной косой черты не игнорируется.

9.8. Функции форматирования данных

Функции форматирования в Postgres Pro предоставляют богатый набор инструментов для преобразования самых разных типов данных (дата/время, целое, числа с плавающей и фиксированной точкой) в форматированные строки и обратно. Все они перечислены в Таблице 9.26. Все эти функции следует одному соглашению: в первом аргументе передаётся значение, которое нужно отформатировать, а во втором — шаблон, определяющий формат ввода или вывода.

Таблица 9.26. Функции форматирования

Функция	**Описание**	**Пример(ы)**
`to_char ( timestamp, text ) → text` `to_char ( timestamp with time zone, text ) → text`	Преобразует время в строку согласно заданному формату.	`to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS')` → `05:31:12`
`to_char ( interval, text ) → text`	Преобразует интервал в строку согласно заданному формату.	`to_char(interval '15h 2m 12s', 'HH24:MI:SS')` → `15:02:12`

`to_char ( *numeric_type* , text ) → text`	Преобразует число в строку согласно заданному формату; поддерживаются типы `integer`, `bigint`, `numeric`, `real`, `double precision`.	`to_char(125, '999')` → `125`
		`to_char(125.8::real, '999D9')` → `125.8`
		`to_char(-125.8, '999D99S')` → `125.80-`

`to_date ( text, text ) → date`	Преобразует строку в дату согласно заданному формату.	`to_date('05 Dec 2000', 'DD Mon YYYY')` → `2000-12-05`
`to_number ( text, text ) → numeric`	Преобразует строку в число согласно заданному формату.	`to_number('12,454.8-', '99G999D9S')` → `-12454.8`
`to_timestamp ( text, text ) → timestamp with time zone`	Преобразует строку в значение времени согласно заданному формату. (См. также `to_timestamp(double precision)` в Таблице 9.33.)	`to_timestamp('05 Dec 2000', 'DD Mon YYYY')` → `2000-12-05 00:00:00-05`

Подсказка

Функции `to_timestamp` и `to_date` предназначены для работы с входными форматами, которые нельзя преобразовать простым приведением. Для большинства стандартных форматов даты/времени работает простое приведение исходной строки к требуемому типу и использовать его гораздо легче. Так же и функцию `to_number` нет необходимости использовать для стандартных представлений чисел.

Шаблон вывода `to_char` может содержать ряд кодов, которые распознаются при форматировании и заменяются соответствующими данными в зависимости от заданного значения. Любой текст, который не является кодом, просто копируется в неизменном виде. Подобным образом в строке шаблона ввода (для других функций) шаблонные коды определяют, какие значения должны поступать из входной строки. Если в строке шаблона есть символы, не относящиеся к шаблонным кодам, соответствующие символы во входной строке просто пропускаются (вне зависимости от того, совпадают ли они с символами в строке шаблона).

Все коды форматирования даты и времени перечислены в Таблице 9.27.

Таблица 9.27. Коды форматирования даты/времени

Код	**Описание**
`HH`	час дня (01–12)
`HH12`	час дня (01–12)
`HH24`	час дня (00–23)
`MI`	минута (00–59)
`SS`	секунда (00–59)
`MS`	миллисекунда (000–999)
`US`	микросекунда (000000–999999)
`FF1`	десятая доля секунды (0–9)
`FF2`	сотая доля секунды (00–99)
`FF3`	миллисекунда (000–999)
`FF4`	десятитысячная доля секунды (0000–9999)
`FF5`	стотысячная доля секунды (00000–99999)
`FF6`	микросекунда (000000–999999)
`SSSS`, `SSSSS`	секунды после полуночи (0–86399)
`AM`, `am`, `PM` или `pm`	обозначение времени до/после полудня (без точек)
`A.M.`, `a.m.`, `P.M.` или `p.m.`	обозначение времени до/после полудня (с точками)
`Y,YYY`	год (4 или более цифр) с разделителем
`YYYY`	год (4 или более цифр)
`YYY`	последние 3 цифры года
`YY`	последние 2 цифры года
`Y`	последняя цифра года
`IYYY`	недельный год по ISO 8601 (4 или более цифр)
`IYY`	последние 3 цифры недельного года по ISO 8601
`IY`	последние 2 цифры недельного года по ISO 8601
`I`	последняя цифра недельного года по ISO 8601
`BC`, `bc`, `AD` или `ad`	обозначение эры (без точек)
`B.C.`, `b.c.`, `A.D.` или `a.d.`	обозначение эры (с точками)
`MONTH`	полное название месяца в верхнем регистре (дополненное пробелами до 9 символов)
`Month`	полное название месяца с большой буквы (дополненное пробелами до 9 символов)
`month`	полное название месяца в нижнем регистре (дополненное пробелами до 9 символов)

Код	Описание
`MON`	сокращённое название месяца в верхнем регистре (3 буквы в английском; в других языках длина может меняться)
`Mon`	сокращённое название месяца с большой буквы (3 буквы в английском; в других языках длина может меняться)
`mon`	сокращённое название месяца в нижнем регистре (3 буквы в английском; в других языках длина может меняться)
`MM`	номер месяца (01–12)
`DAY`	полное название дня недели в верхнем регистре (дополненное пробелами до 9 символов)
`Day`	полное название дня недели с большой буквы (дополненное пробелами до 9 символов)
`day`	полное название дня недели в нижнем регистре (дополненное пробелами до 9 символов)
`DY`	сокращённое название дня недели в верхнем регистре (3 буквы в английском; в других языках может меняться)
`Dy`	сокращённое название дня недели с большой буквы (3 буквы в английском; в других языках длина может меняться)
`dy`	сокращённое название дня недели в нижнем регистре (3 буквы в английском; в других языках длина может меняться)
`DDD`	номер дня в году (001–366)
`IDDD`	номер дня в году по ISO 8601 (001–371; первый день года — понедельник первой недели по ISO)
`DD`	день месяца (01–31)
`D`	номер дня недели, считая с воскресенья (`1`) до субботы (`7`)
`ID`	номер дня недели по ISO 8601, считая с понедельника (`1`) до воскресенья (`7`)
`W`	неделя месяца (1–5) (первая неделя начинается в первый день месяца)
`WW`	номер недели в году (1–53) (первая неделя начинается в первый день года)
`IW`	номер недели в году по ISO 8601 (01–53; первый четверг года относится к неделе 1)
`CC`	век (2 цифры) (двадцать первый век начался 2001-01-01)
`J`	юлианская дата (целое число дней от 24 ноября 4714 г. до н. э. 00:00 по местному времени; см. Раздел B.7)
`Q`	квартал
`RM`	номер месяца римскими цифрами в верхнем регистре (I–XII; I=январь)

Код	Описание
`rm`	номер месяца римскими цифрами в нижнем регистре (i–xii; i=январь)
`TZ`	аббревиатура часового пояса в верхнем регистре
`tz`	аббревиатура часового пояса в нижнем регистре
`TZH`	часы часового пояса
`TZM`	минуты часового пояса
`OF`	смещение часового пояса от UTC (*`HH`* или *`HH`*`:`*`MM`*)

К любым кодам форматирования можно добавить модификаторы, изменяющие их поведение. Например, шаблон форматирования `FMMonth` включает код `Month` с модификатором `FM`. Модификаторы, предназначенные для форматирования даты/времени, перечислены в Таблице 9.28.

Таблица 9.28. Модификаторы кодов для форматирования даты/времени

Модификатор	Описание	Пример
Приставка `FM`	режим заполнения (подавляет ведущие нули и дополнение пробелами)	`FMMonth`
Окончание `TH`	окончание порядкового числительного в верхнем регистре	`DDTH`, например `12TH`
Окончание `th`	окончание порядкового числительного в нижнем регистре	`DDth`, например `12th`
Приставка `FX`	глобальный параметр фиксированного формата (см. замечания)	`FX Month DD Day`
Приставка `TM`	режим перевода (используются локализованные названия дней и месяцев, исходя из lc_time)	`TMMonth`
Окончание `SP`	режим числа прописью (не реализован)	`DDSP`

Замечания по использованию форматов даты/времени:

- `FM` подавляет дополняющие пробелы и нули справа, которые в противном случае будут добавлены, чтобы результат имел фиксированную ширину. В Postgres Pro модификатор `FM` действует только на следующий код, тогда как в Oracle `FM` её действие распространяется на все последующие коды, пока не будет отключено последующим модификатором `FM`.
- `TM` подавляет замыкающие пробелы вне зависимости от указания `FM`.
- Функции `to_timestamp` и `to_date` игнорируют регистр букв во входной строке; поэтому, например, для шаблонов `MON`, `Mon` и `mon` подойдут одни и те же строки. Если используется приставка `TM`, смена регистра производится в соответствии с правилом сортировки, установленным для входной строки (см. Раздел 22.2).
- `to_timestamp` и `to_date` пропускают повторяющиеся пробелы в начале входной строки и вокруг значений даты и времени, если только не используется приставка `FX`. Например, `to_timestamp(' 2000    JUN', 'YYYY MON')` и `to_timestamp('2000 - JUN', 'YYYY-MON')` будут работать, но `to_timestamp('2000    JUN', 'FXYYYY MON')` выдаст ошибку, так как `to_timestamp` ожидает только один пробел. Приставка `FX` должна быть первой в шаблоне.
- Разделитель (пробел или отличный от цифры/буквы символ) в строке шаблона функций `to_timestamp` и `to_date` соответствует любому разделителю во входной строке или пропускается, если только не добавлена приставка `FX`. Например, `to_timestamp('2000JUN', 'YYYY///`

`MON')` и `to_timestamp('2000/JUN', 'YYYY MON')` будут работать, но `to_timestamp('2000//JUN', 'YYYY/MON')` выдаст ошибку, так как количество разделителей во входной строке превышает количество разделителей в шаблоне.

Если добавляется приставка `FX`, разделитель в строке шаблона соответствует ровно одному символу во входной строке. Но заметьте, что символ во входной строке не обязательно должен совпадать с символом разделителя в шаблоне. Например, `to_timestamp('2000/JUN', 'FXYYYY MON')` будет работать, а `to_timestamp('2000/JUN', 'FXYYYY  MON')` выдаст ошибку, потому что второй пробел в строке шаблона забирает букву `J` из входной строки.

- Коду шаблона `TZH` может соответствовать число со знаком. Без приставки `FX` знаки минуса могут быть неоднозначными и восприниматься как разделители. Эта неоднозначность разрешается следующим образом: если число разделителей перед `TZH` в строке шаблона меньше числа разделителей перед знаком минуса во входной строке, знак минус воспринимается как относящийся к `TZH`. В противном случае знак минуса воспринимается как разделитель значений. Например, в `to_timestamp('2000 -10', 'YYYY TZH')` в поле `TZH` попадает `-10`, а в `to_timestamp('2000 -10', 'YYYY  TZH')` в `TZH` попадает значение `10`.
- Шаблоны для `to_char` могут содержать обычный текст; он будет выведен в неизменном виде. Чтобы принудительно вывести текст буквально, даже если он содержит шаблонные коды, подстроку с ним можно заключить в кавычки. Например, в строке `'"Hello Year "YYYY'`, код `YYYY` будет заменён годом, а буква `Y` в слове `Year` останется неизменной. В функциях `to_date`, `to_number` и `to_timestamp` при обработке подстрок в кавычках и буквального текста некоторой длины пропускается такое же число символов во входной строке; например, при обработке подстроки `"XX"` будут пропущены два символа (любые, не обязательно `XX`).

Подсказка

До Postgres Pro 12 во входной строке можно было пропускать произвольный текст, используя в шаблоне символы, отличные от цифр и букв. Например, раньше работало `to_timestamp('2000y6m1d', 'yyyy-MM-DD')`. Теперь для этой цели можно использовать только буквы. Например, шаблоны `to_timestamp('2000y6m1d', 'yyyytMMtDDt')` и `to_timestamp('2000y6m1d', 'yyyy"y"MM"m"DD"d"')` пропускают `y`, `m` и `d`.

- Если вам нужно получить на выходе двойные кавычки, вы должны предварить их символом обратной косой черты, например: `'\"YYYY Month\"'`. В остальном этот символ вне кавычек воспринимается как обычный. Внутри строки в кавычках он указывает, что следующий символ должен восприниматься буквально, каким бы он ни был (но это имеет смысл, только если следующий символ — кавычки или обратная косая черта).
- Если в функциях `to_timestamp` и `to_date` формат года определяется менее, чем 4 цифрами, например, как `YYY`, и в переданном значении года тоже меньше 4 цифр, год пересчитывается в максимально близкий к году 2020, т. е. `95` воспринимается как 1995.
- Функции `to_timestamp` и `to_date` воспринимают отрицательные значения годов как относящиеся к годам до н. э. Если же указать отрицательное значение и добавить явный признак `BC` (до н. э.), год будет относиться к н. э. Нулевое значение года воспринимается как 1 год до н. э.
- В функциях `to_timestamp` и `to_date` с преобразованием `YYYY` связано ограничение, когда обрабатываемый год записывается более чем 4 цифрами. После `YYYY` необходимо будет добавить нецифровой символ или соответствующий код, иначе год всегда будет восприниматься как 4 цифры. Например, в `to_date('200001130', 'YYYYMMDD')` (с годом 20000) год будет интерпретирован как состоящий из 4 цифр; чтобы исправить ситуацию, нужно добавить нецифровой разделитель после года, как в `to_date('20000-1130', 'YYYY-MMDD')`, или код как в `to_date('20000Nov30', 'YYYYMonDD')`.
- Функции `to_timestamp` и `to_date` принимают поле `CC` (век), но игнорируют его, если в шаблоне есть поле `YYY`, `YYYY` или `Y,YYY`. Если `CC` используется с `YY` или `Y`, результатом будет год в данном столетии. Если присутствует только код столетия, без года, подразумевается первый год этого века.

- Функции `to_timestamp` и `to_date` принимают названия и номера дней недели (`DAY`, `D` и связанные типы полей), но игнорируют их при вычислении результата. То же самое происходит с полями квартала (`Q`).
- Функциям `to_timestamp` и `to_date` можно передать даты по недельному календарю ISO 8601 (отличающиеся от григорианских) одним из двух способов:
  - Год, номер недели и дня недели: например, `to_date('2006-42-4', 'IYYY-IW-ID')` возвращает дату `2006-10-19`. Если день недели опускается, он считается равным 1 (понедельнику).
  - Год и день года: например, `to_date('2006-291', 'IYYY-IDDD')` также возвращает `2006-10-19`.

  Попытка ввести дату из смеси полей григорианского и недельного календаря ISO 8601 бессмысленна, поэтому это будет считаться ошибкой. В контексте ISO 8601 понятия «номер месяца» и «день месяца» не существуют, а в григорианском календаре нет понятия номера недели по ISO.

Внимание

Тогда как `to_date` не примет смесь полей григорианского и недельного календаря ISO, `to_char` способна на это, так как форматы вроде `YYYY-MM-DD (IYYY-IDDD)` могут быть полезны. Но избегайте форматов типа `IYYY-MM-DD`; в противном случае с датами в начале года возможны сюрпризы. (За дополнительными сведениями обратитесь к Подразделу 9.9.1.)

- Функция `to_timestamp` воспринимает поля миллисекунд (`MS`) или микросекунд (`US`) как дробную часть число секунд. Например, `to_timestamp('12.3', 'SS.MS')` — это не 3 миллисекунды, а 300, так как это значение воспринимается как 12 + 0.3 секунды. Это значит, что для формата `SS.MS` входные значения `12.3`, `12.30` и `12.300` задают одно и то же число миллисекунд. Чтобы получить три миллисекунды, время нужно записать в виде `12.003`, тогда оно будет воспринято как 12 + 0.003 = 12.003 сек.

  Ещё более сложный пример: `to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US')` будет преобразовано в 15 часов, 12 минут и 2 секунды + 20 миллисекунд + 1230 микросекунд = 2.021230 seconds.
- Нумерация дней недели в `to_char(..., 'ID')` соответствует функции `extract(isodow from ...)`, но нумерация `to_char(..., 'D')` не соответствует нумерации, принятой в `extract(dow from ...)`.
- Функция `to_char(interval)` обрабатывает форматы `HH` и `HH12` в рамках 12 часов, то есть 0 и 36 часов будут выводиться как `12`, тогда как `HH24` выводит число часов полностью, и для значений `interval` результат может превышать 23.

Коды форматирования числовых значений перечислены в Таблице 9.29.

Таблица 9.29. Коды форматирования чисел

Код	Описание
`9`	позиция цифры (может отсутствовать, если цифра незначащая)
`0`	позиция цифры (присутствует всегда, даже если цифра незначащая)
`.` (точка)	десятичная точка
`,` (запятая)	разделитель групп (тысяч)
`PR`	отрицательное значение в угловых скобках

Код	**Описание**
`S`	знак, добавляемый к числу (с учётом локали)
`L`	символ денежной единицы (с учётом локали)
`D`	разделитель целой и дробной части числа (с учётом локали)
`G`	разделитель групп (с учётом локали)
`MI`	знак минус в заданной позиции (если число < 0)
`PL`	знак плюс в заданной позиции (если число > 0)
`SG`	знак плюс или минус в заданной позиции
`RN`	число римскими цифрами (в диапазоне от 1 до 3999)
`TH` или `th`	окончание порядкового числительного
`V`	сдвиг на заданное количество цифр (см. замечания)
`EEEE`	экспоненциальная запись числа

Замечания по использованию форматов чисел:

- `0` обозначает позицию цифры, которая будет выводиться всегда, даже если это незначащий ноль слева или справа. `9` также обозначает позицию цифры, но если это незначащий ноль слева, он заменяется пробелом, а если справа и задан режим заполнения, он удаляется. (Для функции `to_number()` эти два символа равнозначны.)

- Если в формате числа предусмотрено меньше знаков после запятой, чем в форматируемом числе, функция `to_char()` округлит число до указанного количества знаков.

- Символы шаблона `S`, `L`, `D` и `G` представляют знак, символ денежной единицы, десятичную точку и разделитель тысяч, как их определяет текущая локаль (см. lc_monetary и lc_numeric). Символы точка и запятая представляют те же символы, обозначающие десятичную точку и разделитель тысяч, но не зависят от локали.

- Если в шаблоне `to_char()` отсутствует явное указание положения знака, для него резервируется одна позиция рядом с числом (слева от него). Если левее нескольких `9` помещён `S`, знак также будет приписан слева к числу.

- Знак числа, полученный кодами `SG`, `PL` или `MI`, не присоединяется к числу; например, `to_char(-12, 'MI9999')` выдаёт `'-   12'`, тогда как `to_char(-12, 'S9999')` — `'  -12'`. (В Oracle `MI` не может идти перед `9`, наоборот `9` нужно указать перед `MI`.)

- `TH` не преобразует значения меньше 0 и не поддерживает дробные числа.

- `PL`, `SG` и `TH` — расширения Postgres Pro.

- В `to_number` при использовании шаблонных кодов, не обозначающих данные, таких как `L` и `TH`, пропускается соответствующее количество входных символов. При этом не имеет значения, совпадают ли они с символами шаблона, если только это не символы данных (то есть цифры, знак числа, десятичная точка или запятая). Например, для подстроки `TH` будут пропущены два символа, не представляющие данные.

- `V` с `to_char` умножает вводимое значение на `10^n`, где *n* — число цифр, следующих за `V`. `V` с `to_number` подобным образом делит значение. Функции `to_char` и `to_number` не поддерживают `V` с дробными числами (например, `99.9V99` не допускается).

- Код `EEEE` (научная запись) не может сочетаться с любыми другими вариантами форматирования или модификаторами, за исключением цифр и десятичной точки, и должен располагаться в конце строки шаблона (например, `9.99EEEE` — допустимый шаблон).

Для изменения поведения кодов к ним могут быть применены определённые модификаторы. Например, `FM99.99` обрабатывается как код `99.99` с модификатором `FM`. Все модификаторы для форматирования чисел перечислены в Таблице 9.30.

Таблица 9.30. Модификаторы шаблонов для форматирования чисел

Модификатор	Описание	Пример
Приставка `FM`	режим заполнения (подавляет завершающие нули и дополнение пробелами)	`FM99.99`
Окончание `TH`	окончание порядкового числительного в верхнем регистре	`999TH`
Окончание `th`	окончание порядкового числительного в нижнем регистре	`999th`

В Таблице 9.31 приведены некоторые примеры использования функции `to_char`.

Таблица 9.31. Примеры `to_char`

Выражение	Результат
`to_char(current_timestamp, 'Day, DD  HH12:MI:SS')`	`'Tuesday  , 06  05:39:18'`
`to_char(current_timestamp, 'FMDay, FMDD  HH12:MI:SS')`	`'Tuesday, 6  05:39:18'`
`to_char(current_timestamp AT TIME ZONE 'UTC', 'YYYY-MM-DD"T"HH24:MI:SS"Z"')`	`'2022-12-06T05:39:18Z'`, расширенный формат ISO 8601
`to_char(-0.1, '99.99')`	`'  -.10'`
`to_char(-0.1, 'FM9.99')`	`'-.1'`
`to_char(-0.1, 'FM90.99')`	`'-0.1'`
`to_char(0.1, '0.9')`	`' 0.1'`
`to_char(12, '9990999.9')`	`'    0012.0'`
`to_char(12, 'FM9990999.9')`	`'0012.'`
`to_char(485, '999')`	`' 485'`
`to_char(-485, '999')`	`'-485'`
`to_char(485, '9 9 9')`	`' 4 8 5'`
`to_char(1485, '9,999')`	`' 1,485'`
`to_char(1485, '9G999')`	`' 1 485'`
`to_char(148.5, '999.999')`	`' 148.500'`
`to_char(148.5, 'FM999.999')`	`'148.5'`
`to_char(148.5, 'FM999.990')`	`'148.500'`
`to_char(148.5, '999D999')`	`' 148,500'`
`to_char(3148.5, '9G999D999')`	`' 3 148,500'`
`to_char(-485, '999S')`	`'485-'`
`to_char(-485, '999MI')`	`'485-'`
`to_char(485, '999MI')`	`'485 '`
`to_char(485, 'FM999MI')`	`'485'`
`to_char(485, 'PL999')`	`'+485'`
`to_char(485, 'SG999')`	`'+485'`

Выражение	**Результат**
`to_char(-485, 'SG999')`	`'-485'`
`to_char(-485, '9SG99')`	`'4-85'`
`to_char(-485, '999PR')`	`'<485>'`
`to_char(485, 'L999')`	`'DM 485'`
`to_char(485, 'RN')`	`'        CDLXXXV'`
`to_char(485, 'FMRN')`	`'CDLXXXV'`
`to_char(5.2, 'FMRN')`	`'V'`
`to_char(482, '999th')`	`' 482nd'`
`to_char(485, '"Good number:"999')`	`'Good number: 485'`
`to_char(485.8,` `'"Pre:"999" Post:" .999')`	`'Pre: 485 Post: .800'`
`to_char(12, '99V999')`	`' 12000'`
`to_char(12.4, '99V999')`	`' 12400'`
`to_char(12.45, '99V9')`	`' 125'`
`to_char(0.0004859, '9.99EEEE')`	`' 4.86e-04'`

9.9. Операторы и функции даты/времени

Все существующие функции для обработки даты/времени перечислены в Таблице 9.33, а подробнее они описаны в следующих подразделах. Поведение основных арифметических операторов (`+`, `*` и т. д.) описано в Таблице 9.32. Функции форматирования этих типов данных были перечислены в Разделе 9.8. Общую информацию об этих типах вы получили (или можете получить) в Разделе 8.5.

Помимо этого, для типов даты/времени имеются обычные операторы сравнения, показанные в Таблице 9.1. Значения даты и даты со временем (с часовым поясом или без него) можно сравнивать как угодно, тогда как значения только времени (с часовым поясом или без него) и интервалы допустимо сравнивать, только если их типы совпадают. При сравнении даты со временем без часового пояса и даты со временем с часовым поясом предполагается, что первое значение задано в часовом поясе, установленном параметром TimeZone, и оно пересчитывается в UTC для сравнения со вторым значением (внутри уже представленным в UTC). Аналогичным образом, при сравнении значений даты и даты со времени первое считается соответствующим полночи в часовом поясе `TimeZone`.

Все описанные ниже функции и операторы, принимающие аргументы `time` или `timestamp`, фактически представлены в двух вариациях: одна принимает тип `time with time zone` или `timestamp with time zone`, а вторая — `time without time zone` или `timestamp without time zone`. Для краткости эти вариации здесь не разделяются. Кроме того, операторы `+` и `*` определяются парами, наделяющими их переместительным свойством (например, `date + integer` и `integer + date`); здесь приводится только один вариант для каждой пары.

Таблица 9.32. Операторы даты/времени

Оператор **Описание** **Пример(ы)**

`date + integer → date`
Добавляет к дате заданное число дней
`date '2001-09-28' + 7 → 2001-10-05`

`date + interval → timestamp` Добавляет к дате интервал

Оператор **Описание** **Пример(ы)**
date '2001-09-28' + interval '1 hour' → 2001-09-28 01:00:00
date + time → timestamp Добавляет к дате время date '2001-09-28' + time '03:00' → 2001-09-28 03:00:00
interval + interval → interval Складывает интервалы interval '1 day' + interval '1 hour' → 1 day 01:00:00
timestamp + interval → timestamp Добавляет к отметке времени интервал timestamp '2001-09-28 01:00' + interval '23 hours' → 2001-09-29 00:00:00
time + interval → time Добавляет к времени интервал time '01:00' + interval '3 hours' → 04:00:00
- interval → interval Меняет направление интервала - interval '23 hours' → -23:00:00
date - date → integer Вычитает даты, выдавая разницу в днях date '2001-10-01' - date '2001-09-28' → 3
date - integer → date Вычитает из даты заданное число дней date '2001-10-01' - 7 → 2001-09-24
date - interval → timestamp Вычитает из даты интервал date '2001-09-28' - interval '1 hour' → 2001-09-27 23:00:00
time - time → interval Вычитает из одного времени другое time '05:00' - time '03:00' → 02:00:00
time - interval → time Вычитает из времени интервал time '05:00' - interval '2 hours' → 03:00:00
timestamp - interval → timestamp Вычитает из отметки времени интервал timestamp '2001-09-28 23:00' - interval '23 hours' → 2001-09-28 00:00:00
interval - interval → interval Вычитает из одного интервала другой interval '1 day' - interval '1 hour' → 1 day -01:00:00
timestamp - timestamp → interval Вычитает из одной отметки времени другую (преобразуя 24-часовые интервалы в дни подобно justify_hours()) timestamp '2001-09-29 03:00' - timestamp '2001-07-27 12:00' → 63 days 15:00:00
interval * double precision → interval

Оператор **Описание** **Пример(ы)**

Умножает интервал на скалярное значение
`interval '1 second' * 900` → `00:15:00`
`interval '1 day' * 21` → `21 days`
`interval '1 hour' * 3.5` → `03:30:00`

`interval` / `double precision` → `interval`
Делит интервал на скалярное значение
`interval '1 hour' / 1.5` → `00:40:00`

Таблица 9.33. Функции даты/времени

Функция **Описание** **Пример(ы)**

`age` (`timestamp`, `timestamp`) → `interval`
Вычитает аргументы и выдаёт «символический» результат с годами и месяцами, а не просто днями
`age(timestamp '2001-04-10', timestamp '1957-06-13')` → `43 years 9 mons 27 days (43 года 9 месяцев 27 дней)`

`age` (`timestamp`) → `interval`
Вычитает аргумент из `current_date` (полночь текущего дня)
`age(timestamp '1957-06-13')` → `62 years 6 mons 10 days (62 года 6 месяцев 10 дней)`

`clock_timestamp` ( ) → `timestamp with time zone`
Текущая дата и время (меняется в процессе выполнения операторов); см. Подраздел 9.9.5
`clock_timestamp()` → `2019-12-23 14:39:53.662522-05`

`current_date` → `date`
Текущая дата; см. Подраздел 9.9.5
`current_date` → `2019-12-23`

`current_time` → `time with time zone`
Текущее время суток; см. Подраздел 9.9.5
`current_time` → `14:39:53.662522-05`

`current_time` (`integer`) → `time with time zone`
Текущее время суток (с ограниченной точностью); см. Подраздел 9.9.5
`current_time(2)` → `14:39:53.66-05`

`current_timestamp` → `timestamp with time zone`
Текущая дата и время (на момент начала транзакции); см. Подраздел 9.9.5
`current_timestamp` → `2019-12-23 14:39:53.662522-05`

`current_timestamp` (`integer`) → `timestamp with time zone`
Текущие дата и время (на момент начала транзакции; с ограниченной точностью); см. Подраздел 9.9.5
`current_timestamp(0)` → `2019-12-23 14:39:53-05`

`date_add` (`timestamp with time zone`, `interval` [, `text`]) → `timestamp with time zone` Добавляет значение типа `interval` к значению типа `timestamp with time zone`, вычисляя время суток и корректировки перехода на летнее время в соответствии с часовым поясом, указанным в третьем аргументе, или, если он опущен, с текущим параметром

Функция **Описание** **Пример(ы)**
TimeZone. Форма с двумя аргументами эквивалентна оператору `timestamp with time zone` + `interval`. `date_add('2021-10-31 00:00:00+02'::timestamptz, '1 day'::interval, 'Europe/Warsaw')` → `2021-10-31 23:00:00+00`

`date_bin` (`interval`, `timestamp`, `timestamp`) → `timestamp`
Подгоняет заданное значение под интервал, отсчитывая от указанного начального момента; см. Подраздел 9.9.3
`date_bin('15 minutes', timestamp '2001-02-16 20:38:40', timestamp '2001-02-16 20:05:00')` → `2001-02-16 20:35:00`

`date_part` (`text`, `timestamp`) → `double precision`
Возвращает поле даты/времени (равнозначно `extract`); см. Подраздел 9.9.1
`date_part('hour', timestamp '2001-02-16 20:38:40')` → `20`

`date_part` (`text`, `interval`) → `double precision`
Возвращает поле интервала (равнозначно `extract`); см. Подраздел 9.9.1
`date_part('month', interval '2 years 3 months')` → `3`

`date_subtract` (`timestamp with time zone`, `interval` [, `text`]) → `timestamp with time zone`
Вычитает значение типа `interval` из значения типа `timestamp with time zone`, вычисляя время суток и корректировки перехода на летнее время в соответствии с часовым поясом, указанным в третьем аргументе, или текущим параметром TimeZone, если он опущен. Форма с двумя аргументами эквивалентна оператору `timestamp with time zone` - `interval`.
`date_subtract('2021-11-01 00:00:00+01'::timestamptz, '1 day'::interval, 'Europe/Warsaw')` → `2021-10-30 22:00:00+00`

`date_trunc` (`text`, `timestamp`) → `timestamp`
Отсекает компоненты даты до заданной точности; см. Подраздел 9.9.2
`date_trunc('hour', timestamp '2001-02-16 20:38:40')` → `2001-02-16 20:00:00`

`date_trunc` (`text`, `timestamp with time zone`, `text`) → `timestamp with time zone`
Отсекает компоненты даты до заданной точности в указанном часовом поясе; см. Подраздел 9.9.2
`date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney')` → `2001-02-16 13:00:00+00`

`date_trunc` (`text`, `interval`) → `interval`
Отсекает компоненты даты до заданной точности; см. Подраздел 9.9.2
`date_trunc('hour', interval '2 days 3 hours 40 minutes')` → `2 days 03:00:00`

`extract` (*`field`* `from timestamp`) → `numeric`
Возвращает поле даты/времени; см. Подраздел 9.9.1
`extract(hour from timestamp '2001-02-16 20:38:40')` → `20`

`extract` (*`field`* `from interval`) → `numeric`
Возвращает поле интервала; см. Подраздел 9.9.1
`extract(month from interval '2 years 3 months')` → `3`

`isfinite` (`date`) → `boolean`
Проверяет конечность даты (её отличие от +/-бесконечности)
`isfinite(date '2001-02-16')` → `true`

`isfinite` (`timestamp`) → `boolean`

Функция **Описание** **Пример(ы)**
Проверяет конечность времени (его отличие от +/-бесконечности) `isfinite(timestamp 'infinity')` → `false`

`isfinite` (`interval`) → `boolean`
Проверяет конечность интервала (его отличие от +/-бесконечности)
`isfinite(interval '4 hours')` → `true`

`justify_days` (`interval`) → `interval`
Преобразует интервал так, что каждый 30-дневный период считается одним месяцем
`justify_days(interval '1 year 65 days')` → `1 year 2 mons 5 days`

`justify_hours` (`interval`) → `interval`
Преобразует интервал так, что каждый 24-часовой период считается одним днём
`justify_hours(interval '50 hours 10 minutes')` → `2 days 02:10:00`

`justify_interval` (`interval`) → `interval`
Преобразует интервал с применением `justify_days` и `justify_hours` и дополнительно корректирует знаки
`justify_interval(interval '1 mon -1 hour')` → `29 days 23:00:00 (29 дней 23:00:00)`

`localtime` → `time`
Текущее время суток; см. Подраздел 9.9.5
`localtime` → `14:39:53.662522`

`localtime` (`integer`) → `time`
Текущее время суток (с ограниченной точностью); см. Подраздел 9.9.5
`localtime(0)` → `14:39:53`

`localtimestamp` → `timestamp`
Текущая дата и время (на момент начала транзакции); см. Подраздел 9.9.5
`localtimestamp` → `2019-12-23 14:39:53.662522`

`localtimestamp` (`integer`) → `timestamp`
Текущие дата и время (на момент начала транзакции; с ограниченной точностью); см. Подраздел 9.9.5
`localtimestamp(2)` → `2019-12-23 14:39:53.66`

`make_date` (*year* `int`, *month* `int`, *day* `int`) → `date`
Образует дату из полей: year (год), month (месяц) и day (день) (отрицательное значение поля year означает год до н. э.)
`make_date(2013, 7, 15)` → `2013-07-15`

`make_interval` ([*years* `int` [, *months* `int` [, *weeks* `int` [, *days* `int` [, *hours* `int` [, *mins* `int` [, *secs* `double precision`]]]]]]]) → `interval`
Образует интервал из полей: years (годы), months (месяцы), weeks (недели), days (дни), hours (часы), minutes (минуты) и secs (секунды), каждое из которых по умолчанию считается равным нулю.
`make_interval(days => 10)` → `10 days`

`make_time` (*hour* `int`, *min* `int`, *sec* `double precision`) → `time`
Образует время из полей: hour (час), minute (минута) и sec (секунда)
`make_time(8, 15, 23.5)` → `08:15:23.5`

`make_timestamp` (*year* `int`, *month* `int`, *day* `int`, *hour* `int`, *min* `int`, *sec* `double precision`) → `timestamp`

Функция **Описание** **Пример(ы)**
Образует момент времени из полей: year (год), month (месяц), day (день), hour (час), minute (минута) и sec (секунда) (отрицательное значение поля year означает год до н. э.) `make_timestamp(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5`

`make_timestamptz` (*year* `int`, *month* `int`, *day* `int`, *hour* `int`, *min* `int`, *sec* `double precision` [, *timezone* `text`]) → `timestamp with time zone`
Образует дату и время с часовым поясом из полей: year (год), month (месяц), day (день), hour (час), minute (минута) и sec (секунда) (отрицательное значение поля year означает год до н. э.). Если параметр *timezone* (часовой пояс) не указан, используется текущий часовой пояс; в примерах предполагается часовой пояс `Europe/London` (Европа/Лондон).
`make_timestamptz(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5+01`
`make_timestamptz(2013, 7, 15, 8, 15, 23.5, 'America/New_York')` → `2013-07-15 13:15:23.5+01`

`now` ( ) → `timestamp with time zone`
Текущая дата и время (на момент начала транзакции); см. Подраздел 9.9.5
`now()` → `2019-12-23 14:39:53.662522-05`

`statement_timestamp` ( ) → `timestamp with time zone`
Текущая дата и время (на момент начала текущего оператора); см. Подраздел 9.9.5
`statement_timestamp()` → `2019-12-23 14:39:53.662522-05`

`timeofday` ( ) → `text`
Текущая дата и время (как `clock_timestamp` , но в виде строки типа `text`); см. Подраздел 9.9.5
`timeofday()` → `Mon Dec 23 14:39:53.662522 2019 EST`

`transaction_timestamp` ( ) → `timestamp with time zone`
Текущая дата и время (на момент начала транзакции); см. Подраздел 9.9.5
`transaction_timestamp()` → `2019-12-23 14:39:53.662522-05`

`to_timestamp` (`double precision`) → `timestamp with time zone`
Преобразует время эпохи Unix (число секунд с 1970-01-01 00:00:00+00) в дату/время с часовым поясом
`to_timestamp(1284352323)` → `2010-09-13 04:32:03+00`

В дополнение к этим функциям поддерживается SQL-оператор `OVERLAPS`:

```
(начало1, конец1) OVERLAPS (начало2, конец2)
(начало1, длительность1) OVERLAPS (начало2, длительность2)
```

Его результатом будет true, когда два периода времени (определённые своими границами) пересекаются, и false в противном случае. Границы периода можно задать либо в виде пары дат, времени или дат со временем, либо как дату, время (или дату со временем) с интервалом. Когда указывается пара значений, первым может быть и начало, и конец периода: `OVERLAPS` автоматически считает началом периода меньшее значение. Периоды времени считаются наполовину открытыми, т. е. *начало*`<=`*время*`<`*конец*, если только *начало* и *конец* не равны — в этом случае период представляет один момент времени. Это означает, например, что два периода, имеющие только общую границу, не будут считаться пересекающимися.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Результат:true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Результат:false
```

```
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Результат:false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Результат:true
```

При добавлении к метке типа `timestamp` или `timestamp with time zone` значения `interval` (или при вычитании из него `interval`), поля месяцев, дней и микросекунд обрабатываются по очереди. Сначала дата в метке времени изменяется на указанное количество месяцев, сохраняя день неизменным, если он не превышает количество дней в новом месяце, иначе будет задан последний день нового месяца. (Например, 31 марта плюс 1 месяц становится 30 апреля, а 31 марта плюс 2 месяца — 31 мая.) Затем дата изменяется на указанное количество дней. На обоих этих этапах местное время суток остаётся неизменным. Наконец, если указано ненулевое значение микросекунд, оно добавляется или вычитается буквально. При выполнении арифметических действий со значением типа `timestamp with time zone` в часовом поясе с переходом на летнее время это означает, что добавление или вычитание (например) `interval '1 day'` не обязательно приведёт к тому же результату, что и добавление или вычитание `interval '24 hours'`. Например, в часовом поясе `America/Denver`:

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '1 day';
Результат: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '24 hours';
Результат: 2005-04-03 13:00:00-06
```

Эта разница объясняется тем, что `2005-04-03 02:00` в часовом поясе `America/Denver` произошёл переход на летнее время.

Обратите внимание на возможную неоднозначность в поле `months` в результате функции `age`, вызванную тем, что число дней в разных месяцах неодинаково. Вычисляя оставшиеся дни месяца, Postgres Pro рассматривает месяц меньшей из двух дат. Например, результатом `age('2004-06-01', '2004-04-30')` будет `1 mon 1 day`, так как в апреле 30 дней, а то же выражение с датой 30 мая выдаст `1 mon 2 days`, так как в мае 31 день.

Вычитание дат и дат со временем также может быть нетривиальной операцией. Один принципиально простой способ выполнить такое вычисление — преобразовать каждое значение в количество секунд, используя `EXTRACT(EPOCH FROM ...)`, а затем найти разницу результатов; при этом будет получено число *секунд* между двумя датами. При этом будет учтено неодинаковое число дней в месяцах, изменения часовых поясов и переходы на летнее время. При вычитании дат или дат со временем с помощью оператора «`-`» выдаётся число дней (по 24 часа) и часов/минут/секунд между данными значениями, с учётом тех же факторов. Функция `age` возвращает число лет, месяцев, дней и часов/минут/секунд, выполняя вычитание по полям, а затем пересчитывая отрицательные значения. Различие этих подходов иллюстрируют следующие запросы. Показанные результаты были получены для часового пояса `'US/Eastern'`; между двумя заданными датами произошёл переход на летнее время:

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
Результат:10537200.000000
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
        / 60 / 60 / 24;
Результат:121.9583333333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';
Результат:121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01 12:00:00');
Результат:4 mons
```

9.9.1. EXTRACT, date_part

```
EXTRACT(поле FROM источник)
```

Функция `extract` извлекает поля, такие как год или час, из значений даты и времени. *`Источником`* должно быть заданное выражением значение типа `timestamp`, `date`, `time` или `interval`. (Значения `timestamp` и `time` могут быть как с указанием часового пояса, так и без.) *`Поле`* — это идентификатор или строка, с помощью которой выбирается поле для извлечения из значения источника. Для разных типов входных данных допустимы разные поля: например, поля меньше дней не могут быть извлечены из `date`, а поля дней и более не могут быть извлечены из `time`. Функция `extract` возвращает значения типа `numeric`.

Допустимы следующие имена полей:

`century`

Век; для значений `interval` — значение поля года, разделённое на сто

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Результат:20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:21
SELECT EXTRACT(CENTURY FROM DATE '0001-01-01 AD');
Результат:1
SELECT EXTRACT(CENTURY FROM DATE '0001-12-31 BC');
Результат:-1
SELECT EXTRACT(CENTURY FROM INTERVAL '2001 years');
Результат:20
```

`day`

День месяца (1–31); для значений `interval` — количество дней

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:16
SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
Результат:40
```

`decade`

Год, делённый на 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:200
```

`dow`

День недели, считая с воскресенья (`0`) до субботы (`6`)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:5
```

Заметьте, что в `extract` дни недели нумеруются не так, как в функции `to_char(..., 'D')`.

`doy`

День года (1–365/366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:47
```

`epoch`

Для значений `timestamp with time zone` это число секунд с 1970-01-01 00:00:00 UTC (отрицательное для предшествующего времени); для значений `date` и `timestamp` — номинальное число секунд с 1970-01-01 00:00:00 без учёта часового пояса, переходов на летнее время и т. п.; для значений `interval` — общее количество секунд в интервале

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
Результат:982384720.120000
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40.12');
Результат:982355920.120000
```

```
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Результат:442800.000000
```

Преобразовать время эпохи назад, в значение `timestamp with time zone`, с помощью `to_timestamp` можно так:

```
SELECT to_timestamp(982384720.12);
Результат:2001-02-17 04:38:40.12+00
```

Имейте в виду, что применяя `to_timestamp` к времени эпохи, извлечённому из значения `date` или `timestamp`, можно получить не вполне ожидаемый результат: эта функция подразумевает, что изначальное значение задано в часовом поясе UTC, но это может быть не так.

`hour`

Поле часов (0–23 в метках времени, в интервалах — без ограничений)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:20
```

`isodow`

День недели, считая с понедельника (`1`) до воскресенья (`7`)

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Результат:7
```

Результат отличается от `dow` только для воскресенья. Такая нумерация соответствует ISO 8601.

`isoyear`

Год с нумерацией недель согласно ISO 8601, на который приходится дата

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Результат:2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Результат:2006
```

Год по недельному календарю ISO начинается с понедельника недели, в которой оказывается 4 января, так что в начале января или в конце декабря год по ISO может отличаться от года по григорианскому календарю. Подробнее об этом рассказывается в описании поля `week`.

`julian`

Юлианская дата, соответствующая дате или дате/времени. Значение будет дробным, если заданное время отличается от начала суток по местному времени. За дополнительной информацией обратитесь к Разделу B.7.

```
SELECT EXTRACT(JULIAN FROM DATE '2006-01-01');
Результат:2453737
SELECT EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00');
Результат:2453737.50000000000000000000
```

`microseconds`

Значение секунд с дробной частью, умноженное на 1 000 000; заметьте, что оно включает и целые секунды

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Результат:28500000
```

`millennium`

Тысячелетие; для значений `interval` — значение поля года, разделённое на тысячу

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:3
SELECT EXTRACT(MILLENNIUM FROM INTERVAL '2001 years');
Результат:2
```

Годы 20 века относятся ко второму тысячелетию. Третье тысячелетие началось 1 января 2001 г.

`milliseconds`

Значение секунд с дробной частью, умноженное на 1 000; заметьте, что оно включает и целые секунды.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Результат:28500.000
```

`minute`

Минуты (0–59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:38
```

`month`

Номер месяца в году (1–12), а для `interval` — остаток от деления числа месяцев на 12 (0–11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:2
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Результат:3
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Результат:1
```

`quarter`

Квартал (1–4), к которому относится дата

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:1
```

`second`

Секунды, включая дробную часть

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:40.000000
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Результат:28.500000
```

`timezone`

Смещение часового пояса от UTC, представленное в секундах. Положительные значения соответствуют часовым поясам к востоку от UTC, а отрицательные — к западу. (Строго говоря, в Postgres Pro используется не UTC, так как секунды координации не учитываются.)

`timezone_hour`

Поле часов в смещении часового пояса

`timezone_minute`

Поле минут в смещении часового пояса

`week`

Номер недели в году по недельному календарю ISO 8601. По определению, недели ISO 8601 начинаются с понедельника, а первая неделя года включает 4 января этого года. Другими словами, первый четверг года всегда оказывается в 1 неделе этого года.

В системе нумерации недель ISO первые числа января могут относиться к 52-ой или 53-ей неделе предыдущего года, а последние числа декабря — к первой неделе следующего года. Например, `2005-01-01` относится к 53-ей неделе 2004 г., а `2006-01-01` — к 52-ей неделе 2005 г., тогда как `2012-12-31` включается в первую неделю 2013 г. Поэтому для получения согласованных результатов рекомендуется использовать поле `isoyear` в паре с `week`.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:7
```

`year`

Поле года. Учтите, что года `0` не было, и это следует иметь в виду, вычитая из годов нашей эры годы до нашей эры.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Результат:2001
```

При обработке значения `interval` функция `extract` создаёт значения полей, соответствующие интерпретации, которая используется функцией вывода интервала. При этом, если начать с ненормализованного представления интервала, могут быть получены неожиданные результаты, например:

```
SELECT INTERVAL '80 minutes';
Результат: 01:20:00
SELECT EXTRACT(MINUTES FROM INTERVAL '80 minutes');
Результат: 20
```

Примечание

С аргументом +/-бесконечность `extract` возвращает +/-бесконечность для монотонно увеличивающихся полей (`epoch`, `julian`, `year`, `isoyear`, `decade`, `century` и `millennium` для значений `timestamp`; `epoch`, `hour`, `day`, `year`, `decade`, `century` и `millennium` для значений `interval`). Для других полей возвращается NULL. До версии 9.6 PostgreSQL возвращал ноль для всех случаев с бесконечными аргументами.

Функция `extract` в основном предназначена для вычислительных целей. Функции форматирования даты/времени описаны в Разделе 9.8.

Функция `date_part` эмулирует традиционный для Ingres эквивалент стандартной SQL-функции `extract`:

```
date_part('поле', источник)
```

Заметьте, что здесь параметр *`поле`* должен быть строковым значением, а не именем. Функция `date_part` воспринимает те же поля, что и `extract`. По историческим причинам функция `date_part` возвращает значения типа `double precision`. В некоторых случаях это может привести к потере точности. Поэтому вместо неё рекомендуется использовать функцию `extract`.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Результат:16
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Результат:4
```

9.9.2. date_trunc

Функция `date_trunc` работает подобно `trunc` для чисел.

```
date_trunc(поле, значение [, часовой_пояс ])
```

Здесь *`значение`* — выражение типа `timestamp`, `timestamp with time zone` или `interval`. (Значения типов `date` и `time` автоматически приводятся к типам `timestamp` и `interval`, соответственно.) Параметр *`поле`* определяет, до какой точности обрезать переданное значение. В возвращаемом значении, имеющем также тип `timestamp`, `timestamp with time zone` или `interval`, все поля, менее значимые, чем заданное, будут равны нулю (или одному, если это номер дня или месяца).

Параметр *`поле`* может принимать следующие значения:

```
microseconds
milliseconds
second
```

```
minute
hour
day
week
month
quarter
year
decade
century
millennium
```

Когда входное значение имеет тип `timestamp with time zone`, оно обрезается с учётом заданного часового пояса; например, если обрезать значение до поля `day` (день), в результате будет получена полночь в этом часовом поясе. По умолчанию входное значение обрезается с учётом параметра TimeZone, но дополнительный аргумент *`часовой_пояс`* позволяет выбрать и другой пояс. Название часового пояса может задаваться любым из способов, описанных в Подразделе 8.5.3.

Часовой пояс нельзя задать для значений типа `timestamp without time zone` или `interval`. Такие значения всегда воспринимаются как есть.

Несколько примеров (выбран часовой пояс `America/New_York`):

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Результат: 2001-02-16 20:00:00
SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Результат: 2001-01-01 00:00:00
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00');
Результат: 2001-02-16 00:00:00-05
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00', 'Australia/
Sydney');
Результат: 2001-02-16 08:00:00-05
SELECT date_trunc('hour', INTERVAL '3 days 02:47:33');
Результат: 3 days 02:00:00
```

9.9.3. date_bin

Функция `date_bin` «подгоняет» заданное значение под интервал (*шаг*), отсчитывая от указанного начального момента.

`date_bin(шаг, значение, источник)`

Здесь *`значение`* — выражение типа `timestamp`, `timestamp with time zone`. (Значения типов `date` автоматически приводятся к типу `timestamp`.) В качестве *`шага`* передаётся выражение типа `interval`. Результат этой функции, также имеющий тип `timestamp`, `timestamp with time zone` или `interval`, — начало интервала, под который подгоняется *`значение`*.

Примеры:

```
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17', TIMESTAMP '2001-01-01');
Результат: 2020-02-11 15:30:00
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17', TIMESTAMP '2001-01-01
 00:02:30');
Результат: 2020-02-11 15:32:30
```

В случае целых единиц измерения (1 минута, 1 час и т. д.), она даёт тот же результат, что и аналогичный вызов функции `date_trunc`, с той лишь разницей, что `date_bin` может округлить дату до произвольного интервала.

Параметр *`шаг`* интервала должен быть больше нуля и задаваться в единицах измерения меньше месяца.

9.9.4. AT TIME ZONE и AT LOCAL

Оператор `AT TIME ZONE` преобразует дату/время *без* часового пояса в дату/время *с* часовым поясом и обратно, а также пересчитывает значения `time with time zone` для различных часовых поясов. Его вариации показаны в Таблице 9.34.

Таблица 9.34. Варианты `AT TIME ZONE` и `AT LOCAL`

Оператор **Описание** **Пример(ы)**

`timestamp without time zone` `AT TIME ZONE` *`часовой_пояс`* → `timestamp with time zone`
Переводит значение даты/времени *без* часового пояса в дату/время *с* часовым поясом, в предположении, что входное значение задано в указанном поясе.
`timestamp '2001-02-16 20:38:40' at time zone 'America/Denver'` → `2001-02-17 03:38:40+00`

`timestamp without time zone` `AT LOCAL` → `timestamp with time zone`
Переводит значение даты/времени *без* часового пояса в дату/время *с* часовым поясом, заданным значением `TimeZone` сеанса.
`timestamp '2001-02-16 20:38:40' at local` → `2001-02-17 03:38:40+00`

`timestamp with time zone` `AT TIME ZONE` *`часовой_пояс`* → `timestamp without time zone`
Переводит значение даты/времени *с* часовым поясом в дату/время *без* часового пояса, которое соответствует входному значению в указанном поясе.
`timestamp with time zone '2001-02-16 20:38:40-05' at time zone 'America/Denver'` → `2001-02-16 18:38:40`

`timestamp with time zone` `AT LOCAL` → `timestamp without time zone`
Переводит значение даты/времени *с* часовым поясом в дату/время *без* часового пояса, поскольку время будет отображаться с часовым поясом, указанным в качестве значения `TimeZone` сеанса.
`timestamp with time zone '2001-02-16 20:38:40-05' at local` → `2001-02-16 18:38:40`

`time with time zone` `AT TIME ZONE` *`часовой_пояс`* → `time with time zone`
Переводит значение времени *с* часовым поясом в другой часовой пояс. Так как время задаётся без даты, в расчёте используется действующее в данный момент смещение указанного часового пояса от UTC.
`time with time zone '05:34:17-05' at time zone 'UTC'` → `10:34:17+00`

`time with time zone` `AT LOCAL` → `time with time zone`
Переводит значение времени *с* часовым поясом в другой часовой пояс. Так как время задаётся без даты, в расчёте используется действующее в данный момент смещение значения `TimeZone` сеанса от UTC.
Предположим, что для `TimeZone` сеанса установлено значение `UTC`:
`time with time zone '05:34:17-05' at local` → `10:34:17+00`

В этих выражениях желаемый *`часовой_пояс`* можно задать либо в виде текстовой строки (например, `'America/Los_Angeles'`), либо как интервал (например, `INTERVAL '-08:00'`). В первом случае название часового пояса можно указать любым из способов, описанных в Подразделе 8.5.3. Вариант с интервалом полезен, только если для часового пояса смещение от UTC всегда постоянно, что на практике встречается нечасто.

Синтаксис `AT LOCAL` можно использовать в качестве сокращённой записи для `AT TIME ZONE` *`локальный`*, где *`локальный`* — это значение `TimeZone` сеанса.

Примеры (предполагается, что текущее значение параметра TimeZone — `America/Los_Angeles`):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
Результат: 2001-02-16 19:38:40-08
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
```

```
Результат: 2001-02-16 18:38:40
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/
Chicago';
Результат: 2001-02-16 05:38:40
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT LOCAL;
Результат: 2001-02-16 17:38:40
SELECT TIME WITH TIME ZONE '20:38:40-05' AT LOCAL;
Результат: 17:38:40
```

В первом примере для значения, заданного без часового пояса, указывается часовой пояс и полученное время выводится в текущем часовом поясе (заданном параметром `TimeZone`). Во втором примере значение времени смещается в заданный часовой пояс и выдаётся без указания часового пояса. Это позволяет хранить и выводить значения с часовым поясом, отличным от текущего. В третьем примере время в часовом поясе Токио преобразуется для часового пояса Чикаго. В четвёртом примере временная метка со значением часового пояса смещается до часового пояса, указанного текущим значением параметра `TimeZone`, и возвращает значение без часового пояса.

Пятый пример служит в качестве предостережения. Поскольку с входным значением не связана дата, преобразование выполняется с использованием текущей даты сеанса. Поэтому этот статический пример может показывать неверный результат в зависимости от времени года, поскольку в часовом поясе `'America/Los_Angeles'` есть переход на летнее время.

Функция `timezone(`*`часовой_пояс`*`, `*`время`*`)` равнозначна SQL-совместимой конструкции *`время`* `AT TIME ZONE` *`часовой_пояс`*.

Функция `timezone(`*`часовой_пояс`*`, `*`время`*`)` равнозначна SQL-совместимой конструкции *`время`* `AT TIME ZONE` *`часовой_пояс`*.

Функция `timezone(`*`timestamp`*`)` равнозначна SQL-совместимой конструкции *`timestamp`* `AT LOCAL`.

Функция `timezone(`*`время`*`)` равнозначна SQL-совместимой конструкции *`время`* `AT LOCAL`.

9.9.5. Текущая дата/время

Postgres Pro предоставляет набор функций, результат которых зависит от текущей даты и времени. Все следующие функции соответствуют стандарту SQL и возвращают значения, отражающие время начала текущей транзакции:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(точность)
CURRENT_TIMESTAMP(точность)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(точность)
LOCALTIMESTAMP(точность)
```

`CURRENT_TIME` и `CURRENT_TIMESTAMP` возвращают время с часовым поясом. В результатах `LOCALTIME` и `LOCALTIMESTAMP` нет информации о часовом поясе.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME` и `LOCALTIMESTAMP` могут принимать необязательный параметр точности, определяющий, до какого знака после запятой следует округлять поле секунд. Если этот параметр отсутствует, результат будет иметь максимально возможную точность.

Несколько примеров:

```
SELECT CURRENT_TIME;
Результат: 14:39:53.662522-05
SELECT CURRENT_DATE;
Результат: 2019-12-23
SELECT CURRENT_TIMESTAMP;
Результат: 2019-12-23 14:39:53.662522-05
```

```
SELECT CURRENT_TIMESTAMP(2);
Результат: 2019-12-23 14:39:53.66-05
SELECT LOCALTIMESTAMP;
Результат: 2019-12-23 14:39:53.662522
```

Так как эти функции возвращают время начала текущей транзакции, во время транзакции эти значения не меняются. Это считается не ошибкой, а особенностью реализации: цель такого поведения в том, чтобы в одной транзакции «текущее» время было одинаковым и для разных изменений в одной транзакций записывалась одна отметка времени.

> ### Примечание
>
> В других СУБД эти значения могут изменяться чаще.

В Postgres Pro есть также функции, возвращающие время начала текущего оператора, а также текущее время в момент вызова функции. Таким образом, в Postgres Pro есть следующие функции, не описанные в стандарте SQL:

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

Функция `transaction_timestamp()` равнозначна конструкции `CURRENT_TIMESTAMP`, но в её названии явно отражено, что она возвращает. Функция `statement_timestamp()` возвращает время начала текущего оператора (более точно, время получения последнего командного сообщения от клиента). Функции `statement_timestamp()` и `transaction_timestamp()` возвращают одно и то же значение в первом операторе транзакции, но в последующих операторах их показания будут расходиться. Функция `clock_timestamp()` возвращает фактическое текущее время, так что её значение меняется в рамках одного оператора SQL. Функция `timeofday()` существует в Postgres Pro по историческим причинам и, подобно `clock_timestamp()`, она возвращает фактическое текущее время, но представленное в виде форматированной строки типа `text`, а не значения `timestamp with time zone`. Функция `now()` — традиционный для Postgres Pro эквивалент функции `transaction_timestamp()`.

Все типы даты/времени также принимают специальное буквальное значение `now`, подразумевающее текущую дату и время (тоже на момент начала транзакции). Таким образом, результат следующих трёх операторов будет одинаковым:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- см. замечание ниже
```

> ### Подсказка
>
> Не используйте третью форму для указания значения, которое будет вычисляться позднее, например, в предложении `DEFAULT` для столбца таблицы. Система преобразует `now` в значение `timestamp` в момент разбора константы, поэтому когда будет вставляться такое значение по умолчанию, в соответствующем столбце окажется время создания таблицы! Первые две формы будут вычисляться, только когда значение по умолчанию потребуется, так как это вызовы функции. Поэтому они дадут желаемый результат при добавлении строки в таблицу. (См. также Подраздел 8.5.1.4.)

9.9.6. Задержка выполнения

В случае необходимости выполнение серверного процесса можно приостановить, используя следующие функции:

```
pg_sleep( double precision )
pg_sleep_for( interval )
pg_sleep_until( timestamp with time zone )
```

Функция `pg_sleep` переводит процесс текущего сеанса в спящее состояние на указанное число секунд (это число может быть дробным). В дополнение к ней для удобства добавлены две функции: `pg_sleep_for`, принимающая время задержки в типе `interval`, и `pg_sleep_until`, позволяющая задать определённое время выхода из спящего состояния. Например:

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

Примечание

Действительное разрешение интервала задержки зависит от платформы; обычно это 0.01. Фактическая длительность задержки не будет меньше указанного времени, но может быть больше, в зависимости, например от нагрузки на сервер. В частности, не гарантируется, что `pg_sleep_until` проснётся именно в указанное время, но она точно не проснётся раньше.

Предупреждение

Прежде чем вызывать `pg_sleep` или её вариации, убедитесь в том, что в текущем сеансе нет ненужных блокировок. В противном случае в состояние ожидания могут перейти и другие сеансы, так что это отразится на системе в целом.

9.10. Функции для перечислений

Для типов перечислений (описанных в Разделе 8.7) предусмотрено несколько функций, которые позволяют сделать код чище, не «зашивая» в нём конкретные значения перечисления. Эти функции перечислены в Таблице 9.35. В этих примерах подразумевается, что перечисление создано так:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

Таблица 9.35. Функции для перечислений

Функция Описание Пример(ы)

`enum_first` (`anyenum`) → `anyenum`
Возвращает первое значение заданного перечисления.
`enum_first(null::rainbow)` → `red`

`enum_last` (`anyenum`) → `anyenum`
Возвращает последнее значение заданного перечисления.
`enum_last(null::rainbow)` → `purple`

`enum_range` (`anyenum`) → `anyarray`
Возвращает все значения заданного перечисления в упорядоченном массиве.
`enum_range(null::rainbow)` → `{red,orange,yellow,green,blue,purple}`

`enum_range` (`anyenum`, `anyenum`) → `anyarray`
Возвращает набор значений перечисления, лежащих между двумя заданными, в виде упорядоченного массива. Значения в параметрах должны принадлежать одному перечислению. Если в первом параметре передаётся NULL, результат функции начинается с первого значения перечисления, а если во втором — заканчивается последним.
`enum_range('orange'::rainbow, 'green'::rainbow)` → `{orange,yellow,green}`

Функция **Описание** **Пример(ы)**
`enum_range(NULL, 'green'::rainbow)` → `{red,orange,yellow,green}` `enum_range('orange'::rainbow, NULL)` → `{orange,yellow,green,blue,purple}`

Заметьте, что за исключением варианта `enum_range` с двумя аргументами, эти функции не обращают внимание на конкретное переданное им значение; их интересует только объявленный тип. Они возвращают один и тот же результат, когда им передаётся NULL или любое другое значение типа. Обычно эти функции применяются к столбцам таблицы или аргументам внешних функций, а не к предопределённым типам, как в этих примерах.

9.11. Геометрические функции и операторы

Для геометрических типов `point`, `box`, `lseg`, `line`, `path`, `polygon` и `circle` разработан большой набор встроенных функций и операторов, представленный в Таблице 9.36, Таблице 9.37 и Таблице 9.38.

Таблица 9.36. Геометрические операторы

Оператор **Описание** **Пример(ы)**
геометрический_тип + `point` → *геометрический_тип* Добавляет координаты второго аргумента типа `point` к каждой точке первого аргумента, осуществляя таким образом перенос объекта. Имеется для типов `point`, `box`, `path` и `circle`. `box '(1,1),(0,0)' + point '(2,0)'` → `(3,1),(2,0)`

`path` + `path` → `path`
Соединяет два открытых пути (если один из путей замкнутый, возвращает NULL).
`path '[(0,0),(1,1)]' + path '[(2,2),(3,3),(4,4)]'` → `[(0,0),(1,1),(2,2),(3,3),(4,4)]`

геометрический_тип - `point` → *геометрический_тип* Вычитает координаты второго аргумента типа `point` из каждой точки первого аргумента, осуществляя таким образом перенос объекта. Имеется для типов `point`, `box`, `path` и `circle`. `box '(1,1),(0,0)' - point '(2,0)'` → `(-1,1),(-2,0)`

геометрический_тип * `point` → *геометрический_тип*
Умножает координаты каждой точки первого аргумента на координаты второго аргумента типа `point` (координаты точек воспринимаются как комплексные числа с вещественной и мнимой частью, результатом становится их обычное комплексное произведение). Если же рассматривать вторую точку как вектор, данная операция равнозначна умножению размера объекта и расстояния от начала координат на длину вектора с поворотом против часовой стрелки относительно начала координат на угол, равный углу между вектором и осью *x*. Имеется для типов `point`, `box`,[a] `path`, `circle`.
`path '((0,0),(1,0),(1,1))' * point '(3.0,0)'` → `((0,0),(3,0),(3,3))`
`path '((0,0),(1,0),(1,1))' * point(cosd(45), sind(45))` → `((0,0),(0.7071067811865475,0.7071067811865475),(0,1.414213562373095))`

геометрический_тип / `point` → *геометрический_тип* Делит координаты каждой точки первого аргумента на координаты второго аргумента типа `point` (координаты точек воспринимаются как комплексные числа с вещественной и мнимой частью, результатом становится их обычное комплексное частное). Если же рассматривать вторую точку как вектор, данная операция равнозначна делению разме-

Оператор **Описание** **Пример(ы)**

ра объекта и расстояния от начала координат на длину вектора с поворотом по часовой стрелке относительно начала координат на угол, равный углу между вектором и осью *x*. Имеется для типов `point`, `box`,[a] `path`, `circle`.
`path '((0,0),(1,0),(1,1))' / point '(2.0,0)'` → `((0,0),(0.5,0),(0.5,0.5))`
`path '((0,0),(1,0),(1,1))' / point(cosd(45), sind(45))` → `((0,0),(0.7071067811865476,-0.7071067811865476),(1.4142135623730951,0))`

`@-@` *геометрический_тип* → `double precision`
Вычисляет общую длину. Имеется для типов `lseg`, `path`.
`@-@ path '[(0,0),(1,0),(1,1)]'` → `2`

`@@` *геометрический_тип* → `point`
Вычисляет центральную точку. Имеется для типов `box`, `lseg`, `polygon`, `circle`.
`@@ box '(2,2),(0,0)'` → `(1,1)`

`#` *геометрический_тип* → `integer`
Возвращает количество точек. Имеется для типов `path`, `polygon`.
`# path '((1,0),(0,1),(-1,0))'` → `3`

геометрический_тип `#` *геометрический_тип* → `point` Вычисляет точку пересечения, а если пересечения нет, возвращает NULL. Имеется для типов `lseg`, `line`. `lseg '[(0,0),(1,1)]' # lseg '[(1,0),(0,1)]'` → `(0.5,0.5)`

`box # box → box`
Вычисляет пересечение двух прямоугольников, а если пересечения нет, возвращает NULL.
`box '(2,2),(-1,-1)' # box '(1,1),(-2,-2)'` → `(1,1),(-1,-1)`

геометрический_тип `##` *геометрический_тип* → `point` Вычисляет ближайшую к первому объекту точку, принадлежащую второму объекту. Имеется для следующих пар типов: (`point`, `box`), (`point`, `lseg`), (`point`, `line`), (`lseg`, `box`), (`lseg`, `lseg`), (`line`, `lseg`). `point '(0,0)' ## lseg '[(2,0),(0,2)]'` → `(1,1)`
геометрический_тип `<->` *геометрический_тип* → `double precision` Вычисляет расстояние между объектами. Имеется для всех семи геометрических типов, для всех сочетаний типа `point` с другим геометрическим типом, а также для следующих пар типов: (`box`, `lseg`), (`lseg`, `line`), (`polygon`, `circle`) (и пар с обратным порядком). `circle '<(0,0),1>' <-> circle '<(5,0),1>'` → `3`
геометрический_тип `@>` *геометрический_тип* → `boolean` Первый объект содержит второй? Имеется для следующих пар типов: (`box`, `point`), (`box`, `box`), (`path`, `point`), (`polygon`, `point`), (`polygon`, `polygon`), (`circle`, `point`), (`circle`, `circle`). `circle '<(0,0),2>' @> point '(1,1)'` → `t`
геометрический_тип `<@` *геометрический_тип* → `boolean` Первый объект содержится во втором? Имеется для следующих пар типов: (`point`, `box`), (`point`, `lseg`), (`point`, `line`), (`point`, `path`), (`point`, `polygon`), (`point`, `circle`), (`box`, `box`), (`lseg`, `box`), (`lseg`, `line`), (`polygon`, `polygon`), (`circle`, `circle`). `point '(1,1)' <@ circle '<(0,0),2>'` → `t`
геометрический_тип `&&` *геометрический_тип* → `boolean`

Оператор **Описание** **Пример(ы)**
Объекты пересекаются? (Для выполнения этого условия достаточно одной общей точки.) Имеется для типов `box`, `polygon`, `circle`. `box '(1,1),(0,0)' && box '(2,2),(0,0)'` → `t`
геометрический_тип `<<` *геометрический_тип* → `boolean` Первый объект строго слева от второго? Имеется для типов `point`, `box`, `polygon`, `circle`. `circle '<(0,0),1>' << circle '<(5,0),1>'` → `t`
геометрический_тип `>>` *геометрический_тип* → `boolean` Первый объект строго справа от второго? Имеется для типов `point`, `box`, `polygon`, `circle`. `circle '<(5,0),1>' >> circle '<(0,0),1>'` → `t`
геометрический_тип `&<` *геометрический_тип* → `boolean` Первый объект не простирается правее второго? Имеется для типов `box`, `polygon`, `circle`. `box '(1,1),(0,0)' &< box '(2,2),(0,0)'` → `t`
геометрический_тип `&>` *геометрический_тип* → `boolean` Первый объект не простирается левее второго? Имеется для типов `box`, `polygon`, `circle`. `box '(3,3),(0,0)' &> box '(2,2),(0,0)'` → `t`
геометрический_тип `<<|` *геометрический_тип* → `boolean` Первый объект строго ниже второго? Имеется для типов `point`, `box`, `polygon`, `circle`. `box '(3,3),(0,0)' <<| box '(5,5),(3,4)'` → `t`
геометрический_тип `|>>` *геометрический_тип* → `boolean` Первый объект строго выше второго? Имеется для типов `point`, `box`, `polygon`, `circle`. `box '(5,5),(3,4)' |>> box '(3,3),(0,0)'` → `t`
геометрический_тип `&<|` *геометрический_тип* → `boolean` Первый объект не простирается выше второго? Имеется для типов `box`, `polygon`, `circle`. `box '(1,1),(0,0)' &<| box '(2,2),(0,0)'` → `t`
геометрический_тип `|&>` *геометрический_тип* → `boolean` Первый объект не простирается ниже второго? Имеется для типов `box`, `polygon`, `circle`. `box '(3,3),(0,0)' |&> box '(2,2),(0,0)'` → `t`

`box <^ box` → `boolean`
Первый объект ниже (или касается снизу) второго?
`box '((1,1),(0,0))' <^ box '((2,2),(1,1))'` → `t`

`box >^ box` → `boolean`
Первый объект выше (или касается сверху) второго?
`box '((2,2),(1,1))' >^ box '((1,1),(0,0))'` → `t`

геометрический_тип `?#` *геометрический_тип* → `boolean` Объекты пересекаются? Имеется для следующих пар типов: (`box`, `box`), (`lseg`, `box`), (`lseg`, `lseg`), (`lseg`, `line`), (`line`, `box`), (`line`, `line`), (`path`, `path`). `lseg '[(-1,0),(1,0)]' ?# box '(2,2),(-2,-2)'` → `t`

`?- line` → `boolean`
`?- lseg` → `boolean`
Линия является горизонтальной?
`?- lseg '[(-1,0),(1,0)]'` → `t`

Оператор	**Описание**	**Пример(ы)**
`point ?- point` → `boolean`	Точки выровнены по горизонтали (имеют одинаковую координату y)?	`point '(1,0)' ?- point '(0,0)'` → `t`
`?| line` → `boolean` `?| lseg` → `boolean`	Линия является вертикальной?	`?| lseg '[(-1,0),(1,0)]'` → `f`
`point ?| point` → `boolean`	Точки выровнены по вертикали (имеют одинаковую координату x)?	`point '(0,1)' ?| point '(0,0)'` → `t`
`line ?-| line` → `boolean` `lseg ?-| lseg` → `boolean`	Линии перпендикулярны?	`lseg '[(0,0),(0,1)]' ?-| lseg '[(0,0),(1,0)]'` → `t`
`line ?|| line` → `boolean` `lseg ?|| lseg` → `boolean`	Линии параллельны?	`lseg '[(-1,0),(1,0)]' ?|| lseg '[(-1,2),(1,2)]'` → `t`
геометрический_тип `~=` *геометрический_тип* → `boolean`	Объекты совпадают? Имеется для типов `point`, `box`, `polygon`, `circle`.	`polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'` → `t`

[a] При «повороте» прямоугольника эти операторы только перемещают его угловые точки: стороны прямоугольника считаются всегда параллельными осям. Таким образом, эта операция, в отличие от настоящего поворота, изменяет размер прямоугольника.

Внимание

Заметьте, что оператор «идентичности», `~=`, представляет собой обычную проверку равенства значений `point`, `box`, `polygon` и `circle`. Для некоторых геометрических типов определён также оператор `=`, но `=` проверяет только равенство *площадей*. Другие скалярные операторы сравнения (`<=` и т. д.) для тех типов, для которых они реализованы, тоже сравнивают площади.

Примечание

До Postgres Pro 14 применявшиеся к точкам операторы `point <<| point` (строго ниже) и `point |>> point` (строго выше) назывались `<^` и `>^` соответственно. Эти имена по-прежнему доступны, но считаются устаревшими и в конце концов будут удалены.

Таблица 9.37. Геометрические функции

Функция	**Описание**	**Пример(ы)**
`area` (*геометрический_тип*) → `double precision`		

Функция	**Описание**	**Пример(ы)**
	Вычисляет площадь. Имеется для типов `box`, `path`, `circle`. Путь, переданный в аргументе типа `path`, должен быть замкнутым; в противном случае возвращается NULL. Если же путь является самопересекающимся, результат может не иметь смысла.	`area(box '(2,2),(0,0)')` → `4`
`center` (*`geometric_type`*) → `point`	Вычисляет центральную точку. Имеется для типов `box`, `circle`.	`center(box '(1,2),(0,0)')` → `(0.5,1)`
`diagonal` (`box`) → `lseg`	Вычисляет диагональ прямоугольника в виде отрезка (аналогично `lseg(box)`).	`diagonal(box '(1,2),(0,0)')` → `[(1,2),(0,0)]`
`diameter` (`circle`) → `double precision`	Вычисляет диаметр круга.	`diameter(circle '<(0,0),2>')` → `4`
`height` (`box`) → `double precision`	Вычисляет вертикальный размер прямоугольника.	`height(box '(1,2),(0,0)')` → `2`
`isclosed` (`path`) → `boolean`	Путь является замкнутым?	`isclosed(path '((0,0),(1,1),(2,0))')` → `t`
`isopen` (`path`) → `boolean`	Путь является открытым?	`isopen(path '[(0,0),(1,1),(2,0)]')` → `t`
`length` (*`геометрический_тип`*) → `double precision`	Вычисляет общую длину. Имеется для типов `lseg`, `path`.	`length(path '((-1,0),(1,0))')` → `4`
`npoints` (*`геометрический_тип`*) → `integer`	Возвращает количество точек. Имеется для типов `path`, `polygon`.	`npoints(path '[(0,0),(1,1),(2,0)]')` → `3`
`pclose` (`path`) → `path`	Преобразует путь в замкнутый.	`pclose(path '[(0,0),(1,1),(2,0)]')` → `((0,0),(1,1),(2,0))`
`popen` (`path`) → `path`	Преобразует путь в открытый.	`popen(path '((0,0),(1,1),(2,0))')` → `[(0,0),(1,1),(2,0)]`
`radius` (`circle`) → `double precision`	Вычисляет радиус окружности.	`radius(circle '<(0,0),2>')` → `2`
`slope` (`point`, `point`) → `double precision`	Вычисляет наклон линии, проведённой через две точки.	`slope(point '(0,0)', point '(2,1)')` → `0.5`
`width` (`box`) → `double precision`	Вычисляет горизонтальный размер прямоугольника.	

Функция **Описание** **Пример(ы)**
`width(box '(1,2),(0,0)')` → `1`

Таблица 9.38. Функции преобразования геометрических типов

Функция **Описание** **Пример(ы)**

`box ( circle ) → box`
Вычисляет прямоугольник, вписанный в окружность.
`box(circle '<(0,0),2>')` → `(1.414213562373095,1.414213562373095),(-1.414213562373095,-1.414213562373095)`

`box ( point ) → box`
Преобразует точку в пустой прямоугольник.
`box(point '(1,0)')` → `(1,0),(1,0)`

`box ( point, point ) → box`
Преобразует две угловые точки в прямоугольник.
`box(point '(0,1)', point '(1,0)')` → `(1,1),(0,0)`

`box ( polygon ) → box`
Вычисляет окружающий прямоугольник для многоугольника.
`box(polygon '((0,0),(1,1),(2,0))')` → `(2,1),(0,0)`

`bound_box ( box, box ) → box`
Вычисляет окружающий прямоугольник для двух прямоугольников.
`bound_box(box '(1,1),(0,0)', box '(4,4),(3,3)')` → `(4,4),(0,0)`

`circle ( box ) → circle`
Вычисляет минимальную окружность, описанную около прямоугольника.
`circle(box '(1,1),(0,0)')` → `<(0.5,0.5),0.7071067811865476>`

`circle ( point, double precision ) → circle`
Формирует окружность по точке, задающей центр, и радиусу.
`circle(point '(0,0)', 2.0)` → `<(0,0),2>`

`circle ( polygon ) → circle`
Преобразует многоугольник в окружность. Центром окружности будет средняя точка для всех точек многоугольника, а радиусом среднее расстояние от этого центра до точек многоугольника.
`circle(polygon '((0,0),(1,3),(2,0))')` → `<(1,1),1.6094757082487299>`

`line ( point, point ) → line`
Преобразует две точки в проходящую через них линию.
`line(point '(-1,0)', point '(1,0)')` → `{0,-1,0}`

`lseg ( box ) → lseg`
Вычисляет диагональ прямоугольника в виде отрезка.
`lseg(box '(1,0),(-1,0)')` → `[(1,0),(-1,0)]`

`lseg ( point, point ) → lseg`
Формирует отрезок по двум точкам.
`lseg(point '(-1,0)', point '(1,0)')` → `[(-1,0),(1,0)]`

`path ( polygon ) → path`

Функция **Описание** **Пример(ы)**
Преобразует многоугольник в замкнутый путь с тем же набором точек. `path(polygon '((0,0),(1,1),(2,0))')` → `((0,0),(1,1),(2,0))`

`point` (`double precision`, `double precision`) → `point`
Формирует точку по заданным координатам.
`point(23.4, -44.5)` → `(23.4,-44.5)`

`point` (`box`) → `point`
Вычисляет центр прямоугольника.
`point(box '(1,0),(-1,0)')` → `(0,0)`

`point` (`circle`) → `point`
Вычисляет центр окружности.
`point(circle '<(0,0),2>')` → `(0,0)`

`point` (`lseg`) → `point`
Вычисляет середину отрезка.
`point(lseg '[(-1,0),(1,0)]')` → `(0,0)`

`point` (`polygon`) → `point`
Вычисляет центр многоугольника (среднее арифметическое по координатам его точек).
`point(polygon '((0,0),(1,1),(2,0))')` → `(1,0.3333333333333333)`

`polygon` (`box`) → `polygon`
Преобразует прямоугольник в многоугольник с 4 вершинами.
`polygon(box '(1,1),(0,0)')` → `((0,0),(0,1),(1,1),(1,0))`

`polygon` (`circle`) → `polygon`
Преобразует круг в многоугольник с 12 вершинами.
`polygon(circle '<(0,0),2>')` → `((-2,0),(-1.7320508075688774,0.9999999999999999),(-1.0000000000000002,1.7320508075688772),(-1.2246063538223773e-16,2),(0.9999999999999996,1.7320508075688774),(1.732050807568877,1.0000000000000007),(2,2.4492127076447545e-16),(1.7320508075688776,-0.9999999999999994),(1.0000000000000009,-1.7320508075688767),(3.673819061467132e-16,-2),(-0.9999999999999987,-1.732050807568878),(-1.7320508075688767,-1.0000000000000009))`

`polygon` (`integer`, `circle`) → `polygon`
Преобразует окружность в многоугольник с *n* вершинами.
`polygon(4, circle '<(3,0),1>')` → `((2,0),(3,1),(4,1.2246063538223773e-16),(3,-1))`

`polygon` (`path`) → `polygon`
Преобразует замкнутый путь в многоугольник с тем же набором точек.
`polygon(path '((0,0),(1,1),(2,0))')` → `((0,0),(1,1),(2,0))`

К двум компонентам типа `point` (точка) можно обратиться, как к элементам массива с индексами 0 и 1. Например, если `t.p` — столбец типа `point`, `SELECT p[0] FROM t` вернёт координату X, а `UPDATE t SET p[1] = ...` изменит координату Y. Таким же образом, значение типа `box` или `lseg` можно воспринимать как массив двух значений типа `point`.

9.12. Функции и операторы для работы с сетевыми адресами

Типы `cidr` и `inet`, предназначенные для сетевых IP-адресов, поддерживают обычные операторы сравнения, показанные в Таблице 9.1, а также специализированные операторы и функции, показанные в Таблице 9.39 и Таблице 9.40.

Любое значение `cidr` можно привести к типу `inet` неявно, поэтому все функции, показанные выше с типом `inet`, также будут работать со значениями `cidr`. (То, что некоторые из функций описаны для типов `inet` и `cidr` в отдельности, объясняется тем, что их поведение с разными типами различается.) Кроме того, значение `inet` тоже можно привести к типу `cidr`. При этом все биты справа от сетевой маски просто обнуляются, чтобы значение стало допустимым для типа `cidr`.

Таблица 9.39. Операторы для работы с IP-адресами

Оператор **Описание** **Пример(ы)**

`inet << inet → boolean`
Первая подсеть содержится во второй и не равна ей? Этот оператор и следующие четыре проверяют вхождение одной сети в другую или их равенство, при этом рассматривая в адресах только компонент сети (биты справа от сетевой маски игнорируются).
`inet '192.168.1.5' << inet '192.168.1/24'` → `t`
`inet '192.168.0.5' << inet '192.168.1/24'` → `f`
`inet '192.168.1/24' << inet '192.168.1/24'` → `f`

`inet <<= inet → boolean`
Первая подсеть содержится во второй или равна ей?
`inet '192.168.1/24' <<= inet '192.168.1/24'` → `t`

`inet >> inet → boolean`
Первая подсеть содержит вторую подсеть и не равна ей?
`inet '192.168.1/24' >> inet '192.168.1.5'` → `t`

`inet >>= inet → boolean`
Первая подсеть содержит вторую подсеть или равна ей?
`inet '192.168.1/24' >>= inet '192.168.1/24'` → `t`

`inet && inet → boolean`
Одна из двух подсетей содержит другую или равна ей?
`inet '192.168.1/24' && inet '192.168.1.80/28'` → `t`
`inet '192.168.1/24' && inet '192.168.2.0/28'` → `f`

`~ inet → inet`
Вычисляет результат побитового НЕ.
`~ inet '192.168.1.6'` → `63.87.254.249`

`inet & inet → inet`
Вычисляет результат побитового И.
`inet '192.168.1.6' & inet '0.0.0.255'` → `0.0.0.6`

`inet | inet → inet`
Вычисляет результат побитового ИЛИ.
`inet '192.168.1.6' | inet '0.0.0.255'` → `192.168.1.255`

`inet + bigint → inet` Добавляет смещение к адресу.

Оператор **Описание** **Пример(ы)**
`inet '192.168.1.6' + 25` → `192.168.1.31`

`bigint` + `inet` → `inet`
Добавляет смещение к адресу.
`200 + inet '::ffff:fff0:1'` → `::ffff:255.240.0.201`

`inet` - `bigint` → `inet`
Вычитает смещение из адреса.
`inet '192.168.1.43' - 36` → `192.168.1.7`

`inet` - `inet` → `bigint`
Вычисляет разность двух адресов.
`inet '192.168.1.43' - inet '192.168.1.19'` → `24`
`inet '::1' - inet '::ffff:1'` → `-4294901760`

Таблица 9.40. Функции для работы с IP-адресами

Функция **Описание** **Пример(ы)**

`abbrev` (`inet`) → `text`
Выводит адрес в сокращённом текстовом виде. (Результат не отличается от того, что даёт функция вывода `inet`; «сокращённым» он является только в сравнении с явным приведением к типу `text`, которое по историческим причинам не убирает компонент маски сети).
`abbrev(inet '10.1.0.0/32')` → `10.1.0.0`

`abbrev` (`cidr`) → `text`
Выводит адрес в сокращённом текстовом виде. (Сокращение заключается в отбрасывании полностью нулевых октетов в конце сетевой маски; другие примеры приведены в Таблице 8.22.)
`abbrev(cidr '10.1.0.0/16')` → `10.1/16`

`broadcast` (`inet`) → `inet`
Вычисляет широковещательный адрес для сети.
`broadcast(inet '192.168.1.5/24')` → `192.168.1.255/24`

`family` (`inet`) → `integer`
Выдаёт семейство адреса: `4` для адресов IPv4, `6` для адресов IPv6.
`family(inet '::1')` → `6`

`host` (`inet`) → `text`
Выдаёт IP-адрес в текстовом виде, опуская маску сети.
`host(inet '192.168.1.0/24')` → `192.168.1.0`

`hostmask` (`inet`) → `inet`
Вычисляет маску узла для сети в заданном адресе.
`hostmask(inet '192.168.23.20/30')` → `0.0.0.3`

`inet_merge` (`inet`, `inet`) → `cidr`
Вычисляет наименьшую сеть, содержащую обе заданные сети.
`inet_merge(inet '192.168.1.5/24', inet '192.168.2.5/24')` → `192.168.0.0/22`

`inet_same_family` (`inet`, `inet`) → `boolean` Проверяет, относятся ли адреса к одному семейству IP?

Функция Описание Пример(ы)
`inet_same_family(inet '192.168.1.5/24', inet '::1')` → `f`

`masklen` (`inet`) → `integer`
Выдаёт длину маски сети в битах.
`masklen(inet '192.168.1.5/24')` → `24`

`netmask` (`inet`) → `inet`
Вычисляет маску сети для заданного адреса.
`netmask(inet '192.168.1.5/24')` → `255.255.255.0`

`network` (`inet`) → `cidr`
Выдаёт компонент сети для заданного адреса, обнуляя все биты справа от маски сети. (Это равнозначно приведению значения к типу `cidr`.)
`network(inet '192.168.1.5/24')` → `192.168.1.0/24`

`set_masklen` (`inet`, `integer`) → `inet`
Задаёт размер маски сети для значения `inet`. Компонент адреса при этом не меняется.
`set_masklen(inet '192.168.1.5/24', 16)` → `192.168.1.5/16`

`set_masklen` (`cidr`, `integer`) → `cidr`
Задаёт размер маски сети для значения `cidr`. Биты адреса справа от новой маски при этом обнуляются.
`set_masklen(cidr '192.168.1.0/24', 16)` → `192.168.0.0/16`

`text` (`inet`) → `text`
Выдаёт несокращённый IP-адрес и размер маски в виде текста. (Такой же результат получается при явном приведении к типу `text`.)
`text(inet '192.168.1.5')` → `192.168.1.5/32`

Подсказка

Функции `abbrev`, `host` и `text` предназначены в основном для вывода IP-адресов в альтернативных текстовых форматах.

Типы MAC-адресов `macaddr` и `macaddr8` поддерживают обычные операторы сравнения, показанные в Таблице 9.1, а также специализированные функции, показанные в Таблице 9.41. Кроме того, они поддерживают битовые логические операторы: `~`, `&` и `|` (НЕ, И, ИЛИ), показанные выше для IP-адресов.

Таблица 9.41. Функции для работы с MAC-адресами

Функция Описание Пример(ы)

`trunc` (`macaddr`) → `macaddr`
Обнуляет три последних байта адреса. Оставшийся префикс можно сопоставить с конкретным производителем сетевой карты (в Postgres Pro необходимой для этого информации нет).
`trunc(macaddr '12:34:56:78:90:ab')` → `12:34:56:00:00:00`

`trunc` (`macaddr8`) → `macaddr8` Обнуляет 5 последних байт адреса. Оставшийся префикс можно сопоставить с конкретным производителем сетевой карты (в Postgres Pro необходимой для этого информации нет).

Функция **Описание** **Пример(ы)**
`trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')` → `12:34:56:00:00:00:00:00`

`macaddr8_set7bit ( macaddr8 )` → `macaddr8`
Устанавливает в 7 бите адреса единицу, получая тем самым так называемый модифицированный адрес EUI-64 для включения в адрес IPv6.
`macaddr8_set7bit(macaddr8 '00:34:56:ab:cd:ef')` → `02:34:56:ff:fe:ab:cd:ef`

9.13. Функции и операторы текстового поиска

В Таблице 9.42, Таблице 9.43 и Таблице 9.44 собраны все существующие функции и операторы, предназначенные для полнотекстового поиска. Во всех деталях возможности полнотекстового поиска в Postgres Pro описаны в Главе 12.

Таблица 9.42. Операторы текстового поиска

Оператор **Описание** **Пример(ы)**

`tsvector @@ tsquery` → `boolean`
`tsquery @@ tsvector` → `boolean`
Аргумент `tsvector` соответствует аргументу `tsquery`? (Аргументы могут передаваться в любом порядке.)
`to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')` → `t`

`text @@ tsquery` → `boolean`
Переданная текстовая строка, неявно преобразованная функцией `to_tsvector()`, соответствует `tsquery`?
`'fat cats ate rats' @@ to_tsquery('cat & rat')` → `t`

`tsvector || tsvector` → `tsvector`
Соединяет два значения `tsvector`. Если в обоих значениях содержатся позиции лексем, позиции во втором при совмещении корректируются.
`'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector` → `'a':1 'b':2,5 'c':3 'd':4`

`tsquery && tsquery` → `tsquery`
Вычисляет конъюнкцию двух значений `tsquery`, получая тем самым запрос, которому будут соответствовать документы, соответствующие обоим входным запросам.
`'fat | rat'::tsquery && 'cat'::tsquery` → `( 'fat' | 'rat' ) & 'cat'`

`tsquery || tsquery` → `tsquery`
Вычисляет дизъюнкцию двух значений `tsquery`, получая тем самым запрос, которому будут соответствовать документы, соответствующие любому из входных запросов.
`'fat | rat'::tsquery || 'cat'::tsquery` → `'fat' | 'rat' | 'cat'`

`!! tsquery` → `tsquery`
Вычисляет отрицание `tsquery`, получая тем самым запрос, которому будут соответствовать документы, не соответствующие входному запросу.
`!! 'cat'::tsquery` → `!'cat'`

`tsquery <-> tsquery` → `tsquery`
Конструирует фразовый запрос, которому будут соответствовать документы, содержащие подряд идущие лексемы, удовлетворяющие двум входным запросам.
`to_tsquery('fat') <-> to_tsquery('rat')` → `'fat' <-> 'rat'`

`tsquery @> tsquery` → `boolean`

Оператор **Описание** **Пример(ы)**
Первый запрос типа `tsquery` содержит второй? (При этом учитывается только факт нахождения всех лексем из одного запроса в другом; операторы их сочетания игнорируются.) `'cat'::tsquery @> 'cat & rat'::tsquery` → `f`

`tsquery <@ tsquery` → `boolean`
Первый запрос типа `tsquery` содержится во втором? (При этом учитывается только факт нахождения всех лексем из одного запроса в другом; операторы их сочетания игнорируются.)
`'cat'::tsquery <@ 'cat & rat'::tsquery` → `t`
`'cat'::tsquery <@ '!cat & rat'::tsquery` → `t`

Помимо этих специализированных операторов, для типов `tsvector` и `tsquery` имеются обычные операторы сравнения, показанные в Таблице 9.1. Они не очень полезны для поиска, но позволяют, в частности, создавать индексы по столбцам этих типов.

Таблица 9.43. Функции текстового поиска

Функция **Описание** **Пример(ы)**

`array_to_tsvector` (`text[]`) → `tsvector`
Преобразует массив текстовых строк в `tsvector`. Переданные строки используются в виде лексем как есть, без дополнительной обработки. Элементы массива не должны быть пустыми строками или `NULL`.
`array_to_tsvector('{fat,cat,rat}'::text[])` → `'cat' 'fat' 'rat'`

`get_current_ts_config` ( ) → `regconfig`
Возвращает OID текущей конфигурации текстового поиска по умолчанию (задаваемой параметром default_text_search_config).
`get_current_ts_config()` → `english`

`length` (`tsvector`) → `integer`
Возвращает число лексем в значении `tsvector`.
`length('fat:2,4 cat:3 rat:5A'::tsvector)` → `3`

`numnode` (`tsquery`) → `integer`
Возвращает число лексем и операторов в запросе `tsquery`.
`numnode('(fat & rat) | cat'::tsquery)` → `5`

`plainto_tsquery` ([*config* `regconfig`,] *query* `text`) → `tsquery`
Преобразует текст в `tsquery`, выполняя нормализацию слов согласно конфигурации по умолчанию или указанной явно. Знаки пунктуации во входной строке при этом игнорируются (в данном случае она не определяет операторы запроса). Результирующему запросу будут соответствовать документы, содержащие все слова этого текста, кроме стоп-слов.
`plainto_tsquery('english', 'The Fat Rats')` → `'fat' & 'rat'`

`phraseto_tsquery` ([*config* `regconfig`,] *query* `text`) → `tsquery`
Преобразует текст в `tsquery`, выполняя нормализацию слов согласно конфигурации по умолчанию или указанной явно. Знаки пунктуации во входной строке при этом игнорируются (в данном случае она не определяет операторы запроса). Результирующему запросу будут соответствовать фразы, содержащие последовательность слов этого текста, кроме стоп-слов.
`phraseto_tsquery('english', 'The Fat Rats')` → `'fat' <-> 'rat'`

Функция **Описание** **Пример(ы)**
`phraseto_tsquery('english', 'The Cat and Rats')` → `'cat' <2> 'rat'`

`websearch_to_tsquery` ([*config* `regconfig`,] *query* `text`) → `tsquery`
Преобразует текст в `tsquery`, выполняя нормализацию слов согласно конфигурации по умолчанию или указанной явно. Последовательности слов в кавычках преобразуются в проверки фраз. Слово «or» воспринимается как оператор OR (ИЛИ), а символ минуса преобразуется в оператор NOT (НЕ); другие знаки пунктуации игнорируются. Это примерно соответствует поведению ряда распространённых средств поиска в вебе.
`websearch_to_tsquery('english', '"fat rat" or cat dog')` → `'fat' <-> 'rat' | 'cat' & 'dog'`

`querytree` (`tsquery`) → `text`
Формирует представление индексируемой части `tsquery`. Если возвращается пустой результат или просто `T`, это означает, что запрос не индексируемый.
`querytree('foo & ! bar'::tsquery)` → `'foo'`

`setweight` (*vector* `tsvector`, *weight* `"char"`) → `tsvector`
Назначает вес, указанный в аргументе *`weight`*, каждому элементу аргумента *`vector`*.
`setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')` → `'cat':3A 'fat':2A,4A 'rat':5A`

`setweight` (*vector* `tsvector`, *weight* `"char"`, *lexemes* `text[]`) → `tsvector`
Назначает вес, указанный в аргументе *`weight`*, элементам аргумента *`vector`*, перечисленным в аргументе *`lexemes`*. Строки в *`lexemes`* принимаются в виде лексем как есть, без дальнейшей обработки. Строки, не соответствующие ни одной лексеме в *`vector`*, игнорируются.
`setweight('fat:2,4 cat:3 rat:5,6B'::tsvector, 'A', '{cat,rat}')` → `'cat':3A 'fat':2,4 'rat':5A,6A`

`strip` (`tsvector`) → `tsvector`
Убирает позиции и веса из значения `tsvector`.
`strip('fat:2,4 cat:3 rat:5A'::tsvector)` → `'cat' 'fat' 'rat'`

`to_tsquery` ([*config* `regconfig`,] *query* `text`) → `tsquery`
Преобразует текст в `tsquery`, выполняя нормализацию слов согласно конфигурации по умолчанию или указанной явно. Слова должны разделяться операторами `tsquery`.
`to_tsquery('english', 'The & Fat & Rats')` → `'fat' & 'rat'`

`to_tsvector` ([*config* `regconfig`,] *document* `text`) → `tsvector`
Преобразует текст в `tsvector`, выполняя нормализацию слов согласно конфигурации по умолчанию или указанной явно. Результат будет включать информацию о позициях слов.
`to_tsvector('english', 'The Fat Rats')` → `'fat':2 'rat':3`

`to_tsvector` ([*config* `regconfig`,] *document* `json`) → `tsvector`
`to_tsvector` ([*config* `regconfig`,] *document* `jsonb`) → `tsvector`
Преобразует каждое строковое значение в документе JSON в значение `tsvector`, выполняя нормализацию слов согласно конфигурации по умолчанию или указанной явно. В результате выдаются полученные значения, соединённые вместе в порядке следования в документе. При вычислении выдаваемых позиций слов считается, что между каждой парой строковых значений находится одно стоп-слово. (Учтите, что в случае с `jsonb` «порядок следования в документе» полей объекта JSON зависит от реализации; это наглядно показано в примере.)

Функция **Описание** **Пример(ы)**
`to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::json)` → `'dog':5 'fat':2 'rat':3` `to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::jsonb)` → `'dog':1 'fat':4 'rat':5`

`json_to_tsvector` ([*config* `regconfig`,] *document* `json`, *filter* `jsonb`) → `tsvector`
`jsonb_to_tsvector` ([*config* `regconfig`,] *document* `jsonb`, *filter* `jsonb`) → `tsvector`
Выбирает из JSON-документа все элементы, соответствующие фильтру *`filter`*, и преобразует каждый в значение `tsvector`, нормализуя их согласно конфигурации по умолчанию или указанной явно. В результате выдаются полученные значения, соединённые вместе в порядке следования в документе. При вычислении выдаваемых позиций слов считается, что между каждой парой строковых значений находится одно стоп-слово. (Учтите, что в случае с `jsonb` «порядок следования в документе» полей объекта JSON зависит от реализации.) В параметре *`filter`* должен передаваться массив `jsonb`, содержащий ноль или более следующих ключевых слов: `"string"` (включить все строковые значения), `"numeric"` (все числовые значения), `"boolean"` (все логические значения), `"key"` (все ключи) или `"all"` (включить всё вышеперечисленное). В качестве особого значения *`filter`* принимается простое JSON-значение, содержащее одно из этих ключевых слов.
`json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]')` → `'123':5 'fat':2 'rat':3`
`json_to_tsvector('english', '{"cat": "The Fat Rats", "dog": 123}'::json, '"all"')` → `'123':9 'cat':1 'dog':7 'fat':4 'rat':5`

`ts_delete` (*vector* `tsvector`, *lexeme* `text`) → `tsvector`
Удаляет все вхождения лексемы, задаваемой аргументом *`lexeme`*, из значения *`vector`*. Строка *`lexeme`* принимается в виде лексемы как есть, без дальнейшей обработки.
`ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')` → `'cat':3 'rat':5A`

`ts_delete` (*vector* `tsvector`, *lexemes* `text[]`) → `tsvector`
Удаляет все вхождения лексем, переданных в параметре *`lexemes`*, из параметра *`vector`*. Строки в *`lexemes`* принимаются в виде лексем как есть, без дальнейшей обработки. Строки, которые не соответствуют никаким лексемам в *`vector`*, игнорируются.
`ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat'])` → `'cat':3`

`ts_filter` (*vector* `tsvector`, *weights* `"char"[]`) → `tsvector`
Выбирает из значения *`vector`* только элементы с весами, перечисленными в массиве *`weights`*.
`ts_filter('fat:2,4 cat:3b,7c rat:5A'::tsvector, '{a,b}')` → `'cat':3B 'rat':5A`

`ts_headline` ([*config* `regconfig`,] *document* `text`, *query* `tsquery` [, *options* `text`]) → `text`
Выводит в виде выдержек соответствующие поисковому запросу *`query`* фрагменты содержимого *`document`*, которое должно быть просто текстом, а не значением `tsvector`. Слова в документе перед поиском нормализуются согласно конфигурации по умолчанию или указанной явно. Использование этой функции рассматривается в Подразделе 12.3.4; также там описываются возможные значения *`options`*.
`ts_headline('The fat cat ate the rat.', 'cat')` → `The fat <b>cat</b> ate the rat.`

`ts_headline` ([*config* `regconfig`,] *document* `json`, *query* `tsquery` [, *options* `text`]) → `text` `ts_headline` ([*config* `regconfig`,] *document* `jsonb`, *query* `tsquery` [, *options* `text`]) → `text`

Функция **Описание** **Пример(ы)**
Выводит в виде выдержек соответствующие поисковому запросу *`query`* вхождения, найденные в строковых значениях внутри JSON-документа *`document`*. За подробностями обратитесь к Подразделу 12.3.4. `ts_headline('{"cat":"raining cats and dogs"}'::jsonb, 'cat')` → `{"cat": "raining <b>cats</b> and dogs"}`

`ts_rank` ([*`weights`* `real[]`,] *`vector`* `tsvector`, *`query`* `tsquery` [, *`normalization`* `integer`]) → `real`
Вычисляет оценку, показывающую, в какой степени значение *`vector`* соответствует запросу *`query`*. За подробностями обратитесь к Подразделу 12.3.3.
`ts_rank(to_tsvector('raining cats and dogs'), 'cat')` → `0.06079271`

`ts_rank_cd` ([*`weights`* `real[]`,] *`vector`* `tsvector`, *`query`* `tsquery` [, *`normalization`* `integer`]) → `real`
Вычисляет по алгоритму расчёта плотности покрытия оценку, показывающую, в какой степени значение *`vector`* соответствует запросу *`query`* За подробностями обратитесь к Подразделу 12.3.3.
`ts_rank_cd(to_tsvector('raining cats and dogs'), 'cat')` → `0.1`

`ts_rewrite` (*`query`* `tsquery`, *`target`* `tsquery`, *`substitute`* `tsquery`) → `tsquery`
Заменяет в аргументе *`query`* вхождения *`target`* значением *`substitute`*. За подробностями обратитесь к Подразделу 12.4.2.1.
`ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)` → `'b' & ( 'foo' | 'bar' )`

`ts_rewrite` (*`query`* `tsquery`, *`select`* `text`) → `tsquery`
Заменяет фрагменты запроса *`query`*, извлекая искомое вхождение и замену для него с помощью команды `SELECT`. За подробностями обратитесь к Подразделу 12.4.2.1.
`SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')` → `'b' & ( 'foo' | 'bar' )`

`tsquery_phrase` (*`query1`* `tsquery`, *`query2`* `tsquery`) → `tsquery`
Конструирует фразовый запрос, который будет находить подряд идущие лексемы, удовлетворяющие запросам *`query1`* и *`query2`* (как делает оператор `<->`).
`tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'))` → `'fat' <-> 'cat'`

`tsquery_phrase` (*`query1`* `tsquery`, *`query2`* `tsquery`, *`distance`* `integer`) → `tsquery`
Конструирует фразовый запрос, который будет находить вхождения, удовлетворяющие запросам *`query1`* и *`query2`*, на расстоянии ровно в *`distance`* лексем друг от друга.
`tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10)` → `'fat' <10> 'cat'`

`tsvector_to_array` (`tsvector`) → `text[]`
Преобразует `tsvector` в массив лексем.
`tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector)` → `{cat,fat,rat}`

`unnest` (`tsvector`) → `setof record` (*`lexeme`* `text`, *`positions`* `smallint[]`, *`weights`* `text`) Разворачивает `tsvector` в набор строк, по одной лексеме в строке. `select * from unnest('cat:3 fat:2,4 rat:5A'::tsvector)` →

```
 lexeme | positions | weights
--------+-----------+---------
 cat    | {3}       | {D}
 fat    | {2,4}     | {D,D}
 rat    | {5}       | {A}
```

Примечание

Все функции текстового поиска, принимающие необязательный аргумент `regconfig`, будут использовать конфигурацию, указанную в параметре default_text_search_config, когда этот аргумент опущен.

Функции в Таблице 9.44 перечислены отдельно, так как они не очень полезны в традиционных операциях поиска. Они предназначены в основном для разработки и отладки новых конфигураций текстового поиска.

Таблица 9.44. Функции отладки текстового поиска

Функция **Описание** **Пример(ы)**
`ts_debug` ([*`config`* `regconfig`,] *`document`* `text`) → `setof record` (*`alias`* `text`, *`description`* `text`, *`token`* `text`, *`dictionaries`* `regdictionary[]`, *`dictionary`* `regdictionary`, *`lexemes`* `text[]`) Извлекает из текста *`document`* фрагменты, нормализуя их согласно конфигурации текстового поиска по умолчанию или указанной явно, и выдаёт информацию о том, как был обработан каждый фрагмент. За подробностями обратитесь к Подразделу 12.8.1. `ts_debug('english', 'The Brightest supernovaes')` → `(asciiword,"Word, all ASCII",The,{english_stem},english_stem,{}) ...`

`ts_lexize` (*`dict`* `regdictionary`, *`token`* `text`) → `text[]`
Возвращает массив заменяющих лексем, если слово, заданное аргументом *`token`*, есть в словаре, пустой массив, если это стоп-слово, которое есть в словаре, либо NULL, если такого слова нет. За подробностями обратитесь к Подразделу 12.8.3.
`ts_lexize('english_stem', 'stars')` → `{star}`

`ts_parse` (*`parser_name`* `text`, *`document`* `text`) → `setof record` (*`tokid`* `integer`, *`token`* `text`) Извлекает фрагменты из текста *`document`*, применяя анализатор с указанным именем. За подробностями обратитесь к Подразделу 12.8.2. `ts_parse('default', 'foo - bar')` → `(1,foo) ...`
`ts_parse` (*`parser_oid`* `oid`, *`document`* `text`) → `setof record` (*`tokid`* `integer`, *`token`* `text`) Извлекает фрагменты из текста *`document`*, применяя анализатор с указанным OID. За подробностями обратитесь к Подразделу 12.8.2. `ts_parse(3722, 'foo - bar')` → `(1,foo) ...`
`ts_token_type` (*`parser_name`* `text`) → `setof record` (*`tokid`* `integer`, *`alias`* `text`, *`description`* `text`) Возвращает таблицу, описывающую все типы фрагментов, которые может распознать анализатор с указанным именем. За подробностями обратитесь к Подразделу 12.8.2. `ts_token_type('default')` → `(1,asciiword,"Word, all ASCII") ...`
`ts_token_type` (*`parser_oid`* `oid`) → `setof record` (*`tokid`* `integer`, *`alias`* `text`, *`description`* `text`) Возвращает таблицу, описывающую все типы фрагментов, которые может распознать анализатор с указанным OID. За подробностями обратитесь к Подразделу 12.8.2. `ts_token_type(3722)` → `(1,asciiword,"Word, all ASCII") ...`
`ts_stat` (*`sqlquery`* `text` [, *`weights`* `text`]) → `setof record` (*`word`* `text`, *`ndoc`* `integer`, *`nentry`* `integer`) Выполняет запрос *`sqlquery`*, который должен возвращать единственный столбец `tsvector`, и выдаёт статистику по каждой отдельной лексеме, содержащейся в данных. За подробностями обратитесь к Подразделу 12.4.4.

Функция **Описание** **Пример(ы)**
`ts_stat('SELECT vector FROM apod')` → `(foo,10,15) ...`

9.14. Функции генерирования UUID

В Postgres Pro имеется функция для генерирования UUID:

`gen_random_uuid ()` → `uuid`

Она возвращает случайный UUID версии 4. Это наиболее популярный тип UUID, подходящий для большинства приложений.

Модуль uuid-ossp предоставляет дополнительные функции, реализующие другие стандартные алгоритмы генерирования UUID.

Также есть функции для извлечения данных из UUID:

`uuid_extract_timestamp (uuid)` → `timestamp with time zone`

Эта функция извлекает `timestamp with time zone` из UUID версии 1. Для других версий эта функция возвращает NULL. Обратите внимание, что извлечённая метка времени не обязательно точно равна времени генерации UUID, это зависит от реализации, сгенерировашей UUID.

`uuid_extract_version (uuid)` → `smallint`

Эта функция извлекает версию из UUID варианта, описанного в *RFC 4122*. Для других вариантов эта функция возвращает NULL. Например, для UUID, сгенерированного `gen_random_uuid`, эта функция вернёт 4.

В Postgres Pro также реализованы показанные в Таблице 9.1 операторы для сравнения значений UUID.

9.15. XML-функции

Функции и подобные им выражения, описанные в этом разделе, работают со значениями типа `xml`. Информацию о типе `xml` вы можете найти в Разделе 8.13. Подобные функциям выражения `xmlparse` и `xmlserialize`, преобразующие значения `xml` в текст и обратно, здесь повторно не рассматриваются.

Для использования этих функций Postgres Pro нужно задействовать соответствующую библиотеку при сборке: `configure --with-libxml`.

9.15.1. Создание XML-контента

Для получения XML-контента из данных SQL существует целый набор функций и функциональных выражений, особенно полезных для выдачи клиентским приложениям результатов запроса в виде XML-документов.

9.15.1.1. `xmltext`

`xmltext ( text )` → `xml`

Функция `xmltext` возвращает значение XML с одним текстовым узлом, содержащим входной аргумент в качестве содержимого. Предопределённые сущности, такие как амперсанд (`&`), левые и правые угловые скобки (`< >`) и кавычки (`""`) экранируются.

Пример:

```
SELECT xmltext('< foo & bar >');
         xmltext
-------------------------
 &lt; foo &amp; bar &gt;
```

9.15.1.2. `xmlcomment`

`xmlcomment ( text )` → `xml`

Функция `xmlcomment` создаёт XML-значение, содержащее XML-комментарий с заданным текстом. Этот текст не должен содержать «`--`» или заканчиваться знаком «`-`», в противном случае результат не будет допустимым XML-комментарием. Если аргумент этой функции NULL, результатом её тоже будет NULL.

Пример:

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.15.1.3. `xmlconcat`

`xmlconcat ( xml [, ...] )` → `xml`

Функция `xmlconcat` объединяет несколько XML-значений и выдаёт в результате один фрагмент XML-контента. Значения NULL отбрасываются, так что результат будет равен NULL, только если все аргументы равны NULL.

Пример:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

XML-объявления, если они присутствуют, обрабатываются следующим образом. Если во всех аргументах содержатся объявления одной версии XML, эта версия будет выдана в результате; в противном случае версии не будет. Если во всех аргументах определён атрибут standalone со значением «yes», это же значение будет выдано в результате. Если во всех аргументах есть объявление standalone, но минимум в одном со значением «no», в результате будет это значение. В противном случае в результате не будет объявления standalone. Если же окажется, что в результате должно присутствовать объявление standalone, а версия не определена, тогда в результате будет выведена версия 1.0, так как XML-объявление не будет допустимым без указания версии. Указания кодировки игнорируются и будут удалены в любых случаях.

Пример:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?
><bar/>');

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.15.1.4. `xmlelement`

`xmlelement ( NAME` *`имя`* `[, XMLATTRIBUTES (` *`значение_атрибута`* `[AS` *`атрибут`*`] [, ...] )]`
 `[,` *`содержимое`* `[, ...]] )` → `xml`

Выражение `xmlelement` создаёт XML-элемент с заданным именем, атрибутами и содержимым. Аргументы *`имя`* и *`значение_атрибута`*, показанные в синтаксисе, обозначают простые идентификаторы, а не определённые значения. Аргументы *`значение_атрибута`* и *`содержимое`* являются выражениями, которые могут выдавать любой тип данных Postgres Pro. Аргументы внутри `XMLATTRIBUTES` генерируют атрибуты XML-элемента, к которому также добавляются значения *`содержимое`*, в итоге формируя его содержимое.

Примеры:

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

             xmlelement
-------------------------------------
 <foo bar="2007-01-26">content</foo>
```

Если имена элементов и атрибутов содержат символы, недопустимые в XML, эти символы заменяются последовательностями `_x`*`HHHH`*`_`, где *`HHHH`* — шестнадцатеричный код символа в Unicode. Например:

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

Если в качестве значения атрибута используется столбец таблицы, имя атрибута можно не указывать явно, этим именем станет имя столбца. Во всех остальных случаях имя атрибута должно быть определено явно. Таким образом, это выражение допустимо:

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

А следующие варианты — нет:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Содержимое элемента, если оно задано, будет форматировано согласно его типу данных. Когда оно само имеет тип `xml`, из него можно конструировать сложные XML-документы. Например:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Содержимое других типов будет оформлено в виде допустимых символьных данных XML. Это, в частности, означает, что символы <, > и & будут преобразованы в сущности XML. Двоичные данные (данные типа `bytea`) представляются в кодировке base64 или в шестнадцатеричном виде, в зависимости от значения параметра xmlbinary. Следует ожидать, что конкретные представления отдельных типов данных могут быть изменены для приведения преобразований Postgres Pro в соответствие со стандартом SQL:2006 и новее, как описано в Подразделе D.3.1.3.

9.15.1.5. `xmlforest`

`xmlforest` `(` *`содержимое`* `[ AS` *`имя`* `] [, ...] )` → `xml`

Выражение `xmlforest` создаёт последовательность XML-элементов с заданными именами и содержимым. Как и в функции `xmlelement`, каждый аргумент *имя* должен быть простым идентификатором, а выражения *содержимое* могут иметь любой тип данных.

Примеры:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                xmlforest
------------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 ...
```

Как показано во втором примере, имя элемента можно опустить, если источником содержимого служит столбец (в этом случае именем элемента по умолчанию будет имя столбца). В противном случае это имя необходимо указывать.

Имена элементов с символами, недопустимыми для XML, преобразуются так же, как и для `xmlelement`. Данные содержимого тоже приводятся к виду, допустимому для XML (кроме данных, которые уже имеют тип `xml`).

Заметьте, что такие XML-последовательности не являются допустимыми XML-документами, если они содержат больше одного элемента на верхнем уровне, поэтому может иметь смысл вложить выражения `xmlforest` в `xmlelement`.

9.15.1.6. `xmlpi`

`xmlpi ( NAME` *имя* `[,` *содержимое*`] )` → `xml`

Выражение `xmlpi` создаёт инструкцию обработки XML. Как и в функции `xmlelement`, аргумент *имя* должен быть простым идентификатором, а выражения *содержимое* могут иметь любой тип данных. Содержимое, если оно задано, не должно содержать последовательность символов `?>`.

Пример:

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.15.1.7. `xmlroot`

`xmlroot ( xml, VERSION {text|NO VALUE} [, STANDALONE {YES|NO|NO VALUE}] )` → `xml`

Выражение `xmlroot` изменяет свойства корневого узла XML-значения. Если в нём указывается версия, она заменяет значение в объявлении версии корневого узла; также в корневой узел переносится значение свойства standalone.

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
```

```
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.15.1.8. `xmlagg`

`xmlagg ( xml ) → xml`

Функция `xmlagg`, в отличие от других описанных здесь функций, является агрегатной. Она соединяет значения, поступающие на вход агрегатной функции, подобно функции `xmlconcat`, но делает это, обрабатывая множество строк, а не несколько выражений в одной строке. Дополнительно агрегатные функции описаны в Разделе 9.21.

Пример:

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

Чтобы задать порядок сложения элементов, в агрегатный вызов можно добавить предложение `ORDER BY`, описанное в Подразделе 4.2.7. Например:

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

Следующий нестандартный подход рекомендовался в предыдущих версиях и может быть по-прежнему полезен в некоторых случаях:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.15.2. Условия с XML

Описанные в этом разделе выражения проверяют свойства значений `xml`.

9.15.2.1. `IS DOCUMENT`

`xml IS DOCUMENT → boolean`

Выражение `IS DOCUMENT` возвращает true, если аргумент представляет собой правильный XML-документ, false в противном случае (т. е. если это фрагмент содержимого) и NULL, если его аргумент также NULL. Чем документы отличаются от фрагментов содержимого, вы можете узнать в Разделе 8.13.

9.15.2.2. `IS NOT DOCUMENT`

`xml IS NOT DOCUMENT → boolean`

Выражение `IS NOT DOCUMENT` возвращает false, если аргумент представляет собой правильный XML-документ, true в противном случае (т. е. если это фрагмент содержимого) и NULL, если его аргумент — NULL.

9.15.2.3. `XMLEXISTS`

`XMLEXISTS ( text PASSING [BY {REF|VALUE}] xml [BY {REF|VALUE}] ) → boolean`

Функция `xmlexists` вычисляет выражение XPath 1.0 (первый аргумент), используя в качестве элемента контекста переданное XML-значение. Эта функция возвращает false, если в результате этого вычисления выдаётся пустое множество узлов, или true, если выдаётся любое другое значение. Если один из аргументов равен NULL, результатом также будет NULL. Отличный от NULL аргумент, передающий элемент контекста, должен представлять XML-документ, а не фрагмент содержимого или какое-либо значение, недопустимое в XML.

Пример:

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY VALUE
 '<towns><town>Toronto</town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

Предложения `BY REF` и `BY VALUE` в Postgres Pro принимаются, но игнорируются, о чём рассказывается в Подразделе D.3.2.

Согласно стандарту SQL, функция `xmlexists` должна вычислять выражение, используя средства языка XML Query, но Postgres Pro воспринимает только выражения XPath 1.0, что освещается в Подразделе D.3.1.

9.15.2.4. `xml_is_well_formed`

`xml_is_well_formed ( text ) → boolean`

`xml_is_well_formed_document ( text ) → boolean`

`xml_is_well_formed_content ( text ) → boolean`

Эти функции проверяют, представляет ли `текст` правильно оформленный XML, и возвращают соответствующее логическое значение. Функция `xml_is_well_formed_document` проверяет аргумент как правильно оформленный документ, а `xml_is_well_formed_content` — правильно оформленное содержимое. Функция `xml_is_well_formed` может делать первое или второе, в зависимости от значения параметра конфигурации xmloption (`DOCUMENT` или `CONTENT`, соответственно). Это значит, что `xml_is_well_formed` помогает понять, будет ли успешным простое приведение к типу `xml`, тогда как две другие функции проверяют, будут ли успешны соответствующие варианты `XMLPARSE`.

Примеры:

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</pg:foo>');
```

```
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/
stuff">bar</my:foo>');
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

Последний пример показывает, что при проверке также учитываются сопоставления пространств имён.

9.15.3. Обработка XML

Для обработки значений типа `xml` в Postgres Pro представлены функции `xpath` и `xpath_exists`, вычисляющие выражения XPath 1.0, а также табличная функция `XMLTABLE`.

9.15.3.1. `xpath`

`xpath` (*`xpath`* `text`, *`xml`* `xml` [, *`nsarray`* `text[]`]) → `xml[]`

Функция `xpath` вычисляет выражение XPath 1.0 в аргументе *`xpath`* (заданное в виде текста) для заданного XML-значения *`xml`*. Она возвращает массив XML-значений, соответствующих набору узлов, полученному при вычислении выражения XPath. Если выражение XPath выдаёт не набор узлов, а скалярное значение, возвращается массив с одним элементом.

Вторым аргументом должен быть правильно оформленный XML-документ. В частности, в нём должен быть единственный корневой элемент.

В необязательном третьем аргументе функции передаются сопоставления пространств имён. Эти сопоставления должны определяться в двумерном массиве типа `text`, во второй размерности которого 2 элемента (т. е. это должен быть массив массивов, состоящих из 2 элементов). В первом элементе каждого массива определяется псевдоним (префикс) пространства имён, а во втором — его URI. Псевдонимы, определённые в этом массиве, не обязательно должны совпадать с префиксами пространств имён в самом XML-документе (другими словами, для XML-документа и функции `xpath` псевдонимы имеют *локальный* характер).

Пример:

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

Для пространства имён по умолчанию (анонимного) это выражение можно записать так:

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

9.15.3.2. `xpath_exists`

`xpath_exists` (*`xpath`* `text`, *`xml`* `xml` [, *`nsarray`* `text[]`]) → `boolean`

Функция `xpath_exists` представляет собой специализированную форму функции `xpath`. Она возвращает не отдельные XML-значения, удовлетворяющие выражению XPath 1.0, а только один логический признак, показывающий, имеются ли такие значения (то есть выдаёт ли это выражение что-либо, отличное от пустого множества узлов). Эта функция равнозначна стандартному условию `XMLEXISTS`, за исключением того, что она также поддерживает сопоставления пространств имён.

Пример:

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
                     ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.15.3.3. `xmltable`

```
XMLTABLE (
    [XMLNAMESPACES ( uri_пространства_имён AS имя_пространства_имён [, ...] ),]
    выражение_строки PASSING [BY {REF|VALUE}] выражение_документа [BY {REF|VALUE}]
    COLUMNS имя { тип [PATH выражение_столбца] [DEFAULT выражение_по_умолчанию] [NOT
 NULL | NULL]
                  | FOR ORDINALITY }
            [, ...]
) → setof record
```

Выражение `xmltable` строит таблицу из XML-значения, применяя фильтр XPath для извлечения строк и формируя столбцы с заданным определением. Хотя по синтаксису оно подобно функции, оно может применяться только в качестве таблицы в предложении `FROM`.

Необязательное предложение `XMLNAMESPACES` задаёт разделённый запятыми список определений пространств имён, в котором *`uri_пространства_имён`* — выражение типа `text`, а *`имя_пространства_имён`* — простой идентификатор. В нём определяются пространства имён XML, используемые в документе, и их псевдонимы. Определение пространства по умолчанию в настоящее время не поддерживается.

В обязательном аргументе *`выражение_строки`* передаётся выражение XPath 1.0 (в виде значения `text`), которое будет вычисляться для XML-значения *`выражение_документа`*, служащего элементом контекста, и выдавать набор узлов XML. Эти узлы `xmltable` преобразует в выходные строки. Если *`выражение_документа`* — NULL, а также если *`выражение_строки`* выдаёт пустой набор узлов или любое значение, отличное от набора узлов, выходные строки не выдаются.

`Выражение_документа` передаёт элемент контекста для *`выражения_строки`*. Таким элементом должен быть правильно оформленный XML-документ; фрагменты/наборы деревьев не допускаются. Предложения `BY REF` и `BY VALUE` принимаются, но игнорируются, о чём рассказывается в Подразделе D.3.2.

Согласно стандарту SQL, функция `xmltable` должна вычислять выражения, используя средства языка XML Query, но Postgres Pro воспринимает только выражения XPath 1.0; подробнее об этом говорится в Подразделе D.3.1.

В обязательном предложении `COLUMNS` задаётся список столбцов для выходной таблицы. Формат этого предложения показан выше в описании синтаксиса. Для каждого столбца должно задаваться имя и тип данных (если только не указано `FOR ORDINALITY`, что подразумевает тип `integer`). Путь, значение по умолчанию и признак допустимости NULL могут опускаться.

Столбец с признаком `FOR ORDINALITY` будет заполняться номерами строк, начиная с 1, в том порядке, в котором эти строки идут в наборе узлов, представляющем результат *`выражения_строки`*. Признак `FOR ORDINALITY` может быть не больше чем у одного столбца.

Примечание

В XPath 1.0 не определён порядок узлов в наборе, поэтому код, рассчитывающий на определённый порядок результатов, будет зависеть от конкретной реализации. Подробнее об этом рассказывается в Подразделе D.3.1.2.

В *выражении_столбца* задаётся выражение XPath 1.0, которое вычисляется для каждой строки применительно к текущему узлу, полученному в результате вычисления *выражения_строки* и служащему элементом контекста, и выдаёт значение столбца. Если `выражение_столбца` отсутствует, в качестве неявного пути используется имя столбца.

Если выражение XPath возвращает не XML (это может быть строка, логическое или числовое значение в XPath 1.0) и столбец имеет тип Postgres Pro, отличный от `xml`, значение присваивается столбцу так же, как типу Postgres Pro присваивается строковое представление значения. (Если это значение логическое, его строковым представлением будет `1` или `0`, если тип столбца относится к числовой категории, и `true` или `false` в противном случае.)

Если заданное для столбца выражение XPath возвращает непустой набор узлов XML и этот столбец в Postgres Pro имеет тип `xml`, значение ему будет присвоено как есть, когда оно имеет форму документа или содержимого. [1]

Когда выходному столбцу `xml` присваивается не XML-содержимое, оно представляется в виде одного текстового узла, содержащего строковое значение результата. XML-результат, присваиваемый столбцу любого другого типа, должен состоять не более чем из одного узла, иначе выдаётся ошибка. При наличии же ровно одного узла столбцу, с учётом его типа Postgres Pro, присваивается строковое представление этого узла (получаемое по правилам функции `string` в XPath 1.0).

Строковым значением XML-элемента является конкатенация всех текстовых узлов, содержащихся в нём и во вложенных в него элементах, в порядке следования этих узлов в документе. Строковым значением элемента без внутренних текстовых узлов является пустая строка (не `NULL`). Любые атрибуты со свойствами `xsi:nil` игнорируются. Заметьте, что состоящий только из пробельных символов узел `text()` между двумя нетекстовыми элементами при этом сохраняется, и начальные пробелы в узле `text()` не убираются. Правила, определяющие строковые представления для других типов XML-узлов и не XML-значений, можно найти в описании функции `string` языка XPath 1.0.

Представленные здесь правила преобразования не соответствуют в точности тем, что определены в стандарте SQL, о чём рассказывается в Подразделе D.3.1.3.

Когда выражение пути возвращает для данной строки пустой набор узлов (обычно когда нет соответствия этому пути), столбец получает значение `NULL`, если только не задано *выражение_по_умолчанию*. Если же оно задано, столбец получает результат данного выражения.

Указанное *выражение_по_умолчанию* вычисляется не единожды для вызова функции `xmltable`, а каждый раз, когда для столбца требуется очередное значение по умолчанию. Если же это выражение оказывается стабильным или постоянным, повторное вычисление может не выполняться. Это означает, что в *выражении_по_умолчанию* вы можете с пользой применять изменчивые функции, такие как `nextval`.

Столбцы могут иметь признак `NOT NULL`. Если *выражение_столбца* для столбца с признаком `NOT NULL` не соответствует ничему и при этом отсутствует указание `DEFAULT` или *выражение_по_умолчанию* также выдаёт NULL, происходит ошибка.

Примеры:

```
CREATE TABLE xmldata AS SELECT
```

---

[1] Примером формы содержимого является результат, содержащий больше одного элемента-узла на верхнем уровне или какой-либо не пробельный текст вне единственного элемента. Результат выражения XPath может иметь и другую форму, например, если оно выбирает узел атрибута из содержащего этот атрибут элемента. Такой результат будет приведён в форму содержимого, в которой каждый подобный недопустимый узел заменяется строковым представлением, получаемым по правилам функции `string` в XPath 1.0.

```
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
  FROM xmldata,
       XMLTABLE('//ROWS/ROW'
                PASSING data
                COLUMNS id int PATH '@id',
                        ordinality FOR ORDINALITY,
                        "COUNTRY_NAME" text,
                        country_id text PATH 'COUNTRY_ID',
                        size_sq_km float PATH 'SIZE[@unit = "sq_km"]',
                        size_other text PATH
                             'concat(SIZE[@unit!="sq_km"], " ", SIZE[@unit!="sq_km"]/
@unit)',
                        premier_name text PATH 'PREMIER_NAME' DEFAULT 'not specified');

 id | ordinality | COUNTRY_NAME | country_id | size_sq_km |  size_other  | premier_name
----+------------+--------------+------------+------------+--------------
+---------------
  1 |          1 | Australia    | AU         |            |              | not
 specified
  5 |          2 | Japan        | JP         |            | 145935 sq_mi | Shinzo Abe
  6 |          3 | Singapore    | SG         |        697 |              | not
 specified
```

Следующий пример иллюстрирует сложение нескольких узлов text(), использование имени столбца в качестве фильтра XPath и обработку пробельных символов, XML-комментариев и инструкций обработки:

```
CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
   <element>  Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->  bbb<x>xxx</x>CC  </element>
  </root>
$$ AS data;

SELECT xmltable.*
  FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element text);
         element
-------------------------
   Hello2a2   bbbxxxCC
```

Следующий пример показывает, как с помощью предложения `XMLNAMESPACES` можно задать список пространств имён, используемых в XML-документе и в выражениях XPath:

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns" xmlns:B="http://example.com/b">
 <item foo="1" B:bar="2"/>
 <item foo="3" B:bar="4"/>
 <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
  FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                              'http://example.com/b' AS "B"),
             '/x:example/x:item'
                PASSING (SELECT data FROM xmldata)
                COLUMNS foo int PATH '@foo',
                  bar int PATH '@B:bar');
 foo | bar
-----+-----
   1 |   2
   3 |   4
   4 |   5
(3 rows)
```

9.15.4. Отображение таблиц в XML

Следующие функции отображают содержимое реляционных таблиц в значения XML. Их можно рассматривать как средства экспорта в XML:

```
table_to_xml ( table regclass, nulls boolean,
               tableforest boolean, targetns text ) → xml
query_to_xml ( query text, nulls boolean,
               tableforest boolean, targetns text ) → xml
cursor_to_xml ( cursor refcursor, count integer, nulls boolean,
                tableforest boolean, targetns text ) → xml
```

`table_to_xml` отображает в xml содержимое таблицы, имя которой задаётся в параметре *`table`*. Тип `regclass` принимает идентификаторы строк в обычной записи, которые могут содержать указание схемы и кавычки (за подробностями обратитесь к Разделу 8.19). Функция `query_to_xml` выполняет запрос, текст которого передаётся в параметре *`query`*, и отображает в xml результирующий набор. Последняя функция, `cursor_to_xml` выбирает указанное число строк из курсора, переданного в параметре *`cursor`*. Этот вариант рекомендуется использовать с большими таблицами, так как все эти функции создают результирующий xml в памяти.

Если параметр *`tableforest`* имеет значение false, результирующий XML-документ выглядит так:

```
<имя_таблицы>
  <row>
    <имя_столбца1> данные </имя_столбца1>
    <имя_столбца2> данные </имя_столбца2>
  </row>

  <row>
    ...
  </row>

  ...
</имя_таблицы>
```

А если *`tableforest`* равен true, в результате будет выведен следующий фрагмент XML:

```
<имя_таблицы>
  <имя_столбца1> данные </имя_столбца1>
  <имя_столбца2> данные </имя_столбца2>
</имя_таблицы>

<имя_таблицы>
  ...
</имя_таблицы>

...
```

Если имя таблицы неизвестно, например, при отображении результатов запроса или курсора, вместо него в первом случае вставляется `table`, а во втором — `row`.

Выбор между этими форматами остаётся за пользователем. Первый вариант позволяет создать готовый XML-документ, что может быть полезно для многих приложений, а второй удобно применять с функцией `cursor_to_xml`, если её результаты будут собираться в документ позже. Полученный результат можно изменить по вкусу с помощью рассмотренных выше функций создания XML-содержимого, в частности `xmlelement`.

Значения данных эти функции отображают так же, как и ранее описанная функция `xmlelement`.

Параметр *`nulls`* определяет, нужно ли включать в результат значения NULL. Если он установлен, значения NULL в столбцах представляются так:

```
<имя_столбца xsi:nil="true"/>
```

Здесь `xsi` — префикс пространства имён XML Schema Instance. При этом в результирующий XML будет добавлено соответствующее объявление пространства имён. Если же данный параметр равен false, столбцы со значениями NULL просто не будут выводиться.

Параметр *`targetns`* определяет целевое пространство имён для результирующего XML. Если пространство имён не нужно, значением этого параметра должна быть пустая строка.

Следующие функции выдают документы XML Schema, которые содержат схемы отображений, выполняемых соответствующими ранее рассмотренными функциями:

```
table_to_xmlschema ( table regclass, nulls boolean,
                     tableforest boolean, targetns text ) → xml
query_to_xmlschema ( query text, nulls boolean,
                     tableforest boolean, targetns text ) → xml
cursor_to_xmlschema ( cursor refcursor, nulls boolean,
                      tableforest boolean, targetns text ) → xml
```

Чтобы результаты отображения данных в XML соответствовали XML-схемам, важно, чтобы соответствующим функциям передавались одинаковые параметры.

Следующие функции выдают отображение данных в XML и соответствующую XML-схему в одном документе (или фрагменте), объединяя их вместе. Это может быть полезно там, где желательно получить самодостаточные результаты с описанием:

```
table_to_xml_and_xmlschema ( table regclass, nulls boolean,
                             tableforest boolean, targetns text ) → xml
query_to_xml_and_xmlschema ( query text, nulls boolean,
                             tableforest boolean, targetns text ) → xml
```

В дополнение к ним есть следующие функции, способные выдать аналогичные представления для целых схем в базе данных или даже для всей текущей базы:

```
schema_to_xml ( schema name, nulls boolean,
                tableforest boolean, targetns text ) → xml
schema_to_xmlschema ( schema name, nulls boolean,
```

```
                      tableforest boolean, targetns text ) → xml
schema_to_xml_and_xmlschema ( schema name, nulls boolean,
                              tableforest boolean, targetns text ) → xml

database_to_xml ( nulls boolean,
                  tableforest boolean, targetns text ) → xml
database_to_xmlschema ( nulls boolean,
                        tableforest boolean, targetns text ) → xml
database_to_xml_and_xmlschema ( nulls boolean,
                                tableforest boolean, targetns text ) → xml
```

Эти функции пропускают таблицы, которые не может читать текущий пользователь. Функции уровня базы также пропускают схемы, для которых текущий пользователь не имеет права использования (`USAGE`).

Заметьте, что объём таких данных может быть очень большим, а XML должен сформироваться в памяти. Поэтому, вместо того чтобы пытаться отобразить в XML сразу всё содержимое больших схем или баз данных, лучше делать это по таблицам, возможно, даже используя курсор.

Результат отображения содержимого схемы будет выглядеть так:

```
<имя_схемы>

отображение-таблицы1

отображение-таблицы2

...

</имя_схемы>
```

Формат отображения таблицы определяется параметром *`tableforest`*, описанным выше.

Результат отображения содержимого базы данных будет таким:

```
<имя_БД>

<имя_схемы1>
  ...
</имя_схемы1>

<имя_схемы2>
  ...
</имя_схемы2>

...

</имя_БД>
```

Здесь отображение схемы имеет вид, показанный выше.

В качестве примера, иллюстрирующего использование результата этих функций, на Примере 9.1 показано XSLT-преобразование, которое переводит результат функции `table_to_xml_and_xmlschema` в HTML-документ, содержащий таблицу с данными. Подобным образом результаты этих функций можно преобразовать и в другие форматы на базе XML.

Пример 9.1. XSLT-преобразование, переводящее результат SQL/XML в формат HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/
xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/
xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.16. Функции и операторы JSON

В этом разделе описываются:

- функции и операторы, предназначенные для работы с данными JSON
- язык путей SQL/JSON
- функции запросов SQL/JSON

Postgres Pro реализует *модель данных SQL/JSON*, обеспечивая встроенную поддержку типов данных JSON в среде SQL. В этой модели данные представляются последовательностями элементов. Каждый элемент может содержать скалярные значения SQL, дополнительно определённое в SQL/JSON значение null и составные структуры данных, образуемые объектами и массивами JSON. Данная модель по сути формализует модель данных, описанную в спецификации JSON *RFC 7159*.

Поддержка SQL/JSON позволяет обрабатывать данные JSON наряду с обычными данными SQL, используя при этом транзакции, например:

- Загружать данные JSON в базу и сохранять их в обычных столбцах SQL в виде символьных или двоичных строк.
- Создавать объекты и массивы JSON из реляционных данных.
- Обращаться к данным JSON, используя функции запросов SQL/JSON и выражения языка путей SQL/JSON.

Чтобы узнать больше о стандарте SQL/JSON, обратитесь к sqltr-19075-6. Типы JSON, поддерживаемые в Postgres Pro, описаны в Разделе 8.14.

9.16.1. Обработка и создание данных JSON

Примечание

Функции, работающие с JSONB, не принимают символы `'\u0000'`. Чтобы избежать ошибок и заменять их на лету, необходимо указать символ Unicode в параметре конфигурации unicode_nul_character_replacement_in_jsonb.

В Таблице 9.45 показаны имеющиеся операторы для работы с данными JSON (см. Раздел 8.14). Кроме них для типа `jsonb`, но не для `json`, определены обычные операторы сравнения, показанные в Таблице 9.1. Они следуют правилам упорядочивания для операций B-дерева, описанным в Подразделе 8.14.4. В Разделе 9.21 вы также можете узнать об агрегатной функции `json_agg`, которая агрегирует значения записи в виде JSON, и агрегатной функции `json_object_agg`, агрегирующей пары значений в объект JSON, а также их аналогах для `jsonb`, функциях `jsonb_agg` и `jsonb_object_agg`.

Таблица 9.45. Операторы для типов `json` и `jsonb`

Оператор	Описание	Пример(ы)
`json -> integer` → `json` `jsonb -> integer` → `jsonb`	Извлекает *n*-ый элемент JSON-массива (элементы массива нумеруются с 0, а отрицательные числа задают позиции с конца).	`'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> 2` → `{"c":"baz"}` `'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> -3` → `{"a":"foo"}`
`json -> text` → `json` `jsonb -> text` → `jsonb`	Извлекает поле JSON-объекта по заданному ключу.	`'{"a": {"b":"foo"}}'::json -> 'a'` → `{"b":"foo"}`
`json ->> integer` → `text` `jsonb ->> integer` → `text`	Извлекает *n*-ый элемент из JSON-массива, в виде значения `text`.	`'[1,2,3]'::json ->> 2` → `3`
`json ->> text` → `text` `jsonb ->> text` → `text`	Извлекает поле JSON-объекта по заданному ключу, в виде значения `text`.	`'{"a":1,"b":2}'::json ->> 'b'` → `2`
`json #> text[]` → `json` `jsonb #> text[]` → `jsonb`	Извлекает внутренний JSON-объект по заданному пути, элементами которого могут быть индексы массивов или ключи.	

Оператор **Описание** **Пример(ы)**
`'{"a": {"b": ["foo","bar"]}}'::json #> '{a,b,1}'` → `"bar"`

`json #>> text[]` → `text`
`jsonb #>> text[]` → `text`
Извлекает внутренний JSON-объект по заданному пути в виде значения `text`.
`'{"a": {"b": ["foo","bar"]}}'::json #>> '{a,b,1}'` → `bar`

Примечание

Если структура входного JSON не соответствует запросу, например указанный ключ или элемент массива отсутствует, операторы извлечения поля/элемента/пути не выдают ошибку, а возвращают NULL.

Некоторые из следующих операторов существуют только для `jsonb`, как показано в Таблице 9.46. В Подразделе 8.14.4 описано, как эти операторы могут использоваться для эффективного поиска в индексированных данных `jsonb`.

Таблица 9.46. Дополнительные операторы `jsonb`

Оператор **Описание** **Пример(ы)**

`jsonb @> jsonb` → `boolean`
Первое значение JSON содержит второе? (Что означает «содержит», подробно описывается в Подразделе 8.14.3.)
`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb` → `t`

`jsonb <@ jsonb` → `boolean`
Первое значение JSON содержится во втором?
`'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb` → `t`

`jsonb ? text` → `boolean`
Текстовая строка присутствует в значении JSON в качестве ключа верхнего уровня или элемента массива?
`'{"a":1, "b":2}'::jsonb ? 'b'` → `t`
`'["a", "b", "c"]'::jsonb ? 'b'` → `t`

`jsonb ?| text[]` → `boolean`
Какие-либо текстовые строки из массива присутствуют в качестве ключей верхнего уровня или элементов массива?
`'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'd']` → `t`

`jsonb ?& text[]` → `boolean`
Все текстовые строки из массива присутствуют в качестве ключей верхнего уровня или элементов массива?
`'["a", "b", "c"]'::jsonb ?& array['a', 'b']` → `t`

`jsonb || jsonb` → `jsonb` Соединяет два значения `jsonb`. При соединении двух массивов получается массив, содержащий все их элементы. При соединении двух объектов получается объект с объединённым набором ключей и значений, при этом в случае совпадения ключей выбирается значение из второго объекта. Все другие варианты соединения реализуются путём преобразования аргументов, отличных от массивов, в массивы с одним элементом, которые

Оператор **Описание** **Пример(ы)**

затем как массивы и соединяются. Эта операция не рекурсивна — объединение производится только на верхнем уровне структуры объекта или массива.
`'["a", "b"]'::jsonb || '["a", "d"]'::jsonb` → `["a", "b", "a", "d"]`
`'{"a": "b"}'::jsonb || '{"c": "d"}'::jsonb` → `{"a": "b", "c": "d"}`
`'[1, 2]'::jsonb || '3'::jsonb` → `[1, 2, 3]`
`'{"a": "b"}'::jsonb || '42'::jsonb` → `[{"a": "b"}, 42]`
Чтобы вставить один массив в другой в качестве массива, поместите его в дополнительный массив, например:
`'[1, 2]'::jsonb || jsonb_build_array('[3, 4]'::jsonb)` → `[1, 2, [3, 4]]`

`jsonb - text` → `jsonb`
Удаляет ключ (и его значение) из JSON-объекта или соответствующие строковые значения из JSON-массива.
`'{"a": "b", "c": "d"}'::jsonb - 'a'` → `{"c": "d"}`
`'["a", "b", "c", "b"]'::jsonb - 'b'` → `["a", "c"]`

`jsonb - text[]` → `jsonb`
Удаляет из левого операнда все перечисленные ключи или элементы массива.
`'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[]` → `{}`

`jsonb - integer` → `jsonb`
Удаляет из массива элемент в заданной позиции (отрицательные номера позиций отсчитываются от конца). Выдаёт ошибку, если переданное значение JSON — не массив.
`'["a", "b"]'::jsonb - 1` → `["a"]`

`jsonb #- text[]` → `jsonb`
Удаляет поле или элемент массива с заданным путём, в составе которого могут быть индексы массивов или ключи.
`'["a", {"b":1}]'::jsonb #- '{1,b}'` → `["a", {}]`

`jsonb @? jsonpath` → `boolean`
Возвращает ли путь JSON какой-либо элемент для указанного значения JSON? (Это полезно только для выражений пути JSON по стандарту SQL, но не для выражений проверки предикатов, поскольку они всегда возвращают значение.)
`'{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)'` → `t`

`jsonb @@ jsonpath` → `boolean`
Возвращает результат проверки предиката пути JSON для указанного значения JSON. (Это полезно только для выражений проверки предикатов, но не для выражений пути JSON по стандарту SQL, поскольку возвращается `NULL`, если результат пути не является одним логическим значением.)
`'{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2'` → `t`

Примечание

Операторы `jsonpath` `@?` и `@@` подавляют следующие ошибки: отсутствие поля объекта или элемента массива, несовпадение типа элемента JSON и ошибки в числах и дате/времени. Описанные ниже функции, связанные с `jsonpath`, тоже могут подавлять ошибки такого рода. Это может быть полезно, когда нужно произвести поиск по набору документов JSON, имеющих различную структуру.

В Таблице 9.47 показаны функции, позволяющие создавать значения типов `json` и `jsonb`. Для некоторых функций в этой таблице имеется предложение `RETURNING`, которое определяет возвращаемый тип данных. Это должен быть `json`, `jsonb`, `bytea`, тип символьной строки (`text`, `char` или `varchar`) или тип, который можно привести к `json`. По умолчанию возвращается тип `json`.

Таблица 9.47. Функции для создания JSON

Функция **Описание** **Пример(ы)**

`to_json` (`anyelement`) → `json`
`to_jsonb` (`anyelement`) → `jsonb`
Преобразует произвольное SQL-значение в `json` или `jsonb`. Массивы и составные структуры преобразуются рекурсивно в массивы и объекты (многомерные массивы становятся в JSON массивами массивов). Для других типов, для которых определено приведение к `json`, применяется эта функция приведения;[a] для всех остальных выдаётся скалярное значение. Значения всех скалярных типов, кроме числового, логического и NULL, представляются в текстовом виде; при этом может добавляться экранирование символов, необходимое для получения допустимых строковых значений `json` или `jsonb`.
`to_json('Fred said "Hi."'::text)` → `"Fred said \"Hi.\""`
`to_jsonb(row(42, 'Fred said "Hi."'::text))` → `{"f1": 42, "f2": "Fred said \"Hi.\""}`

`array_to_json` (`anyarray` [, `boolean`]) → `json`
Преобразует массив SQL в JSON-массив. Эта функция работает так же, как `to_json`, но если в необязательном логическом параметре передаётся true, между элементами массива верхнего уровня дополнительно добавляются переводы строк.
`array_to_json('{{1,5},{99,100}}'::int[])` → `[[1,5],[99,100]]`

`json_array` ([{ *выражение_значения* [`FORMAT JSON`] } [, ...]] [{ `NULL` | `ABSENT` } `ON NULL`] [`RETURNING` *тип_данных* [`FORMAT JSON` [`ENCODING UTF8`]]])
`json_array` ([*выражение_запроса*] [`RETURNING` *тип_данных* [`FORMAT JSON` [`ENCODING UTF8`]]])
Создаёт массив JSON либо из набора параметров *выражения_значения* , либо из результатов *выражения_запроса* , которое должно быть запросом SELECT, возвращающим один столбец. С указанием `ABSENT ON NULL` значения NULL игнорируются. Такие значения всегда игнорируются, если используется *выражение_запроса* .
`json_array(1,true,json '{"a":null}')` → `[1, true, {"a":null}]`
`json_array(SELECT * FROM (VALUES(1),(2)) t)` → `[1, 2]`

`row_to_json` (`record` [, `boolean`]) → `json`
Преобразует составное значение SQL в JSON-объект. Эта функция работает так же, как `to_json`, но если в необязательном логическом параметре передаётся true, между элементами верхнего уровня дополнительно добавляются переводы строк.
`row_to_json(row(1,'foo'))` → `{"f1":1,"f2":"foo"}`

`json_build_array` (`VARIADIC "any"`) → `json`
`jsonb_build_array` (`VARIADIC "any"`) → `jsonb`
Формирует JSON-массив (возможно, разнородный) из переменного списка аргументов. Каждый аргумент преобразуется методом `to_json` или `to_jsonb` .
`json_build_array(1, 2, 'foo', 4, 5)` → `[1, 2, "foo", 4, 5]`

`json_build_object` (`VARIADIC "any"`) → `json`
`jsonb_build_object` (`VARIADIC "any"`) → `jsonb`
Формирует JSON-объект из переменного списка аргументов. По соглашению в этом списке перечисляются по очереди ключи и значения. Аргументы, задающие ключи, приводятся к текстовому типу, а аргументы-значения преобразуются методом `to_json` или `to_jsonb` .

Функция **Описание** **Пример(ы)**
`json_build_object('foo', 1, 2, row(3,'bar'))` → `{"foo" : 1, "2" : {"f1":3,"f2":"bar"}}`
`json_object` ([{ *выражение_ключа* { `VALUE` | ':' } *выражение_значения* [`FORMAT JSON` [`ENCODING UTF8`]] }[, ...]] [{ `NULL` | `ABSENT` } `ON NULL`] [{ `WITH` | `WITHOUT` } `UNIQUE` [`KEYS`]] [`RETURNING` *тип_данных* [`FORMAT JSON` [`ENCODING UTF8`]]]) Создаёт объект JSON из всех заданных пар ключ/значение или пустой объект, если ни одна пара не задана. В аргументе *выражение_ключа* передаётся скалярное выражение, определяющее ключ JSON, который преобразуется в тип `text`. Этот параметр не может быть `NULL` и не должен иметь тип, приводимый к `json`. С указанием `WITH UNIQUE KEYS` в *выражении_ключа* не должно быть дублирующихся значений. С указанием `ABSENT ON NULL` вся пара пропускается, если *выражение_значения* равно `NULL`. Если указать `NULL ON NULL` или опустить предложение, ключ определяется со значением `NULL`. `json_object('code' VALUE 'P123', 'title': 'Jaws')` → `{"code" : "P123", "title" : "Jaws"}`

`json_object` (`text[]`) → `json`
`jsonb_object` (`text[]`) → `jsonb`
Формирует объект JSON из текстового массива. Этот массив должен иметь либо одну размерность с чётным числом элементов (в этом случае они воспринимаются как чередующиеся ключи/значения), либо две размерности и при этом каждый внутренний массив содержит ровно два элемента, которые воспринимаются как пара ключ/значение. Все значения преобразуются в строки JSON.
`json_object('{a, 1, b, "def", c, 3.5}')` → `{"a" : "1", "b" : "def", "c" : "3.5"}`
`json_object('{{a, 1}, {b, "def"}, {c, 3.5}}')` → `{"a" : "1", "b" : "def", "c" : "3.5"}`

`json_object` (*keys* `text[]`, *values* `text[]`) → `json`
`jsonb_object` (*keys* `text[]`, *values* `text[]`) → `jsonb`
Эта форма `json_object` принимает ключи и значения по парам из двух отдельных текстовых массивов. В остальных отношениях она не отличается от вариации с одним аргументом.
`json_object('{a,b}', '{1,2}')` → `{"a": "1", "b": "2"}`

`json` (*выражение* [`FORMAT JSON` [`ENCODING UTF8`]] [{ `WITH` | `WITHOUT` } `UNIQUE` [`KEYS`]]) → `json`
Преобразует выражение, заданное в виде строки типа `text` или `bytea` (в кодировке UTF-8), в значение JSON. Если в *выражении* передаётся NULL, возвращается значение SQL NULL. С указанием `WITH UNIQUE` в *выражении* не должно быть объектов с дублирующимися ключами.
`json('{"a":123, "b":[true,"foo"], "a":"bar"}')` → `{"a":123, "b":[true, "foo"], "a":"bar"}`

`json_scalar` (*выражение*)
Преобразует заданное скалярное значение SQL в скалярное значение JSON. Если передаётся NULL, возвращается SQL NULL. Если передаётся число или логическое значение, возвращается соответствующее числовое или логическое значение JSON. Для любого другого значения возвращается строка JSON.
`json_scalar(123.45)` → `123.45`
`json_scalar(CURRENT_TIMESTAMP)` → `"2022-05-10T10:51:04.62128-04:00"`

`json_serialize` (*выражение* [`FORMAT JSON` [`ENCODING UTF8`]] [`RETURNING` *тип_данных* [`FORMAT JSON` [`ENCODING UTF8`]]])

Функция **Описание** **Пример(ы)**
Преобразует выражение SQL/JSON в символьную или двоичную строку. Аргумент *выражение* может быть любого типа JSON, любого типа символьных строк или `bytea` в кодировке UTF-8. Возвращаемый тип в `RETURNING` может быть любым типом символьных строк или `bytea`, по умолчанию — `text`. `json_serialize('{ "a" : 1 } ' RETURNING bytea)` → `\x7b20226122203a2031207d20`

[a] Например, в расширении hstore определено преобразование из `hstore` в `json`, так что значения `hstore`, преобразуемые функциями создания JSON, будут представлены в виде объектов JSON, а не как примитивные строковые значения.

В Таблице 9.48 описаны средства SQL/JSON для проверки JSON.

Таблица 9.48. Функции проверки SQL/JSON

Сигнатура функции **Описание** **Пример(ы)**
выражение `IS` [`NOT`] `JSON` [{ `VALUE` | `SCALAR` | `ARRAY` | `OBJECT` }] [{ `WITH` | `WITHOUT` } `UNIQUE` [`KEYS`]] Этот предикат проверяет, может ли аргумент *выражение* представлять собой JSON (возможно, указанного типа). Если указано `SCALAR`, `ARRAY` или `OBJECT`, проверяется, относится ли JSON к этому конкретному типу. С указанием `WITH UNIQUE KEYS` все объекты в *выражении* также проверяются на наличие дубликатов ключей.

```
SELECT js,
  js IS JSON "json?",
  js IS JSON SCALAR "scalar?",
  js IS JSON OBJECT "object?",
  js IS JSON ARRAY "array?"
FROM (VALUES
      ('123'), ('"abc"'), ('{"a": "b"}'), ('[1,2]'),('abc')) foo(js);
     js     | json? | scalar? | object? | array?
------------+-------+---------+---------+--------
 123        | t     | t       | f       | f
 "abc"      | t     | t       | f       | f
 {"a": "b"} | t     | f       | t       | f
 [1,2]      | t     | f       | f       | t
 abc        | f     | f       | f       | f
```

```
SELECT js,
  js IS JSON OBJECT "object?",
  js IS JSON ARRAY "array?",
  js IS JSON ARRAY WITH UNIQUE KEYS "array w. UK?",
  js IS JSON ARRAY WITHOUT UNIQUE KEYS "array w/o UK?"
FROM (VALUES ('[{"a":"1"},
 {"b":"2","b":"3"}]')) foo(js);
-[ RECORD 1 ]-+--------------------
js            | [{"a":"1"},        +
              |  {"b":"2","b":"3"}]
object?       | f
array?        | t
array w. UK?  | f
array w/o UK? | t
```

В Таблице 9.49 показаны функции, предназначенные для работы со значениями `json` и `jsonb`.

Таблица 9.49. Функции для обработки JSON

Функция
Описание
Пример(ы)

`json_array_elements` (`json`) → `setof json`

`jsonb_array_elements` (`jsonb`) → `setof jsonb`

Разворачивает JSON-массив верхнего уровня в набор значений JSON.

`select * from json_array_elements('[1,true, [2,false]]')` →

```
   value
-----------
 1
 true
 [2,false]
```

`json_array_elements_text` (`json`) → `setof text`

`jsonb_array_elements_text` (`jsonb`) → `setof text`

Разворачивает JSON-массив верхнего уровня в набор значений `text`.

`select * from json_array_elements_text('["foo", "bar"]')` →

```
   value
-----------
 foo
 bar
```

`json_array_length` (`json`) → `integer`

`jsonb_array_length` (`jsonb`) → `integer`

Возвращает число элементов во внешнем JSON-массиве верхнего уровня.

`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')` → `5`

`jsonb_array_length('[]')` → `0`

`json_each` (`json`) → `setof record` (*`key`* `text`, *`value`* `json`)

`jsonb_each` (`jsonb`) → `setof record` (*`key`* `text`, *`value`* `jsonb`)

Разворачивает JSON-объект верхнего уровня в набор пар ключ/значение (key/value).

`select * from json_each('{"a":"foo", "b":"bar"}')` →

```
 key | value
-----+-------
 a   | "foo"
 b   | "bar"
```

`json_each_text` (`json`) → `setof record` (*`key`* `text`, *`value`* `text`)

`jsonb_each_text` (`jsonb`) → `setof record` (*`key`* `text`, *`value`* `text`)

Разворачивает JSON-объект верхнего уровня в набор пар ключ/значение (key/value). Возвращаемые значения будут иметь тип `text`.

`select * from json_each_text('{"a":"foo", "b":"bar"}')` →

```
 key | value
-----+-------
 a   | foo
 b   | bar
```

`json_extract_path` (*`from_json`* `json`, `VARIADIC` *`path_elems`* `text[]`) → `json`

`jsonb_extract_path` (*`from_json`* `jsonb`, `VARIADIC` *`path_elems`* `text[]`) → `jsonb`

Извлекает внутренний JSON-объект по заданному пути. (То же самое делает оператор `#>`, но в некоторых случаях может быть удобнее записать путь в виде списка отдельных аргументов.)

Функция
Описание
Пример(ы)

`json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6')` → `"foo"`

`json_extract_path_text` (*`from_json`* `json`, `VARIADIC` *`path_elems`* `text[]`) → `text`

`jsonb_extract_path_text` (*`from_json`* `jsonb`, `VARIADIC` *`path_elems`* `text[]`) → `text`

Извлекает внутренний JSON-объект по заданному пути в виде значения `text`. (То же самое делает оператор `#>>`.)

`json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6')` → `foo`

`json_object_keys` (`json`) → `setof text`

`jsonb_object_keys` (`jsonb`) → `setof text`

Выдаёт множество ключей в JSON-объекте верхнего уровня.

`select * from json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}')` →

```
 json_object_keys
------------------
 f1
 f2
```

`json_populate_record` (*`base`* `anyelement`, *`from_json`* `json`) → `anyelement`

`jsonb_populate_record` (*`base`* `anyelement`, *`from_json`* `jsonb`) → `anyelement`

Разворачивает JSON-объект верхнего уровня в строку, имеющую составной тип аргумента *`base`*. В JSON-объекте просматриваются поля, имена которых соответствуют именам столбцов выходного типа, и их значения вставляются в эти столбцы результата. (Поля, не соответствующие именам никаких выходных столбцов, пропускаются.) Обычно в *`base`* просто передаётся `NULL`, что означает, что выходные столбцы, которым не нашлось соответствие в объекте, получат значения `NULL`. Однако если в аргументе *`base`* передаётся не `NULL`, то для таких столбцов будут использованы значения из этого аргумента.

Для преобразования значения JSON в SQL-тип выходного столбца последовательно применяются следующие правила:

- Значение NULL в JSON всегда преобразуется в SQL NULL.
- Если выходной столбец имеет тип `json` или `jsonb`, значение JSON воспроизводится без изменений.
- Если выходной столбец имеет составной тип (тип кортежа) и значение JSON является объектом JSON, поля этого объекта преобразуются в столбцы типа выходного кортежа в результате рекурсивного применения этих правил.
- Подобным образом, если выходной столбец имеет тип-массив и значение JSON представляет массив JSON, элементы данного массива преобразуются в элементы выходного массива в результате рекурсивного применения этих правил.
- Если же значение JSON является строкой, содержимое этой строки передаётся входной функции преобразования для типа данных целевого столбца.
- В противном случае функции преобразования для типа данных целевого столбца передаётся обычное текстовое представление значения JSON.

В следующем примере значение JSON фиксировано, но обычно такая функция обращается с использованием `LATERAL` к столбцу `json` или `jsonb` из другой таблицы, фигурирующей в предложении `FROM`. Функция `json_populate_record` в предложении `FROM` будет работать эффективно, так как все извлечённые столбцы можно использовать, не выполняя повторные вызовы функции.

`create type subrowtype as (d int, e text); create type myrowtype as (a int, b text[], c subrowtype);`

`select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}, "x": "foo"}')` →

Функция
Описание
Пример(ы)

```
 a |     b     |      c
---+-----------+-------------
 1 | {2,"a b"} | (4,"a b c")
```

`jsonb_populate_record_valid` (*base* `anyelement`, *from_json* `json`) → `boolean`

Функция для тестирования `jsonb_populate_record`. Возвращает `true`, если `jsonb_populate_record` завершится без ошибки для данного входного объекта JSON, то есть ввод был допустимым, или `false` в противном случае.

`CREATE TYPE jsb_char2 AS (a char(2));`

`SELECT jsonb_populate_record_valid(NULL::jsb_char2, '{"a": "aaa"}');` →

```
 jsonb_populate_record_valid
-----------------------------
 f
(1 row)
```

`SELECT * FROM jsonb_populate_record(NULL::jsb_char2, '{"a": "aaa"}') q;` →

```
ОШИБКА:  значение не умещается в тип character(2)
```

`SELECT jsonb_populate_record_valid(NULL::jsb_char2, '{"a": "aa"}');` →

```
 jsonb_populate_record_valid
-----------------------------
 t
(1 row)
```

`SELECT * FROM jsonb_populate_record(NULL::jsb_char2, '{"a": "aa"}') q;` →

```
 a
----
 aa
(1 row)
```

`json_populate_recordset` (*base* `anyelement`, *from_json* `json`) → `setof anyelement`

`jsonb_populate_recordset` (*base* `anyelement`, *from_json* `jsonb`) → `setof anyelement`

Разворачивает JSON-массив верхнего уровня с объектами в набор строк, имеющих составной тип аргумента *base*. Каждый элемент JSON-массива обрабатывается так же, как было описано выше для `json[b]_populate_record`.

`create type twoints as (a int, b int);`

`select * from json_populate_recordset(null::twoints, '[{"a":1,"b":2}, {"a":3,"b":4}]')` →

```
 a | b
---+---
 1 | 2
 3 | 4
```

`json_to_record` (`json`) → `record`

`jsonb_to_record` (`jsonb`) → `record`

Разворачивает JSON-объект верхнего уровня в строку, имеющую составной тип, определённый в предложении `AS`. (Как и со всеми функциями, возвращающими значение `record`, вызывающий запрос должен явно определять структуру записи в `AS`.) Выходная запись заполняется полями JSON-объекта так же, как было описано выше для `json[b]_populate_record`. Так как этой функции не передаётся запись, столбцы, для которых не находится соответствие, всегда получают значения `NULL`.

`create type myrowtype as (a int, b text);`

Функция
Описание
Пример(ы)

`select * from json_to_record('{"a":1,"b":[1,2,3],"c":[1,2,3], "e":"bar","r": {"a": 123, "b": "a b c"}}') as x(a int, b text, c int[], d text, r myrowtype)` →

```
 a |    b    |    c    | d |       r
---+---------+---------+---+---------------
 1 | [1,2,3] | {1,2,3} |   | (123,"a b c")
```

`json_to_recordset` (`json`) → `setof record`

`jsonb_to_recordset` (`jsonb`) → `setof record`

Разворачивает JSON-массив верхнего уровня с объектами в набор строк, имеющих составной тип, определённый в предложении `AS`. (Как и со всеми функциями, возвращающими значение `record`, вызывающий запрос должен явно определять структуру записи в `AS`.) Каждый элемент JSON-массива обрабатывается так же, как было описано выше для `json[b]_populate_record`.

`select * from json_to_recordset('[{"a":1,"b":"foo"}, {"a":"2", "c":"bar"}]') as x(a int, b text)` →

```
 a |  b
---+-----
 1 | foo
 2 |
```

`jsonb_set` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`create_if_missing`* `boolean`]) → `jsonb`

Возвращает объект *`target`*, в котором элемент, на который указывает путь *`path`*, заменяется значением *`new_value`* либо значение *`new_value`* добавляется, когда параметр *`create_if_missing`* равен true (это значение по умолчанию) и элемент, на который указывает *`path`*, не существует. Чтобы это изменение произошло, все предыдущие элементы, на которые указывает путь, должны существовать. В противном случае *`target`* возвращается без изменений. Как и с операторами, принимающими пути, отрицательные целые числа, фигурирующие в *`path`*, отсчитывают элементы с конца JSON-массива. Если на последнем шаге пути указывается индекс, выходящий за границы массива, и параметр *`create_if_missing`* равен true, новое значение добавляется в начало массива, когда индекс отрицательный, или в конец, когда он положительный.

`jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', '[2,3,4]', false)` → `[{"f1": [2, 3, 4], "f2": null}, 2, null, 3]`

`jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}', '[2,3,4]')` → `[{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]`

`jsonb_set_lax` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`create_if_missing`* `boolean` [, *`null_value_treatment`* `text`]]) → `jsonb`

Если значение *`new_value`* отлично от `NULL`, эта функция действует так же, как и `jsonb_set`. В противном случае она действует согласно значению *`null_value_treatment`*, которое может принимать значение `'raise_exception'`, `'use_json_null'`, `'delete_key'` или `'return_target'`. Значение по умолчанию: `'use_json_null'`.

`jsonb_set_lax('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', null)` → `[{"f1": null, "f2": null}, 2, null, 3]`

`jsonb_set_lax('[{"f1":99,"f2":null},2]', '{0,f3}', null, true, 'return_target')` → `[{"f1": 99, "f2": null}, 2]`

`jsonb_insert` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`insert_after`* `boolean`]) → `jsonb`

Функция **Описание** **Пример(ы)**

Возвращает объект *target* с вставленном в него значением *new_value*. Когда элемент, на который указывает путь *path*, является элементом массива, *new_value* вставляется перед этим элементом, если параметр *insert_after* равен false (по умолчанию), либо после него, если *insert_after* равен true. Когда элемент, на который указывает *path*, является полем объекта, *new_value* будет вставлено только если у объекта ещё нет такого ключа. Чтобы это изменение произошло, все предыдущие элементы, на которые указывает путь, должны существовать. В противном случае *target* возвращается без изменений. Как и с операторами, принимающими пути, отрицательные целые числа, фигурирующие в *path*, отсчитывают элементы с конца JSON-массива. Если на последнем шаге пути указывается индекс, выходящий за границы массива, новое значение добавляется в начало массива, когда индекс отрицательный, или в конец, когда он положительный.
`jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"')` → `{"a": [0, "new_value", 1, 2]}`
`jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true)` → `{"a": [0, 1, "new_value", 2]}`

`json_strip_nulls` (`json`) → `json`
`jsonb_strip_nulls` (`jsonb`) → `jsonb`
Удаляет из данного значения JSON все поля объектов, имеющие значения null, на всех уровнях вложенности. Значения null, не относящиеся к полям объектов, сохраняются без изменений.
`json_strip_nulls('[{"f1":1, "f2":null}, 2, null, 3]')` → `[{"f1":1},2,null,3]`

`jsonb_path_exists` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `boolean`
Определяет, выдаёт ли путь JSON какой-либо элемент при заданном значении JSON. (Это полезно только для выражений пути JSON по стандарту SQL, а не для выражений проверки предикатов, поскольку они всегда возвращают значение.) В случае присутствия аргумента *vars* он должен быть JSON-объектом, поля которого будут подставляться в выражение `jsonpath` под их именами. Если передаётся аргумент *silent* и он равен `true`, функция подавляет те же ошибки, что и операторы `@?` и `@@`.
`jsonb_path_exists('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `t`

`jsonb_path_match` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `boolean`
Возвращает результат SQL типа `boolean` проверки предиката пути JSON для заданного значения JSON. (Это полезно только для выражений проверки предикатов, но не для выражений пути JSON по стандарту SQL, поскольку возвращается `NULL`, если результат пути не является одним логическим значением.) Дополнительные аргументы *vars* и *silent* действуют так же, как и для `jsonb_path_exists`.
`jsonb_path_match('{"a":[1,2,3,4,5]}', 'exists($.a[*] ? (@ >= $min && @ <= $max))', '{"min":2, "max":4}')` → `t`

`jsonb_path_query` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `setof jsonb` Возвращает все элементы JSON, полученные по указанному пути JSON для заданного значения JSON. Для выражений пути JSON по стандарту SQL возвращает значения JSON, выбранные из *target*. Для выражений проверки предикатов возвращает результат проверки предиката: `true`, `false` или `null`. Необязательные аргументы *vars* и *silent* действуют так же, как и для `jsonb_path_exists`. `select * from jsonb_path_query('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` →

Функция
Описание
Пример(ы)

```
 jsonb_path_query
------------------
 2
 3
 4
```

`jsonb_path_query_array` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `jsonb`

Возвращает все элементы JSON, полученные по указанному пути для заданного значения JSON, в виде JSON-массива. Параметры у этой функции те же, что и у `jsonb_path_query` .

`jsonb_path_query_array('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `[2, 3, 4]`

`jsonb_path_query_first` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `jsonb`

Возвращает первый элемент JSON, полученный по указанному пути для заданного значения JSON, либо `NULL`, если этому пути не соответствуют никакие элементы. Параметры у этой функции те же, что и у `jsonb_path_query` .

`jsonb_path_query_first('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `2`

`jsonb_path_exists_tz` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `boolean`

`jsonb_path_match_tz` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `boolean`

`jsonb_path_query_tz` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `setof jsonb`

`jsonb_path_query_array_tz` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `jsonb`

`jsonb_path_query_first_tz` (*target* `jsonb`, *path* `jsonpath` [, *vars* `jsonb` [, *silent* `boolean`]]) → `jsonb`

Эти функции работают подобно их двойникам без суффикса `_tz` , описанным выше, за исключением того, что данные функции поддерживают сравнение значений с датой/временем, для которых должны учитываться часовые пояса. В следующем примере дата без указания времени `2015-08-02` должна преобразоваться в дату/время с часовым поясом, поэтому результат будет зависеть от текущего значения TimeZone. Вследствие этой зависимости данные функции помечены как стабильные (не постоянные), и поэтому их нельзя использовать в индексах. Их двойники являются постоянными и могут использоваться в индексах, но при попытке выполнить такое сравнение они будут выдавать ошибку.

`jsonb_path_exists_tz('["2015-08-01 12:00:00 -05"]', '$[*] ? (@.datetime() < "2015-08-02".datetime())')` → `t`

`jsonb_pretty` (`jsonb`) → `text`

Преобразует данное значение JSON в визуально улучшенное текстовое представление с отступами.

`jsonb_pretty('[{"f1":1,"f2":null}, 2]')` →

```
[
    {
        "f1": 1,
        "f2": null
    },
```

Функция **Описание** **Пример(ы)**
2]

`json_typeof ( json ) → text`
`jsonb_typeof ( jsonb ) → text`
Возвращает тип значения на верхнем уровне JSON в виде текстовой строки. Возможные типы: `object`, `array`, `string`, `number`, `boolean` и `null`. (Не следует путать эту строку `null` с SQL-значением NULL; см. примеры.)
`json_typeof('-123.4')` → `number`
`json_typeof('null'::json)` → `null`
`json_typeof(NULL::json) IS NULL` → `t`

9.16.2. Язык путей SQL/JSON

Выражения путей SQL/JSON определяют элементы, извлекаемые из данных JSON, подобно тому, как выражения XPath позволяют обращаться из SQL к XML. В Postgres Pro выражения путей представляются в виде типа данных `jsonpath` и могут использовать любые элементы, описанные в Подразделе 8.14.7.

Операторы и функции запросов к JSON передают поступившее им выражение *обработчику путей* для вычисления. Если выражению соответствуют фигурирующие в запросе данные JSON, в результате выдаётся соответствующий элемент JSON или набор элементов. Если совпадение не найдено, в зависимости от функции выдаётся `NULL`, `false` или ошибка. Выражения путей записываются на языке путей SQL/JSON и могут включать сложные арифметические выражения и функции.

Выражение пути состоит из последовательности элементов, допустимых для типа `jsonpath`. Обычно оно вычисляется слева направо, но при необходимости порядок операций можно изменить, добавив скобки. В случае успешного вычисления выдаётся последовательность элементов JSON, и результат вычисления возвращается в функцию JSON-запроса, которая завершает обработку выражения.

Для обращения к поступившему в запрос значению JSON (*элементу контекста*) в выражении пути используется переменная `$`. Первый элемент пути всегда должен быть `$`. Затем могут следовать один или более операторов обращения, которые, опускаясь в структуре JSON с одного уровня на другой, извлекают элементы, вложенные в текущий элемент контекста. При этом каждый оператор обращения имеет дело с результатом вычисления, полученным на предыдущем шаге, и выдаёт ноль, один или более выходных элементов на каждый входной элемент.

Допустим, нужно проанализировать данные JSON с GPS-трекера, например:

```
SELECT '{
  "track": {
    "segments": [
      {
        "location":   [ 47.763, 13.4034 ],
        "start time": "2018-10-14 10:05:14",
        "HR": 73
      },
      {
        "location":   [ 47.706, 13.2635 ],
        "start time": "2018-10-14 10:39:21",
        "HR": 135
      }
    ]
  }
}' AS json \gset
```

(Пример выше можно скопировать и вставить в psql, чтобы настроить всё для следующих примеров. Затем psql расширит `:'json'` до строковой константы в соответствующих кавычках, содержащей значение JSON.)

Чтобы получить доступные сегменты отслеживания, используйте оператор доступа `.`*ключ* для спуска по окружающим объектам JSON, например:

```
=> select jsonb_path_query(:'json', '$.track.segments');
jsonb_path_query
-----------------------------------------------------------
------------------------------------------------------------
--------------------------------------------
[{"HR": 73, "location": [47.763, 13.4034], "start time": "2018-10-14 10:05:14"}, {"HR":
 135, "location": [47.706, 13.2635], "start time": "2018-10-14 10:39:21"}]
```

Для извлечения содержимого массива обычно используется оператор `[*]`. В следующем примере возвращаются координаты местоположения для всех доступных сегментов треков:

```
=> select jsonb_path_query(:'json', '$.track.segments[*].location');
jsonb_path_query
-------------------
[47.763, 13.4034]
[47.706, 13.2635]
```

Здесь мы начали с целого входного значения JSON (`$`), затем оператор обращения `.track` выбрал объект JSON, связанный с ключом объекта `"track"`, оператор обращения `.segments` выбрал массив JSON, связанный с ключом `"segments"` внутри этого объекта, оператор обращения `[*]` выбрал каждый элемент этого массива (создав ряд элементов), а оператор обращения `.location` выбрал массив JSON, связанный с ключом `"location"` внутри каждого из этих объектов. В этом примере у каждого из этих объектов был ключ `"location"`, но если бы это было не так, вывод оператора обращения `.location` был бы пустым для этого элемента ввода.

Чтобы получить координаты только первого сегмента, можно задать соответствующий индекс в операторе обращения `[]`. Обратите внимание, что индексы в JSON-массивах отсчитываются с 0:

```
=> select jsonb_path_query(:'json', '$.track.segments[0].location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
```

Результат каждого шага вычисления выражения может быть обработан одним или несколькими операторами и методами `jsonpath`, перечисленными в Подразделе 9.16.2.3. Перед именем метода должна стоять точка. Например, так можно получить размер массива:

```
=> select jsonb_path_query(:'json', '$.track.segments.size()');
 jsonb_path_query
------------------
 2
```

Другие примеры использования операторов и методов `jsonpath` в выражениях пути приведены ниже в Подразделе 9.16.2.3.

Определяя путь, также можно использовать *выражения фильтра*, работающие подобно предложению `WHERE` в SQL. Выражение фильтра начинается со знака вопроса и содержит условие в круглых скобках:

```
? (условие)
```

Выражения фильтра указываются сразу после шага вычисления пути, к которому они должны применяться. Результаты шага проходят через фильтр, и на выходе остаются только те элементы, которые удовлетворяют заданному условию. В SQL/JSON действует троичная логика, то есть результатом проверки условия может быть `true`, `false` или `unknown` (неизвестность). Значение `unknown` играет ту же роль, что и `NULL` в SQL, и может быть проверено предикатом `is unknown`. На последующих шагах вычисления пути будут обрабатываться только те элементы, для которых выражение фильтра выдало `true`.

Функции и операторы, которые можно использовать в выражениях фильтра, перечислены в Таблице 9.51. Переменная `@` в выражении фильтра представляет рассматриваемое значение (результат предыдущего шага в пути). Для получения внутренних элементов этого значения вы можете добавить операторы обращения после `@`.

Например, нужно получить все значения пульса выше 130. Это можно сделать следующим образом:

```
=> select jsonb_path_query(:'json', '$.track.segments[*].HR ? (@ > 130)');
jsonb_path_query
------------------
135
```

Чтобы получить в результате время начала соответствующих сегментов, нужно отфильтровать ненужные сегменты, а затем выбрать время, так что фильтр будет применяться к предыдущему шагу и путь окажется другим:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ? (@.HR > 130)."start time"');
   jsonb_path_query
-----------------------
 "2018-10-14 10:39:21"
```

Можно также использовать несколько выражений фильтра по очереди, когда это требуется. Например, следующее выражение выбирает время начала всех сегментов с определёнными координатами и высоким показателем пульса:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ? (@.location[1] < 13.4) ?
 (@.HR > 130)."start time"');
   jsonb_path_query
-----------------------
 "2018-10-14 10:39:21"
```

Также возможно использовать фильтры на разных уровнях вложенности. В следующем примере сначала сегменты фильтруются по координатам, а затем для подходящих сегментов, если они находятся, выбираются значения высокого пульса:

```
=> select jsonb_path_query(:'json', '$.track.segments[*] ? (@.location[1] < 13.4).HR ?
 (@ > 130)');
 jsonb_path_query
------------------
 135
```

Можно также вкладывать выражения фильтра одно в другое. Данное выражение возвращает количество сегментов в треке, если он содержит сегменты с высокими показателями пульса, или пустую последовательность, если таких сегментов нет:

```
=> select jsonb_path_query(:'json', '$.track ? (exists(@.segments[*] ? (@.HR >
 130))).segments.size()');
 jsonb_path_query
------------------
 2
```

9.16.2.1. Отличия от стандарта SQL

Реализация языка путей SQL/JSON в Postgres Pro имеет следующие отличия от стандарта SQL/JSON.

9.16.2.1.1. Выражения проверки булевых предикатов

Как расширение стандарта SQL, выражение пути Postgres Pro может быть булевым предикатом, тогда как стандарт SQL допускает предикаты только в фильтрах. В то время как выражения пути стандарта SQL возвращают соответствующий элемент(ы) запрошенного значения JSON, выражения проверки предикатов возвращают один из трёх возможных результатов типа `jsonb` проверки предиката: `true`, `false` или `null`. Например, можно написать такое выражение фильтра стандарта SQL:

```
=> select jsonb_path_query(:'json', '$.track.segments ?(@[*].HR > 130)');
                          jsonb_path_query
-----------------------------------------------------------------------------------
 {"HR": 135, "location": [47.706, 13.2635], "start time": "2018-10-14 10:39:21"}
```

Аналогичное выражение проверки предикатов просто возвращает `true`, указывая на то, что совпадение существует:

```
=> select jsonb_path_query(:'json', '$.track.segments[*].HR > 130');
 jsonb_path_query
------------------
 true
```

> ### Примечание
>
> Выражения проверки предикатов обязательны для оператора `@@` (и функции `jsonb_path_match`) и не должны использоваться с оператором `@?` (или функцией `jsonb_path_exists`).

9.16.2.1.2. Интерпретация регулярных выражений

Есть небольшие различия в интерпретации шаблонов регулярных выражений, используемых в фильтрах `like_regex`; имеющиеся особенности описаны в Подразделе 9.16.2.4.

9.16.2.2. Строгий и нестрогий режимы

Когда вы обращаетесь к данным JSON, выражение пути может не соответствовать фактической структуре данных JSON. Попытка обратиться к несуществующему члену объекта или элементу массива определяется как ошибка структурного типа. Для обработки такого рода ошибок в выражениях путей SQL/JSON предусмотрены два режима:

- lax (по умолчанию) — нестрогий режим, в котором обработчик путей неявно адаптирует обрабатываемые данные к указанному пути. Любые структурные ошибки, которые не могут быть устранены указанными ниже способами, подавляются, и запрос не возвращает для них совпадения.
- strict — строгий режим, в котором структурные ошибки выдаются как есть.

Нестрогий режим упрощает сопоставление документа JSON с выражением пути в случаях, когда данные JSON не соответствуют ожидаемой схеме. Если операнд не удовлетворяет требованиям определённой операции, он может перед выполнением этой операции автоматически оборачиваться в массив SQL/JSON или наоборот, разворачиваться так, чтобы его элементы образовали последовательность SQL/JSON. Также в нестрогом режиме операторы сравнения автоматически разворачивают свои операнды, что позволяет легко сравнивать массивы SQL/JSON. Массив с одним элементом в таком режиме считается равным своему элементу. Автоматическое разворачивание не выполняется в следующих случаях:

- В выражении пути фигурируют методы `size()` и `type()`, возвращающие соответственно число элементов в массиве и тип.
- Обрабатываемые данные JSON содержат вложенные массивы. В этом случае разворачивается только массив верхнего уровня, а внутренние массивы остаются без изменений. Таким образом, неявное разворачивание может опускаться на каждом шаге вычисления пути только на один уровень.

Например, обрабатывая данные GPS, показанные выше, в нестрогом режиме можно не обращать внимание на то, что в них содержится массив сегментов:

```
=> select jsonb_path_query(:'json', 'lax $.track.segments.location');
 jsonb_path_query
```

```
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
```

В строгом режиме указанный путь должен точно соответствовать структуре запрашиваемого документа JSON, поэтому использование этого выражения пути вызовет ошибку:

```
=> select jsonb_path_query(:'json', 'strict $.track.segments.location');
ОШИБКА:  выражение обращения к члену в jsonpath может применяться только к объекту
```

Чтобы получить тот же результат, что и в нестрогом режиме, нужно явно развернуть массив `segments`:

```
=> select jsonb_path_query(:'json', 'strict $.track.segments[*].location');
 jsonb_path_query
-------------------
 [47.763, 13.4034]
 [47.706, 13.2635]
```

Поведение разворачивания в нестрогом режиме может выдавать несколько неожиданные результаты. Например, следующий запрос с оператором обращения `.**` выберет каждое значение `HR` дважды:

```
=> select jsonb_path_query(:'json', 'lax $.**.HR');
 jsonb_path_query
------------------
 73
 135
 73
 135
```

Это происходит потому, что оператор `.**` выбирает и массив `segments`, и каждый из его элементов, а обращение `.HR` в нестрогом режиме автоматически разворачивает массивы. Во избежание подобных сюрпризов мы рекомендуем использовать оператор обращения `.**` только в строгом режиме. Следующий запрос выбирает каждое значение `HR` в единственном экземпляре:

```
=> select jsonb_path_query(:'json', 'strict $.**.HR');
 jsonb_path_query
------------------
 73
 135
```

Развёртывание массивов также может привести к неожиданным результатам. Рассмотрим пример, в котором выбираются все массивы `location`:

```
=> select jsonb_path_query(:'json', 'lax $.track.segments[*].location');
jsonb_path_query
-------------------
[47.763, 13.4034]
[47.706, 13.2635]
(2 rows)
```

Как и ожидалось, возвращаются полные массивы. Но применение выражения фильтра приводит к тому, что массивы разворачиваются для оценки каждого элемента, возвращая только те элементы, которые соответствуют выражению:

```
=> select jsonb_path_query(:'json', 'lax $.track.segments[*].location ?(@[*] > 15)');
jsonb_path_query
------------------
47,763
47,706
(2 rows)
```

Это происходит несмотря на то, что выражение пути выбирает полные массивы. Используйте строгий режим для восстановления выбора только массивов:

```
=> select jsonb_path_query(:'json', 'strict $.track.segments[*].location ?(@[*] >
 15)');
jsonb_path_query
-------------------
[47.763, 13.4034]
[47.706, 13.2635]
(2 rows)
```

9.16.2.3. Операторы и методы SQL/JSON

В Таблице 9.50 показаны операторы и методы, поддерживаемые в значениях `jsonpath`. Обратите внимание, что унарные операторы и методы могут применяться к множеству значений, полученных на предыдущем шаге пути, тогда как бинарные операторы (сложение и т. п.) применяются только к отдельным значениям. В нестрогом режиме методы, применяемые к массиву, будут исполняться для каждого значения в массиве. Исключениями являются `.type()` и `.size()`, которые применяются к самому массиву.

Таблица 9.50. Операторы и методы `jsonpath`

Оператор/Метод	Описание	Пример(ы)
число + *число* → *число*	Сложение	`jsonb_path_query('[2]', '$[0] + 3')` → 5
+ *число* → *число*	Унарный плюс (нет операции); в отличие от сложения, он может итерационно применяться к множеству значений	`jsonb_path_query_array('{"x": [2,3,4]}', '+ $.x')` → `[2, 3, 4]`
число - *число* → *число*	Вычитание	`jsonb_path_query('[2]', '7 - $[0]')` → 5
- *число* → *число*	Смена знака; в отличие от вычитания, этот оператор может итерационно применяться к множеству значений	`jsonb_path_query_array('{"x": [2,3,4]}', '- $.x')` → `[-2, -3, -4]`
число * *число* → *число*	Умножение	`jsonb_path_query('[4]', '2 * $[0]')` → 8
число / *число* → *число*	Деление	`jsonb_path_query('[8.5]', '$[0] / 2')` → `4.2500000000000000`
число % *число* → *число*	Остаток от деления	`jsonb_path_query('[32]', '$[0] % 10')` → 2
значение . `type()` → *строка*	Тип элемента JSON (см. `json_typeof`)	`jsonb_path_query_array('[1, "2", {}]', '$[*].type()')` → `["number", "string", "object"]`
значение . `size()` → *число*	Размер элемента JSON (число элементов в массиве либо 1, если это не массив)	

Оператор/Метод	Описание	Пример(ы)
		`jsonb_path_query('{"m": [11, 15]}', '$.m.size()')` → `2`
значение `. boolean()` → *логическое_значение*	Логическое значение, преобразованное из логического значения JSON, числа или строки	`jsonb_path_query_array('[1, "yes", false]', '$[*].boolean()')` → `[true, true, false]`
значение `. string()` → *строка*	Строковое значение, преобразованное из логического значения JSON, числа, строки или значения типа дата-время	`jsonb_path_query_array('[1.23, "xyz", false]', '$[*].string()')` → `["1.23", "xyz", "false"]` `jsonb_path_query('"2023-08-15 12:34:56"', '$.timestamp().string()')` → `"2023-08-15T12:34:56"`
значение `. double()` → *число*	Приблизительное число с плавающей точкой, преобразованное из строки или числа JSON	`jsonb_path_query('{"len": "1.9"}', '$.len.double() * 2')` → `3.8`
число `. ceiling()` → *число*	Ближайшее целое, большее или равное заданному числу	`jsonb_path_query('{"h": 1.3}', '$.h.ceiling()')` → `2`
число `. floor()` → *число*	Ближайшее целое, меньшее или равное заданному числу	`jsonb_path_query('{"h": 1.7}', '$.h.floor()')` → `1`
число `. abs()` → *число*	Модуль заданного числа (абсолютное значение)	`jsonb_path_query('{"z": -0.3}', '$.z.abs()')` → `0.3`
значение `. bigint()` → *bigint*	Большое целочисленное значение, преобразованное из числового или строкового значения JSON	`jsonb_path_query('{"len": "9876543219"}', '$.len.bigint()')` → `9876543219`
значение `. decimal( [` *точность* `[ ,` *масштаб* `] ] )` → *decimal*	Округлённое десятичное значение, преобразованное из числового или строкового значения JSON (значения *точность* и *масштаб* должны быть целыми числами)	`jsonb_path_query('1234.5678', '$.decimal(6, 2)')` → `1234.57`
значение `. integer()` → *целое_число*	Целочисленное значение, преобразованное из числового или строкового значения JSON	`jsonb_path_query('{"len": "12345"}', '$.len.integer()')` → `12345`
значение `. number()` → *число*	Числовое значение, преобразованное из числового или строкового значения JSON	`jsonb_path_query('{"len": "123.45"}', '$.len.number()')` → `123.45`
строка `. datetime()` → *тип_даты_времени* (см. примечание)	Значение даты/времени, полученное из строки	`jsonb_path_query('["2015-8-1", "2015-08-12"]', '$[*] ? (@.datetime() < "2015-08-2".datetime())')` → `"2015-8-1"`

Оператор/Метод **Описание** **Пример(ы)**
string `. datetime(` *шаблон*`)` → *тип_даты_времени* (см. примечание) Значение даты/времени, преобразованное из строки по шаблону `to_timestamp` `jsonb_path_query_array('["12:30", "18:40"]', '$[*].datetime("HH24:MI")')` → `["12:30:00", "18:40:00"]`
строка `. date()` → *дата* Значение даты, преобразованное из строки `jsonb_path_query('"2023-08-15"', '$.date()')` → `"2023-08-15"`
строка `. time()` → *время без часового пояса* Значение времени без часового пояса, полученное из строки `jsonb_path_query('"12:34:56"', '$.time()')` → `"12:34:56"`
строка `. time(` *точность*`)` → *время без часового пояса* Значение времени без часового пояса, преобразованное из строки, с долями секунды, округлённой до заданной точности `jsonb_path_query('"12:34:56.789"', '$.time(2)')` → `"12:34:56.79"`
строка `. time_tz()` → *время с часовым поясом* Значение времени с часовым поясом, полученное из строки `jsonb_path_query('"12:34:56 +05:30"', '$.time_tz()')` → `"12:34:56+05:30"`
строка `. time_tz(` *точность*`)` → *время с часовым поясом* Значение времени с часовым поясом, преобразованное из строки, с долями секунды, округлённой до заданной точности `jsonb_path_query('"12:34:56.789 +05:30"', '$.time_tz(2)')` → `"12:34:56.79+05:30"`
строка `. timestamp()` → *дата и время без часового пояса* Значение даты/времени без часового пояса, полученное из строки `jsonb_path_query('"2023-08-15 12:34:56"', '$.timestamp()')` → `"2023-08-15T12:34:56"`
строка `. timestamp(` *точность*`)` → *дата и время без часового пояса* Значение даты-времени без часового пояса, преобразованное из строки, с долями секунды, округлённой до заданной точности `jsonb_path_query('"2023-08-15 12:34:56.789"', '$.timestamp(2)')` → `"2023-08-15T12:34:56.79"`
строка `. timestamp_tz()` → *дата и время с часовым поясом* Значение даты/времени с часовым поясом, полученное из строки `jsonb_path_query('"2023-08-15 12:34:56 +05:30"', '$.timestamp_tz()')` → `"2023-08-15T12:34:56+05:30"`
строка `. timestamp_tz(` *точность*`)` → *дата и время с часовым поясом* Значение даты/времени с часовым поясом, преобразованное из строки, с долями секунды, округлённой до заданной точности `jsonb_path_query('"2023-08-15 12:34:56.789 +05:30"', '$.timestamp_tz(2)')` → `"2023-08-15T12:34:56.79+05:30"`
object `. keyvalue()` → *массив* Пары ключ-значение, представленные в виде массива объектов со следующими тремя полями: `"key"`, `"value"` и `"id"`; в `"id"` содержится уникальный идентификатор объекта, к которому относится данная пара ключ-значение

Оператор/Метод **Описание** **Пример(ы)**
`jsonb_path_query_array('{"x": "20", "y": 32}', '$.keyvalue()')` → `[{"id": 0, "key": "x", "value": "20"}, {"id": 0, "key": "y", "value": 32}]`

Примечание

Результирующим типом методов `datetime()` и `datetime(`*шаблон*`)` может быть `date`, `timetz`, `time`, `timestamptz` или `timestamp`. Эти два метода определяют тип своего результата автоматически.

Метод `datetime()` пытается последовательно сопоставить поступившую на вход строку с ISO-форматами типов `date`, `timetz`, `time`, `timestamptz` и `timestamp`. Встретив первый подходящий формат, он останавливается и возвращает соответствующий тип данных.

Метод `datetime(`*шаблон*`)` определяет результирующий тип в соответствии с полями заданного шаблона.

Методы `datetime()` и `datetime(`*шаблон*`)` применяют те же правила разбора строки, что и SQL-функция `to_timestamp` (см. Раздел 9.8), но с тремя исключениями. Во-первых, эти методы не позволяют использовать в шаблоне поля, которым не находится соответствие. Во-вторых, в шаблоне допускаются только следующие разделители: знак минуса, точка, косая черта, запятая, апостроф, точка с запятой, запятая и пробел. В-третьих, разделители в шаблоне должны в точности соответствовать входной строке.

Если требуется сравнить значения разных типов даты/времени, применяется неявное приведение типа. Значение `date` может быть приведено к типу `timestamp` или `timestamptz`; `timestamp` — к типу `timestamptz`, а `time` — к `timetz`. Однако все эти приведения, кроме первого, зависят от текущего значения TimeZone и поэтому не могут выполняться в функциях `jsonpath`, не учитывающих часовой пояс. Аналогично, в других методах, связанных с датой/временем и преобразующих строки в типы даты/времени, также выполняется это преобразование, в котором может участвовать параметр TimeZone с текущим значением. Поэтому эти преобразования также могут выполняться только в функциях `jsonpath` с поддержкой часового пояса.

В Таблице 9.51 перечислены допустимые элементы выражения фильтра.

Таблица 9.51. Элементы выражения фильтра `jsonpath`

Предикат/значение **Описание** **Пример(ы)**

значение `==` *значение* → `boolean`
Проверка равенства (все операторы сравнения, включая этот, работают с любыми скалярными значениями JSON)
`jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == 1)')` → `[1, 1]`
`jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == "a")')` → `["a"]`

значение `!=` *значение* → `boolean`
значение `<>` *значение* → `boolean`
Проверка неравенства
`jsonb_path_query_array('[1, 2, 1, 3]', '$[*] ? (@ != 1)')` → `[2, 3]`
`jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <> "b")')` → `["a", "c"]`

значение `<` *значение* → `boolean`

Предикат/значение **Описание** **Пример(ы)**
Проверка «меньше» `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ < 2)')` → `[1]`
значение `<=` *значение* → `boolean` Проверка «меньше или равно» `jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <= "b")')` → `["a", "b"]`
значение `>` *значение* → `boolean` Проверка «больше» `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ > 2)')` → `[3]`
значение `>=` *значение* → `boolean` Проверка «больше или равно» `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ >= 2)')` → `[2, 3]`

`true` → `boolean`
JSON-константа `true`
`jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == true)')` → `{"name": "Chris", "parent": true}`

`false` → `boolean`
JSON-константа `false`
`jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == false)')` → `{"name": "John", "parent": false}`

`null` → *значение* JSON-константа `null` (заметьте, что в отличие от SQL сравнение с `null` работает традиционным образом) `jsonb_path_query('[{"name": "Mary", "job": null}, {"name": "Michael", "job": "driver"}]', '$[*] ? (@.job == null) .name')` → `"Mary"`
логическое_значение `&&` *логическое_значение* → `boolean` Логическое И `jsonb_path_query('[1, 3, 7]', '$[*] ? (@ > 1 && @ < 5)')` → `3`
логическое_значение `||` *логическое_значение* → `boolean` Логическое ИЛИ `jsonb_path_query('[1, 3, 7]', '$[*] ? (@ < 1 || @ > 5)')` → `7`

`!` *логическое_значение* → `boolean`
Логическое НЕ
`jsonb_path_query('[1, 3, 7]', '$[*] ? (!(@ < 5))')` → `7`

логическое_значение `is unknown` → `boolean` Проверяет, является ли `unknown` результатом логического условия. `jsonb_path_query('[-1, 2, 7, "foo"]', '$[*] ? ((@ > 0) is unknown)')` → `"foo"`
строка `like_regex` *строка* [`flag` *строка*] → `boolean` Проверяет, соответствует ли первый операнд регулярному выражению, которое задаёт второй операнд с необязательным аргументом `flag`, влияющим на поведение выражения (см. Подраздел 9.16.2.4).

Предикат/значение **Описание** **Пример(ы)**
`jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c")')` → `["abc", "abdacb"]` `jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c" flag "i")')` → `["abc", "aBdC", "abdacb"]`
строка `starts with` *строка* → `boolean` Проверяет, является ли второй операнд начальной подстрокой первого. `jsonb_path_query('["John Smith", "Mary Stone", "Bob Johnson"]', '$[*] ? (@ starts with "John")')` → `"John Smith"`

`exists (` *выражение_пути* `)` → `boolean`
Проверяет, соответствует ли выражению пути минимум один элемент SQL/JSON. Возвращает `unknown`, если вычисление выражения пути могло привести к ошибке; это используется во втором примере для недопущения ошибки «ключ не найден» в строгом режиме.
`jsonb_path_query('{"x": [1, 2], "y": [2, 4]}', 'strict $.* ? (exists (@ ? (@[*] > 2)))')` → `[2, 4]`
`jsonb_path_query_array('{"value": 41}', 'strict $ ? (exists (@.name)) .name')` → `[]`

9.16.2.4. Регулярные выражения SQL/JSON

Выражения путей SQL/JSON могут содержать фильтры `like_regex`, позволяющие сопоставлять текст с регулярным выражением. Например, следующий запрос пути SQL/JSON выберет все строки в массиве, которые начинаются с английской гласной в любом регистре:

```
$[*] ? (@ like_regex "^[aeiou]" flag "i")
```

Необязательная строка `flag` может содержать один или несколько следующих символов: `i`, делающий поиск регистронезависимым, `m`, допускающий сопоставление `^` и `$` с переводами строк, `s`, допускающий сопоставление `.` с символом новой строки, и `q`, берущий в кавычки весь шаблон (в результате производится простой поиск подстроки).

Стандарт SQL/JSON заимствует определение регулярных выражений от оператора `LIKE_REGEX`, который, в свою очередь, реализуется по стандарту XQuery. Однако в Postgres Pro оператор `LIKE_REGEX` в настоящее время отсутствует. Поэтому фильтр `like_regex` реализован с использованием механизма регулярных выражений POSIX, который описан в Подразделе 9.7.3. Вследствие этого наблюдается ряд небольших отклонений от описанного в стандарте поведения SQL/JSON, о которых рассказывается в Подразделе 9.7.3.8. Заметьте однако, что описанная там несовместимость букв флагов не проявляется на уровне SQL/JSON, так как заданные в SQL/JSON флаги XQuery переводятся во флаги, воспринимаемые механизмом POSIX.

Помните, что аргумент, задающий шаблон для `like_regex`, является строкой пути JSON и записывается по правилам, описанным в Подразделе 8.14.7. Это в частности означает, что каждую косую черту в регулярном выражении надо дублировать. Например, чтобы отобрать строковые значения, находящиеся на уровне корня документа и содержащие только цифры, нужно написать:

```
$.* ? (@ like_regex "^\\d+$")
```

9.16.3. Функции запросов SQL/JSON

Функции SQL/JSON `JSON_EXISTS()`, `JSON_QUERY()` и `JSON_VALUE()`, описанные в Таблица 9.52, можно использовать для запросов к документам JSON. Все эти функции используют *выражение_пути* (запрос пути SQL/JSON) к *элементу_контекста* (документу). За более подробным описанием возможного содержимого *выражения_пути* обратитесь к Подразделу 9.16.2. В *выражении_пути* также можно ссылаться на переменные, значения которых указаны с соответствующими именами в предложении `PASSING`, которое поддерживается всеми функциями. *элемент_контекста* может быть значением `jsonb` или символьной строкой, которая может быть приведена к `jsonb`.

Таблица 9.52. Функции запросов SQL/JSON

Сигнатура функции
Описание
Пример(ы)

```
JSON_EXISTS (
      элемент_контекста , выражение_пути
      [PASSING { значение AS имя_переменной } [, ...]]
      [{ TRUE | FALSE |UNKNOWN | ERROR } ON ERROR]) → boolean
```

- Возвращает true, если в результате применения *`выражения_пути`* SQL/JSON к *`элементу_контекста`* были получены какие-либо элементы, и false в противном случае.
- Предложение `ON ERROR` определяет поведение, если во время вычисления *`выражения_пути`* возникает ошибка. Указание `ERROR` приведёт к появлению ошибки с соответствующим сообщением. Другие параметры включают возврат значений типа `boolean` (`FALSE` или `TRUE`) или значения `UNKNOWN`, которое фактически равнозначно SQL NULL. Если предложение `ON ERROR` не указано, по умолчанию возвращается `FALSE`.

Примеры:

`JSON_EXISTS(jsonb '{"key1": [1,2,3]}', 'strict $.key1[*] ? (@ > $x)' PASSING 2 AS x)` → `t`

`JSON_EXISTS(jsonb '{"a": [1,2,3]}', 'lax $.a[5]' ERROR ON ERROR)` → `f`

`JSON_EXISTS(jsonb '{"a": [1,2,3]}', 'strict $.a[5]' ERROR ON ERROR)` →

`ОШИБКА: индекс массива jsonpath вне диапазона`

```
JSON_QUERY (
      элемент_контекста , выражение_пути
      [PASSING { значение AS имя_переменной } [, ...]]
      [ RETURNING тип_данных [ FORMAT JSON [ ENCODING UTF8 ] ] ]
      [{ WITHOUT | WITH { CONDITIONAL | [UNCONDITIONAL] } } [ARRAY] WRAPPER]
      [{ KEEP | OMIT } QUOTES [ON SCALAR STRING]]
      [{ ERROR | NULL | EMPTY { [ARRAY] | OBJECT } | DEFAULT выражение } ON EMPTY]
      [{ ERROR | NULL | EMPTY { [ARRAY] | OBJECT } | DEFAULT выражение } ON
       ERROR]) → jsonb
```

- Возвращает результат применения *`выражения_пути`* SQL/JSON к *`элементу_контекста`*.
- По умолчанию результат возвращается как значение типа `jsonb`, хотя можно использовать предложение `RETURNING` для возврата значения и другого типа, к которому можно привести тип `jsonb`.
- Если выражение пути может вернуть несколько значений, может потребоваться обернуть эти значения с помощью предложения `WITH WRAPPER`, чтобы превратить его в допустимую строку JSON, поскольку по умолчанию они не обёртываются, как если бы было указано `WITHOUT WRAPPER`. Предложение `WITH WRAPPER` по умолчанию означает `WITH UNCONDITIONAL WRAPPER`, то есть даже одно значение результата будет обёрнуто. Чтобы применить обёртку только при наличии нескольких значений, укажите `WITH CONDITIONAL WRAPPER`. Если указано предложение `WITHOUT WRAPPER`, получение нескольких значений в результате будет рассматриваться как ошибка.
- Если результатом является скалярная строка, по умолчанию возвращаемое значение заключается в кавычки, чтобы сделать его допустимым значением JSON. Это можно сделать явно, указав `KEEP QUOTES`. И наоборот, кавычки можно опустить, указав `OMIT QUOTES`. Чтобы гарантировать, что в результате будет получено допустимое значение JSON, `OMIT QUOTES` нельзя указывать одновременно с `WITH WRAPPER`.

Сигнатура функции
Описание
Пример(ы)

- Предложение `ON EMPTY` определяет поведение, если вычисление *выражения_пути* возвращает пустое множество. Предложение `ON ERROR` определяет поведение, если возникает ошибка при вычислении *выражения_пути*, при приведении значения результата к типу `RETURNING` или если вычисление *выражение_пути* возвращает пустое множество при вычислении выражения `ON EMPTY`.
- Как для `ON EMPTY`, так и для `ON ERROR` указание `ERROR` приведёт к выводу соответствующего сообщения об ошибке. При других вариантах возвращаются SQL NULL, пустой массив (`EMPTY [ARRAY]`), пустой объект (`EMPTY OBJECT`) или указанное пользователем выражение (`DEFAULT` *выражение*), которое может быть приведено к JSONB или типу, указанному в `RETURNING`. По умолчанию, если не указано `ON EMPTY` или `ON ERROR`, возвращается значение SQL NULL.

Примеры:

`JSON_QUERY(jsonb '[1,[2,3],null]', 'lax $[*][$off]' PASSING 1 AS off WITH CONDITIONAL WRAPPER)` → `3`

`JSON_QUERY(jsonb '{"a": "[1, 2]"}', 'lax $.a' OMIT QUOTES)` → `[1, 2]`

`JSON_QUERY(jsonb '{"a": "[1, 2]"}', 'lax $.a' RETURNING int[] OMIT QUOTES ERROR ON ERROR)` →

```
ОШИБКА:  ошибочный литерал массива: "[1, 2]"
ПОДРОБНОСТИ:  После размерностей массива отсутствует "]".
```

```
JSON_VALUE (
    элемент_контекста , выражение_пути
    [PASSING { значение AS имя_переменной } [, ...]]
    [ RETURNING тип_данных ]
    [{ ERROR | NULL | DEFAULT выражение } ON EMPTY]
    [{ ERROR | NULL | DEFAULT выражение } ON ERROR]) → text
```

- Возвращает результат применения *выражения_пути* SQL/JSON к *элементу_контекста*.
- Используйте `JSON_VALUE()` только в том случае, если ожидается, что полученное значение будет представлять собой один скалярный элемент SQL/JSON, получение нескольких значений будет считаться ошибкой. Если ожидается, что извлечённое значение может быть объектом или массивом, используйте функцию `JSON_QUERY`.
- По умолчанию результат, который должен быть единственным скалярным значением, возвращается как значение типа `text`, хотя можно использовать предложение `RETURNING` для возврата значения и другого типа, к которому можно привести тип `text`.
- Предложения `ON ERROR` и `ON EMPTY` имеют схожую семантику, указанную в описании `JSON_QUERY`, но разные наборы значений, возвращаемые вместо ошибки.
- Обратите внимание, что у скалярных строк, возвращаемых функцией `JSON_VALUE`, всегда удаляются кавычки, что эквивалентно указанию `OMIT QUOTES` в `JSON_QUERY`.

Примеры:

`JSON_VALUE(jsonb '"123.45"', '$' RETURNING float)` → `123.45`

`JSON_VALUE(jsonb '"03:04 2015-02-01"', '$.datetime("HH24:MI YYYY-MM-DD")' RETURNING date)` → `2015-02-01`

`JSON_VALUE(jsonb '[1,2]', 'strict $[$off]' PASSING 1 as off)` → `2`

`JSON_VALUE(jsonb '[1,2]', 'strict $[*]' DEFAULT 9 ON ERROR)` → `9`

> **Примечание**
>
> Выражение *`элемент_контекста`* неявно приводится к типу `jsonb`, если передаётся не этого типа. Однако обратите внимание, что любые ошибки разбора, возникающие во время этого преобразования, выдаются безусловно, то есть не обрабатываются в соответствии с (указанным или неявным) предложением `ON ERROR`.

> **Примечание**
>
> Функция `JSON_VALUE()` возвращает SQL NULL, если *`выражение_пути`* возвращает JSON `null`, в отличие от функции `JSON_QUERY()`, возвращающей JSON `null` как есть.

9.16.4. JSON_TABLE

`JSON_TABLE` — это функция SQL/JSON, которая обрабатывает данные JSON и выдаёт результаты в виде реляционного представления, к которому можно обращаться как к обычной таблице SQL. Использовать `JSON_TABLE` можно внутри предложения `FROM` оператора `SELECT`, `UPDATE` или `DELETE` и как источник данных в операторе `MERGE`.

Принимая данные JSON, функция `JSON_TABLE` обрабатывает выражение пути и извлекает часть представленных данных, которая будет использоваться в качестве *шаблона строк* для создаваемого представления. Каждое значение SQL/JSON, выдаваемое шаблоном строк, служит источником для отдельной строки в создаваемом представлении.

Чтобы разбить шаблон строк на столбцы, в функции `JSON_TABLE` применяется предложение `COLUMNS`, определяющее схему создаваемого представления. В этом предложении для каждого создаваемого столбца задаётся отдельное выражение пути, обрабатывающее шаблон строк, извлекающее элемент JSON и возвращающее его в виде отдельного значения SQL для данного столбца.

Данные JSON, хранящиеся на вложенном уровне шаблона строк, можно извлечь с помощью предложения `NESTED PATH`. Каждое предложение `NESTED PATH` можно использовать для генерации одного или нескольких столбцов с использованием данных из вложенного уровня шаблона строк. Эти столбцы можно указать в предложении `COLUMNS`, которое аналогично предложению COLUMNS верхнего уровня. Строки, сгенерированные из NESTED COLUMNS, называются *дочерними строками* и объединяются со строкой, созданной из столбцов, которые указаны в родительском предложении `COLUMNS`, чтобы получить строку в конечном представлении. Сами дочерние столбцы могут содержать спецификацию `NESTED PATH`, что позволяет извлекать данные, расположенные на произвольных уровнях вложенности. Столбцы, созданные несколькими `NESTED PATH` на одном уровне, считаются *соседними*, а их строки после соединения с родительской строкой объединяются с помощью UNION.

Строки, формируемые функцией `JSON_TABLE`, соединяются как последующие (`LATERAL`) со строкой, из которой они сформированы, поэтому нет необходимости явно соединять создаваемое представление с исходной таблицей, содержащей данные JSON. При этом, используя предложение `PLAN`, можно определить, как соединять столбцы, которые возвращает `NESTED PATH`.

Каждое предложение `NESTED PATH` может создать один или несколько столбцов. Столбцы, созданные `NESTED PATH` на одном уровне, считаются *соседними*; при этом столбцы, созданные вложенным выражением `NESTED PATH`, считаются потомками столбца, сформированного другим выражением `NESTED PATH` или выражением строки на более высоком уровне. При формировании результата сначала вместе составляются соседние столбцы, а после этого полученные строки соединяются с родительской строкой.

Синтаксис:

```
JSON_TABLE (
```

```
  элемент_контекста, выражение_пути [AS имя_пути_json] [PASSING { значение
 AS имя_переменной } [, ...]]
  COLUMNS ( столбец_таблицы_json [, ...] )
  [{ ERROR | EMPTY } ON ERROR]
  [PLAN ( план_таблицы_json ) | PLAN DEFAULT ( { INNER | OUTER } [, { CROSS | UNION }]
 | { CROSS | UNION } [, { INNER | OUTER }] )]
)

Здесь столбец_таблицы_json:
    имя тип [PATH описание_пути_json]
        [{ WITHOUT | WITH { CONDITIONAL | [UNCONDITIONAL] } } [ARRAY] WRAPPER]
        [{ KEEP | OMIT } QUOTES [ON SCALAR STRING]]
        [{ ERROR | NULL | EMPTY { [ARRAY] | OBJECT } | DEFAULT выражение } ON EMPTY]
        [{ ERROR | NULL | EMPTY { [ARRAY] | OBJECT } | DEFAULT выражение } ON ERROR]
  | имя тип EXISTS [PATH выражение_пути]
        [{ ERROR | TRUE | FALSE | UNKNOWN } ON ERROR]
  | NESTED PATH описание_пути_json [AS имя_пути]
        COLUMNS ( столбец_таблицы_json [, ...] )
  | имя FOR ORDINALITY
здесь план_таблицы_json:
    имя_пути_json [{ OUTER | INNER } первичный_план_таблицы_json]
  | первичный_план_таблицы_json { UNION первичный_план_таблицы_json } [...]
  | первичный_план_таблицы_json { CROSS первичный_план_таблицы_json } [...]
здесь первичный_план_таблицы_json:
    имя_пути_json | ( план_таблицы_json )
```

Каждый элемент синтаксиса описан ниже более подробно.

элемент_контекста, *выражение_пути* `[AS` *имя_пути_json*`] [PASSING {` *значение* `AS` *имя_переменной* `} [, ...]]`

Выражение *элемент_контекста* указывает входной документ для запроса, *выражение_пути* — это выражение пути SQL/JSON, определяющее запрос, а *имя_пути_json* — необязательное имя для *выражения_пути*. Необязательное предложение `PASSING` может предоставлять значения данных для переменных, указанных в *выражении_пути*. Результат обработки входных данных с использованием вышеупомянутых элементов называется *шаблоном строк*, который используется в качестве источника для значений строк в создаваемом представлении.

`COLUMNS` (*столбец_таблицы_json* [, ...])

Предложение `COLUMNS`, определяющее схему сконструированного представления. В этом предложении можно указать, что каждый столбец должен быть заполнен значением SQL/JSON, полученным путём применения выражения пути JSON к шаблону строк. Есть следующие варианты *столбца_таблицы_json*:

имя *тип* `[FORMAT JSON [ENCODING UTF8]] [ PATH` *выражение_пути* `]`

Вставляет значение SQL/JSON, полученное путём применения *выражения_пути* к шаблону строк, в выходную строку представления после приведения его к указанному *типу*.

`FORMAT JSON` явно указывает, что ожидается допустимый объект `json`. Указывать `FORMAT JSON` имеет смысл только в том случае, если *тип* — это `bpchar`, `bytea`, `character variationing`, `name`, `json`, `jsonb` или `text` или домен над этими типами.

Также можно указать предложения `WRAPPER` и `QUOTES` для форматирования вывода или `ON EMPTY` и `ON ERROR` для соответствующей обработки этих сценариев.

Примечание

Это предложение внутри преобразуется в `JSON_VALUE` или `JSON_QUERY` с соответствующей семантикой. Предложение преобразуется в `JSON_QUERY`, если указанный тип не является скалярным типом или присутствует одно из следующих предложений: `FORMAT JSON`, `WRAPPER` или `QUOTES`.

имя *тип* `EXISTS` [`PATH` *выражение_пути*]

Вставляет логическое значение, полученное путём применения *выражения_пути* к шаблону строк, в выходную строку представления после приведения её к указанному *типу*.

Значение показывает, будут ли получены какие-либо значения в результате применения выражения `PATH` к шаблону строк.

Указанный *тип* должен иметь приведение к типу `boolean`.

Также можно использовать предложение `ON ERROR`, чтобы выдавалась ошибка или возвращалось указанное значение при возникновении ошибки во время вычисления пути JSON или приведения значения SQL/JSON к указанному типу. По умолчанию в обоих случаях возвращается значение `NULL`.

Примечание

Это предложение внутри преобразуется в `JSON_EXISTS` с соответствующей семантикой.

`NESTED [ PATH ]` *выражение_пути* [`AS` *имя_пути_json*] `COLUMNS` (*столбец_таблицы_json* [, ...])

Извлекает значения SQL/JSON из вложенных уровней шаблона строк, создаёт один или несколько столбцов, как определено во вложенном предложении `COLUMNS`, и вставляет извлечённые значения SQL/JSON в эти столбцы. Выражение *столбец_таблицы_json* во вложенном предложении `COLUMNS` имеет тот же синтаксис, что и в родительском предложении `COLUMNS`.

Синтаксис `NESTED PATH` является рекурсивным, поэтому вкладывая одно предложение `NESTED PATH` в другое, можно опускаться ниже от уровня к уровню. Это позволяет развернуть иерархию объектов и массивов JSON в одном вызове функции, а не связывать несколько выражений `JSON_TABLE` в операторе SQL.

Используя предложение `PLAN`, можно определить, как объединять столбцы, возвращаемые предложениями `NESTED PATH`.

имя `FOR ORDINALITY`

Добавляет столбец, обеспечивающий последовательную нумерацию строк. В таблице может быть только один столбец нумерации. Нумерация строк начинается с единицы. Для дочерних строк, сформированных предложениями `NESTED PATH`, повторяется номер родительской строки.

Примечание

Во всех вариантах *столбца_таблицы_json*, описанных выше, если не указано предложение `PATH`, используется выражение пути `$.`*имя*, где *имя* — указанное имя столбца.

`AS` *имя_пути_json*

Необязательный параметр *имя_пути_json* служит идентификатором заданного параметра *выражение_пути*. Имя должно быть уникальным и отличаться от имён столбцов. Когда применяется предложение `PLAN`, необходимо задать имена для всех путей, включая шаблон строк. Имена путей в предложении `PLAN` не могут повторяться.

`PLAN` (*план_таблицы_json*)

Определяет, как соединять данные, возвращаемые предложениями `NESTED PATH`, с создаваемым представлением.

Соединять столбцы, реализуя отношения родитель/потомок, можно следующими способами:

`INNER`

Используйте `INNER JOIN`, чтобы родительская строка была исключена из вывода, если для неё не нашлось дочерних строк при объединении с данными, возвращёнными `NESTED PATH`.

`OUTER`

Используйте `LEFT OUTER JOIN`, чтобы родительская строка всегда включалась в вывод, даже если для неё не нашлось дочерних строк при объединении с данными, возвращёнными `NESTED PATH`. Если соответствующие значения отсутствуют, в дочерние столбцы будут вставлены значения NULL.

По умолчанию для соединения столбцов используется этот вариант.

Соединять соседние столбцы можно следующими способами:

`UNION`

Сгенерировать одну строку для каждого значения отдельного соседнего столбца. Соседи данного столбца при этом заполняются значениями NULL.

Этот вариант выбирается по умолчанию для соединения соседних столбцов.

`CROSS`

Сгенерировать одну строку для каждой комбинации значений из соседних столбцов.

`PLAN DEFAULT` (*`OUTER`* | *`INNER`* [, *`UNION`* | *`CROSS`*])

Эти указания могут также задаваться в обратном порядке. В этом случае `INNER` или `OUTER` определяет план соединения родительских/дочерних столбцов, а `UNION` или `CROSS` влияет на объединение соседних столбцов. Форма `PLAN DEFAULT` переопределяет план по умолчанию для всех столбцов сразу. Несмотря на то, что в форме `PLAN DEFAULT` имена путей не указываются, для соответствия стандарту SQL/JSON имена должны задаваться для всех путей, если используется предложение `PLAN`.

Использовать `PLAN DEFAULT` проще, чем указывать полный `PLAN`, и зачастую этого достаточно для получения желаемого результата.

{ `ERROR` | `EMPTY` } `ON ERROR`

Необязательное предложение `ON ERROR` можно использовать для указания способа обработки ошибок при вычислении *выражения_пути* верхнего уровня. Используйте предложение `ERROR`, чтобы выдавались ошибки, и `EMPTY`, чтобы возвращалась пустая таблица (то есть таблица, содержащая 0 строк). Обратите внимание, что это предложение не влияет на ошибки, возникающие при вычислении столбцов, поведение которых зависит от указания для них предложения `ON ERROR`.

Примеры

В приведённых примерах будет использоваться следующая таблица, содержащая данные JSON:

```
CREATE TABLE my_films ( js jsonb );

INSERT INTO my_films VALUES (
'{ "favorites" : [
   { "kind" : "comedy", "films" : [
     { "title" : "Bananas",
       "director" : "Woody Allen"},
```

```
      { "title" : "The Dinner Game",
        "director" : "Francis Veber" } ] },
    { "kind" : "horror", "films" : [
      { "title" : "Psycho",
        "director" : "Alfred Hitchcock" } ] },
    { "kind" : "thriller", "films" : [
      { "title" : "Vertigo",
        "director" : "Alfred Hitchcock" } ] },
    { "kind" : "drama", "films" : [
      { "title" : "Yojimbo",
        "director" : "Akira Kurosawa" } ] }
  ] }');
```

Составьте запрос, читающий из таблицы `my_films` данные о фильмах и создающий представление, в котором жанр, название и режиссёр фильма распределяются по отдельным столбцам:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE ( js, '$.favorites[*]' COLUMNS (
   id FOR ORDINALITY,
   kind text PATH '$.kind',
   NESTED PATH '$.films[*]' COLUMNS (
     title text PATH '$.title',
     director text PATH '$.director'))) AS jt;
----+----------+------------------+-------------------
 id |   kind   |       title      |    director
----+----------+------------------+-------------------
 1  | comedy   | Bananas          | Woody Allen
 1  | comedy   | The Dinner Game  | Francis Veber
 2  | horror   | Psycho           | Alfred Hitchcock
 3  | thriller | Vertigo          | Alfred Hitchcock
 4  | drama    | Yojimbo          | Akira Kurosawa
 (5 rows)
```

Ниже представлена изменённая версия запроса с использованием аргументов `PASSING` в фильтре, указанном в выражении пути JSON верхнего уровня, и различными параметрами для отдельных столбцов:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE (js, '$.favorites[*] ? (@.films[*].director == $filter)'
   PASSING 'Alfred Hitchcock' AS filter
     COLUMNS (
     id FOR ORDINALITY,
     kind text PATH '$.kind',
     title text FORMAT JSON PATH '$.films[*].title' OMIT QUOTES,
     director text PATH '$.films[*].director' KEEP QUOTES)) AS jt;
```

```
id | kind | title | director
----+----------+---------+--------------------
1 | horror | Psycho | "Alfred Hitchcock"
2 | thriller | Vertigo | "Alfred Hitchcock"
(2 строки)
```

Ниже представлена изменённая версия запроса с использованием `NESTED PATH` для заполнения столбцов title и director, демонстрирующая их соединение с родительскими столбцами id и kind:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE ( js, '$.favorites[*] ? (@.films[*].director == $filter)'
   PASSING 'Alfred Hitchcock' AS filter
```

```
    COLUMNS (
     id FOR ORDINALITY,
     kind text PATH '$.kind',
     NESTED PATH '$.films[*]' COLUMNS (
       title text FORMAT JSON PATH '$.title' OMIT QUOTES,
       director text PATH '$.director' KEEP QUOTES))) AS jt;
```

```
 id |   kind   |  title  |      director
----+----------+---------+--------------------
  1 | horror   | Psycho  | "Alfred Hitchcock"
  2 | thriller | Vertigo | "Alfred Hitchcock"
(2 rows)
```

Ниже представлен тот же запрос, но без фильтра в корневом пути:

```
SELECT jt.* FROM
 my_films,
 JSON_TABLE ( js, '$.favorites[*]'
   COLUMNS (
    id FOR ORDINALITY,
    kind text PATH '$.kind',
    NESTED PATH '$.films[*]' COLUMNS (
      title text FORMAT JSON PATH '$.title' OMIT QUOTES,
      director text PATH '$.director' KEEP QUOTES))) AS jt;
```

```
 id |   kind   |      title      |      director
----+----------+-----------------+--------------------
  1 | comedy   | Bananas         | "Woody Allen"
  1 | comedy   | The Dinner Game | "Francis Veber"
  2 | horror   | Psycho          | "Alfred Hitchcock"
  3 | thriller | Vertigo         | "Alfred Hitchcock"
  4 | drama    | Yojimbo         | "Akira Kurosawa"
(5 rows)
```

Ниже представлен ещё один запрос, использующий в качестве входных данных другой объект `JSON`. В нём показано объединение «соседей» между путями `NESTED $.movies[*]` и `$.books[*]`, а также использование столбца `FOR ORDINALITY` на уровнях `NESTED` (столбцы `movie_id`, `book_id` и `author_id`):

```
SELECT * FROM JSON_TABLE (
'{"favorites":
    [{"movies":
      [{"name": "One", "director": "John Doe"},
       {"name": "Two", "director": "Don Joe"}],
     "books":
      [{"name": "Mystery", "authors": [{"name": "Brown Dan"}]},
       {"name": "Wonder", "authors": [{"name": "Jun Murakami"}, {"name":"Craig Doe"}]}]
}]}'::json, '$.favorites[*]'
COLUMNS (
  user_id FOR ORDINALITY,
  NESTED '$.movies[*]'
    COLUMNS (
    movie_id FOR ORDINALITY,
    mname text PATH '$.name',
    director text),
  NESTED '$.books[*]'
    COLUMNS (
      book_id FOR ORDINALITY,
      bname text PATH '$.name',
      NESTED '$.authors[*]'
        COLUMNS (
```

```
          author_id FOR ORDINALITY,
          author_name text PATH '$.name'))));

 user_id | movie_id | mname | director | book_id |  bname  | author_id | author_name
---------+----------+-------+----------+---------+---------+-----------+--------------
       1 |        1 | One   | John Doe |         |         |           |
       1 |        2 | Two   | Don Joe  |         |         |           |
       1 |          |       |          |       1 | Mystery |         1 | Brown Dan
       1 |          |       |          |       2 | Wonder  |         1 | Jun Murakami
       1 |          |       |          |       2 | Wonder  |         2 | Craig Doe
(5 rows)
```

9.17. Функции для работы с последовательностями

В этом разделе описаны функции для работы с объектами, представляющими *последовательности*. Такие объекты (также называемыми генераторами последовательностей или просто последовательностями) являются специальными таблицами из одной строки и создаются командой CREATE SEQUENCE. Используются они обычно для получения уникальных идентификаторов строк таблицы. Функции, перечисленные в Таблице 9.53, предоставляют простые и безопасные для параллельного использования методы получения очередных значений таких последовательностей.

Таблица 9.53. Функции для работы с последовательностями

Функция	**Описание**
`nextval ( regclass ) → bigint`	Продвигает объект последовательности к следующему значению и возвращает это значение. Это действие атомарно: даже если вызвать `nextval` параллельно в нескольких сеансах, в каждом вызове будет гарантированно получено неповторяющееся значение последовательности. Если объект последовательности создаётся с параметрами по умолчанию, при каждом очередном вызове `nextval` будет выдаваться следующее по порядку значение, начиная с 1. Изменить это поведение можно, установив соответствующие параметры команды CREATE SEQUENCE. Этой функции требуется право `USAGE` или `UPDATE` для последовательности.
`setval ( regclass, bigint [, boolean] ) → bigint`	Устанавливает для объекта последовательности текущее значение и может также установить флаг `is_called` . В форме с двумя параметрами устанавливает для последовательности заданное значение поля `last_value` и значение `true` для флага `is_called` , показывающего, что при следующем вызове `nextval` последовательность должна сначала продвинуться к очередному значению, которое будет возвращено. При этом `currval` также возвратит заданное значение. В форме с тремя параметрами флагу `is_called` тоже можно присвоить `true` или `false`. Со значением `true` она действует так же, как и форма с двумя параметрами. Если же присвоить этому флагу значение `false`, первый вызов `nextval` после этого вернёт именно заданное значение, а продвижение последовательности произойдёт при последующем вызове `nextval`. Кроме того, значение, возвращаемое `currval` в этом случае, не меняется. Например: `SELECT setval('myseq', 42);` *При следующем вызове nextval выдаст 43* `SELECT setval('myseq', 42, true);` *То же самое* `SELECT setval('myseq', 42, false);` *При следующем вызове nextval выдаст 42* Результатом `setval` будет просто значение её второго аргумента. Этой функции требуется право `UPDATE` для последовательности.
`currval ( regclass ) → bigint`	Возвращает значение, выданное при последнем вызове `nextval` для этой последовательности в текущем сеансе. (Если в данном сеансе `nextval` ни разу не вызывалась для данной последовательности, возвращается ошибка.) Так как это значение ограничено рам-

Функция Описание
ками сеанса, эта функция выдаёт предсказуемый результат вне зависимости от того, вызвалась ли впоследствии `nextval` в других сеансах или нет. Этой функции требуется право `USAGE` или `SELECT` для последовательности.
`lastval` () → `bigint` Возвращает значение, выданное при последнем вызове `nextval` в текущем сеансе. Эта функция подобна `currval`, но она не принимает в параметрах имя последовательности, а обращается к той последовательности, для которой вызывалась `nextval` в последний раз в текущем сеансе. Если в текущем сеансе функция `nextval` ещё не вызывалась, при вызове `lastval` произойдёт ошибка. Этой функции требуется право `USAGE` или `SELECT` для последней использованной последовательности.

Внимание

Во избежание блокирования параллельных транзакций, пытающихся получить значения одной последовательности, значение, выданное функцией `nextval`, не высвобождается для повторного использования, если вызывающая транзакция впоследствии прерывается. Это означает, что в случае сбоев базы данных или прерывания транзакций в последовательности задействованных значений могут появляться пропуски. Пропуски могут появиться и без прерывания транзакции. Например, команда `INSERT` с предложением `ON CONFLICT` вычислит кортеж, претендующий на добавление, произведя все требуемые вызовы `nextval`, прежде чем выявит конфликты, которые могут привести к отработке правил `ON CONFLICT` вместо добавления. Таким образом, объекты последовательностей Postgres Pro *не годятся для получения непрерывных последовательностей*.

В том же ключе изменения состояния последовательности, произведённые функцией `setval`, сразу же становятся видимыми для других транзакций и не отменяются при откате транзакции, вызвавшей эту функцию.

Если кластер базы прерывает работу до фиксации транзакции, содержащей вызов функции `nextval` или `setval`, изменение состояния последовательности может не попасть в постоянное хранилище, поэтому неизвестно, какое состояние будет иметь последовательность после перезапуска кластера: исходное или изменённое. Это не создаёт проблем при использовании последовательности внутри базы данных, так как другие изменения незафиксированных транзакций также не будут видны. Однако если значение последовательности будет сохраняться и использоваться вне базы данных, следует предварительно убедиться в том, что вызов функции `nextval` успешно зафиксирован.

Последовательность, с которой будет работать любая из вышеописанных функций, определяется аргументом типа `regclass`, который представляет собой просто OID последовательности в системном каталоге `pg_class`. Однако вам не нужно вручную вычислять OID, так как процедура ввода значения `regclass` может сделать это за вас. За подробностями обратитесь к Разделу 8.19.

9.18. Условные выражения

В этом разделе описаны SQL-совместимые условные выражения, которые поддерживаются в Postgres Pro.

Подсказка

Если возможностей этих условных выражений оказывается недостаточно, вероятно, имеет смысл перейти к написанию серверных функций на более мощном языке программирования.

Примечание

Хотя конструкции `COALESCE`, `GREATEST` и `LEAST` синтаксически похожи на функции, они не являются обычными функциями, и поэтому им нельзя передать в аргументах явно описанный массив `VARIADIC`.

9.18.1. CASE

Выражение `CASE` в SQL представляет собой общее условное выражение, напоминающее операторы if/else в других языках программирования:

```
CASE WHEN условие THEN результат
     [WHEN ...]
     [ELSE результат]
END
```

Предложения `CASE` можно использовать везде, где допускаются выражения. Каждое *`условие`* в нём представляет собой выражение, возвращающее результат типа `boolean`. Если результатом выражения оказывается true, значением выражения `CASE` становится *`результат`*, следующий за условием, а остальная часть выражения `CASE` не вычисляется. Если же условие не выполняется, за ним таким же образом проверяются все последующие предложения `WHEN`. Если не выполняется ни одно из *`условий`* `WHEN`, значением `CASE` становится *`результат`*, записанный в предложении `ELSE`. Если при этом предложение `ELSE` отсутствует, результатом выражения будет NULL.

Пример:

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

Типы данных всех выражений *`результатов`* должны приводиться к одному выходному типу. Подробнее это описано в Разделе 10.5.

Существует также «простая» форма выражения `CASE`, разновидность вышеприведённой общей формы:

```
CASE выражение
    WHEN значение THEN результат
    [WHEN ...]
    [ELSE результат]
END
```

В такой форме сначала вычисляется первое *`выражение`*, а затем его результат сравнивается с выражениями *`значений`* в предложениях `WHEN`, пока не будет найдено равное ему. Если такого значения не находится, возвращается *`результат`* предложения `ELSE` (или NULL). Эта форма больше похожа на оператор `switch`, существующий в языке C.

Показанный ранее пример можно записать по-другому, используя простую форму `CASE`:

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

В выражении `CASE` вычисляются только те подвыражения, которые необходимы для получения результата. Например, так можно избежать ошибки деления на ноль:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

Примечание

Как было описано в Подразделе 4.2.14, всё же возможны ситуации, когда подвыражения вычисляются на разных этапах, так что железной гарантии, что в «`CASE` вычисляются только необходимые подвыражения», в принципе нет. Например, константное подвыражение `1/0` обычно вызывает ошибку деления на ноль на этапе планирования, хотя эта ветвь `CASE` может вовсе не вычисляться во время выполнения.

9.18.2. `COALESCE`

`COALESCE(`*`значение`*` [, ...])`

Функция `COALESCE` возвращает первый попавшийся аргумент, отличный от NULL. Если же все аргументы равны NULL, результатом тоже будет NULL. Это часто используется при отображении данных для подстановки некоторого значения по умолчанию вместо значений NULL:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Этот запрос вернёт значение `description`, если оно не равно NULL, либо `short_description`, если оно не NULL, и строку `(none)`, если оба эти значения равны NULL.

Аргументы должны быть приводимыми к одному общему типу, который и будет типом результата (подробнее об этом говорится в Разделе 10.5).

Как и выражение `CASE`, `COALESCE` вычисляет только те аргументы, которые необходимы для получения результата; то есть, аргументы правее первого отличного от NULL аргумента не вычисляются. Эта функция соответствует стандарту SQL, а в некоторых других СУБД её аналоги называются `NVL` и `IFNULL`.

9.18.3. `NULLIF`

`NULLIF(`*`значение1`*`, `*`значение2`*`)`

Функция `NULLIF` выдаёт значение NULL, если *`значение1`* равно *`значение2`*; в противном случае она возвращает *`значение1`*. Это может быть полезно для реализации обратной операции к `COALESCE`. В частности, для примера, показанного выше:

```
SELECT NULLIF(value, '(none)') ...
```

В данном примере если `value` равно `(none)`, выдаётся null, а иначе возвращается значение `value`.

Два её аргумента должны быть сравнимых типов. Если говорить точнее, они сравниваются точно так же, как сравнивались бы в записи *`значение1 = значение2`*, так что для этих типов должен существовать подходящий оператор `=`.

Результат будет иметь тот же тип, что и первый аргумент, но есть одна тонкость. Эта функция фактически возвращает первый аргумент подразумеваемого оператора `=`, который в некоторых случаях преобразуется к типу второго аргумента. Например, `NULLIF(1, 2.2)` возвращает `numeric`, так как оператор `integer = numeric` не существует, существует только оператор `numeric = numeric`.

9.18.4. `GREATEST` и `LEAST`

`GREATEST(`*`значение`*` [, ...])`

`LEAST(`*`значение`*` [, ...])`

Функции `GREATEST` и `LEAST` выбирают наибольшее или наименьшее значение из списка выражений. Все эти выражения должны приводиться к общему типу данных, который станет типом результата (подробнее об этом в Разделе 10.5).

Значения NULL в этом списке игнорируются, так что результат выражения будет равен NULL, только если все его аргументы равны NULL. (Это отклонение от стандарта SQL. Согласно стандарту, возвращается значение NULL, если какой-либо аргумент равен NULL. Подобным образом ведут себя и некоторые другие базы данных.)

9.19. Функции и операторы для работы с массивами

В Таблице 9.54 показаны имеющиеся специальные операторы для типов-массивов. Кроме них для массивов определены обычные операторы сравнения, показанные в Таблице 9.1. Эти операторы сравнения сопоставляют содержимое массивов по элементам, используя при этом функцию сравнения для B-дерева, определённую для типа данного элемента по умолчанию, и упорядочивают их по первому различию. В многомерных массивах элементы просматриваются по строкам (индекс последней размерности меняется в первую очередь). Если содержимое двух массивов совпадает, а размерности отличаются, результат их сравнения будет определяться первым отличием в размерностях.

Таблица 9.54. Операторы для работы с массивами

Оператор **Описание** **Пример(ы)**

`anyarray @> anyarray → boolean`
Первый массив содержит второй (имеется ли для каждого элемента второго массива равный ему в первом)? (Повторяющиеся элементы рассматриваются на общих основаниях, поэтому массивы `ARRAY[1]` и `ARRAY[1,1]` считаются содержащими друг друга.)
`ARRAY[1,4,3] @> ARRAY[3,1,3]` → `t`

`anyarray <@ anyarray → boolean`
Первый массив содержится во втором?
`ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6]` → `t`

`anyarray && anyarray → boolean`
Массивы пересекаются (у них есть общие элементы)?
`ARRAY[1,4,3] && ARRAY[2,1]` → `t`

`anycompatiblearray || anycompatiblearray → anycompatiblearray` Соединяет два массива. Если один из операндов — NULL или пустой массив, оператор никак не действует; в противном случае число размерностей массивов должно быть одинаковым (этот случай показан в первом примере) или могут отличаться на один (это ил-

Оператор **Описание** **Пример(ы)**

люстрирует второй пример). Если это массивы не одинаковых типов элементов, они будут приведены к общему типу (см. Раздел 10.5).
`ARRAY[1,2,3] || ARRAY[4,5,6,7]` → `{1,2,3,4,5,6,7}`
`ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9.9]]` → `{{1,2,3},{4,5,6},{7,8,9.9}}`

`anycompatible || anycompatiblearray` → `anycompatiblearray`
Вставляет элемент в начало массива (массив должен быть пустым или одномерным).
`3 || ARRAY[4,5,6]` → `{3,4,5,6}`

`anycompatiblearray || anycompatible` → `anycompatiblearray`
Вставляет элемент в конец массива (массив должен быть пустым или одномерным).
`ARRAY[4,5,6] || 7` → `{4,5,6,7}`

Подробнее поведение операторов с массивами описано в Разделе 8.15. За дополнительными сведениями об операторах, поддерживающих индексы, обратитесь к Разделу 11.2.

В Таблице 9.55 перечислены функции, предназначенные для работы с массивами. Дополнительная информация о них и примеры использования приведены в Разделе 8.15.

Таблица 9.55. Функции для работы с массивами

Функция **Описание** **Пример(ы)**

`array_append` (`anycompatiblearray`, `anycompatible`) → `anycompatiblearray`
Добавляет элемент в конец массива (так же, как оператор `anycompatiblearray || anycompatible`).
`array_append(ARRAY[1,2], 3)` → `{1,2,3}`

`array_cat` (`anycompatiblearray`, `anycompatiblearray`) → `anycompatiblearray`
Соединяет два массива (так же, как оператор `anycompatiblearray || anycompatiblearray`).
`array_cat(ARRAY[1,2,3], ARRAY[4,5])` → `{1,2,3,4,5}`

`array_dims` (`anyarray`) → `text`
Возвращает текстовое представление размерностей массива.
`array_dims(ARRAY[[1,2,3], [4,5,6]])` → `[1:2][1:3]`

`array_fill` (`anyelement`, `integer[]` [, `integer[]`]) → `anyarray`
Возвращает массив, заполненный заданным значением и имеющий размерности, указанные во втором аргументе. В необязательном третьем аргументе могут быть заданы нижние границы для каждой размерности (по умолчанию `1`).
`array_fill(11, ARRAY[2,3])` → `{{11,11,11},{11,11,11}}`
`array_fill(7, ARRAY[3], ARRAY[2])` → `[2:4]={7,7,7}`

`array_length` (`anyarray`, `integer`) → `integer`
Возвращает длину указанной размерности массива. (Для пустых или несуществующих размерностей массива выдаёт не 0, а NULL.)
`array_length(array[1,2,3], 1)` → `3`
`array_length(array[]::int[], 1)` → `NULL`
`array_length(array['text'], 2)` → `NULL`

`array_lower` (`anyarray`, `integer`) → `integer` Возвращает нижнюю границу указанной размерности массива.

Функция **Описание** **Пример(ы)**
`array_lower('[0:2]={1,2,3}'::integer[], 1)` → `0`

`array_ndims` (`anyarray`) → `integer`
Возвращает число размерностей массива.
`array_ndims(ARRAY[[1,2,3], [4,5,6]])` → `2`

`array_position` (`anycompatiblearray`, `anycompatible` [, `integer`]) → `integer`
Возвращает позицию первого вхождения второго аргумента в массиве либо `NULL` в случае отсутствия соответствующего элемента. Если задан третий аргумент, поиск начинается с заданной позиции. Массив должен быть одномерным. Эта функция определяет равенство как `IS NOT DISTINCT FROM`, что позволяет искать и значения `NULL`.
`array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon')` → `2`

`array_positions` (`anycompatiblearray`, `anycompatible`) → `integer[]`
Возвращает массив позиций всех вхождений второго аргумента в одномерном массиве, заданном первым аргументом. Эта функция определяет равенство как `IS NOT DISTINCT FROM`, что позволяет искать и значения `NULL`. Результат `NULL` возвращается, только если в качестве массива передаётся `NULL`; в случае же отсутствия искомого значения в заданном массиве возвращается пустой массив.
`array_positions(ARRAY['A','A','B','A'], 'A')` → `{1,2,4}`

`array_prepend` (`anycompatible`, `anycompatiblearray`) → `anycompatiblearray`
Вставляет элемент в начало массива (так же, как оператор `anycompatibleelement || anycompatiblearray`).
`array_prepend(1, ARRAY[2,3])` → `{1,2,3}`

`array_remove` (`anycompatiblearray`, `anycompatible`) → `anycompatiblearray`
Удаляет из массива все элементы, равные заданному значению. Массив должен быть одномерным. Эта функция определяет равенство как `IS NOT DISTINCT FROM`, что позволяет удалять и элементы `NULL`.
`array_remove(ARRAY[1,2,3,2], 2)` → `{1,3}`

`array_replace` (`anycompatiblearray`, `anycompatible`, `anycompatible`) → `anycompatiblearray`
Заменяет каждый элемент массива, равный второму аргументу, значением третьего аргумента.
`array_replace(ARRAY[1,2,5,4], 5, 3)` → `{1,2,3,4}`

`array_sample` (*`array`* `anyarray`, *`n`* `integer`) → `anyarray`
Возвращает массив из *`n`* элементов, случайно выбранных из *`array`*. Значение *`n`* не может превышать длину первого измерения *`array`*. Если в *`array`* передаётся многомерный массив, берутся «элементы» первой размерности.
`array_sample(ARRAY[1,2,3,4,5,6], 3)` → `{2,6,1}`
`array_sample(ARRAY[[1,2],[3,4],[5,6]], 2)` → `{{5,6},{1,2}}`

`array_shuffle` (`anyarray`) → `anyarray`
Случайным образом перемешивает элементы первой размерности массива.
`array_shuffle(ARRAY[[1,2],[3,4],[5,6]])` → `{{5,6},{1,2},{3,4}}`

`array_to_string` (*`array`* `anyarray`, *`delimiter`* `text` [, *`null_string`* `text`]) → `text` Представляет все элементы массива в виде текстовых строк и объединяет эти строки через разделитель, заданный параметром *`delimiter`*. Если в параметре *`null_string`* передана отличная от `NULL` строка, эта строка будет представлять содержащиеся в массиве элементы `NULL`, в противном случае такие элементы опускаются. См. также `string_to_array` .

<table>
<tr><th>Функция
Описание
Пример(ы)</th></tr>
<tr><td><code>array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')</code> → <code>1,2,3,*,5</code></td></tr>
<tr><td><code>array_upper</code> (<code>anyarray</code>, <code>integer</code>) → <code>integer</code>
Возвращает верхнюю границу указанной размерности массива.
<code>array_upper(ARRAY[1,8,3,7], 1)</code> → <code>4</code></td></tr>
<tr><td><code>cardinality</code> (<code>anyarray</code>) → <code>integer</code>
Возвращает общее число элементов в массиве (0, если массив пуст).
<code>cardinality(ARRAY[[1,2],[3,4]])</code> → <code>4</code></td></tr>
<tr><td><code>trim_array</code> (<i>array</i> <code>anyarray</code>, <i>n</i> <code>integer</code>) → <code>anyarray</code>
Сокращает массив, удаляя последние <i>n</i> элементов. Если массив многомерный, сокращается только первая размерность.
<code>trim_array(ARRAY[1,2,3,4,5,6], 2)</code> → <code>{1,2,3,4}</code></td></tr>
<tr><td><code>unnest</code> (<code>anyarray</code>) → <code>setof anyelement</code>
Разворачивает массив в набор строк. Элементы массива прочитываются в порядке хранения.
<code>unnest(ARRAY[1,2])</code> →

<code>1</code>
<code>2</code>

<code>unnest(ARRAY[['foo','bar'],['baz','quux']])</code> →

<code>foo</code>
<code>bar</code>
<code>baz</code>
<code>quux</code></td></tr>
<tr><td><code>unnest</code> (<code>anyarray</code>, <code>anyarray</code> [, ...]) → <code>setof anyelement, anyelement [, ...]</code>
Разворачивает массивы (возможно разных типов) в набор кортежей. Если массивы имеют разную длину, кортежи дополняются до большей длины значениями <code>NULL</code>. Эта форма допускается только в предложении FROM; см. Подраздел 7.2.1.4.
<code>select * from unnest(ARRAY[1,2], ARRAY['foo','bar','baz']) as x(a,b)</code> →

<code> a | b</code>
<code>---+-----</code>
<code> 1 | foo</code>
<code> 2 | bar</code>
<code> | baz</code></td></tr>
</table>

Вы также можете узнать об агрегатной функции, работающей с массивами, `array_agg` в Разделе 9.21.

9.20. Диапазонные/мультидиапазонные функции и операторы

Диапазонные типы данных рассматриваются в Разделе 8.17.

В Таблице 9.56 показаны имеющиеся специальные операторы для диапазонных типов, а в Таблице 9.57 — для мультидиапазонных типов. Кроме них для диапазонов и мультидиапазонов определены обычные операторы сравнения, показанные в Таблице 9.1. Операторы сравнения сначала сравнивают нижние границы диапазонов, и только если они равны, сравнивают верхние границы. Операторы сравнения мультидиапазонов сравнивают каждый диапазон, пока не найдут неравенство. Такие операторы сравнения обычно не помогают получить полезный в целом вариант сортировки, но позволяют строить по диапазонам уникальные индексы.

Таблица 9.56. Диапазонные операторы

Оператор	Описание	Пример(ы)
`anyrange @> anyrange` → `boolean`	Первый диапазон содержит второй?	`int4range(2,4) @> int4range(2,3)` → `t`
`anyrange @> anyelement` → `boolean`	Диапазон содержит заданный элемент?	`'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp` → `t`
`anyrange <@ anyrange` → `boolean`	Первый диапазон содержится во втором?	`int4range(2,4) <@ int4range(1,7)` → `t`
`anyelement <@ anyrange` → `boolean`	Заданный элемент содержится в диапазоне?	`42 <@ int4range(1,7)` → `f`
`anyrange && anyrange` → `boolean`	Диапазоны пересекаются (у них есть общие элементы)?	`int8range(3,7) && int8range(4,12)` → `t`
`anyrange << anyrange` → `boolean`	Первый диапазон располагается строго слева от второго?	`int8range(1,10) << int8range(100,110)` → `t`
`anyrange >> anyrange` → `boolean`	Первый диапазон располагается строго справа от второго?	`int8range(50,60) >> int8range(20,30)` → `t`
`anyrange &< anyrange` → `boolean`	Первый диапазон не простирается правее второго?	`int8range(1,20) &< int8range(18,20)` → `t`
`anyrange &> anyrange` → `boolean`	Первый диапазон не простирается левее второго?	`int8range(7,20) &> int8range(5,10)` → `t`
`anyrange -|- anyrange` → `boolean`	Диапазоны примыкают друг к другу?	`numrange(1.1,2.2) -|- numrange(2.2,3.3)` → `t`
`anyrange + anyrange` → `anyrange`	Вычисляет объединение диапазонов. Диапазоны должны пересекаться или касаться друг друга, чтобы их объединением был один диапазон (но см. `range_merge()`).	`numrange(5,15) + numrange(10,20)` → `[5,20)`
`anyrange * anyrange` → `anyrange`	Вычисляет пересечение диапазонов.	`int8range(5,15) * int8range(10,20)` → `[10,15)`
`anyrange - anyrange` → `anyrange`	Вычисляет разность диапазонов. При этом второй диапазон, если он содержится в первом, должен располагаться так, чтобы в результате его вычитания получался один диапазон.	`int8range(5,15) - int8range(10,20)` → `[5,10)`

Таблица 9.57. Мультидиапазонные операторы

Оператор **Описание** **Пример(ы)**

`anymultirange @> anymultirange → boolean`
Первый мультидиапазон содержит второй?
`'{[2,4)}'::int4multirange @> '{[2,3)}'::int4multirange → t`

`anymultirange @> anyrange → boolean`
Мультидиапазон содержит заданный диапазон?
`'{[2,4)}'::int4multirange @> int4range(2,3) → t`

`anymultirange @> anyelement → boolean`
Мультидиапазон содержит заданный элемент?
`'{[2011-01-01,2011-03-01)}'::tsmultirange @> '2011-01-10'::timestamp → t`

`anyrange @> anymultirange → boolean`
Диапазон содержит заданный мультидиапазон?
`'[2,4)'::int4range @> '{[2,3)}'::int4multirange → t`

`anymultirange <@ anymultirange → boolean`
Первый мультидиапазон содержится во втором?
`'{[2,4)}'::int4multirange <@ '{[1,7)}'::int4multirange → t`

`anymultirange <@ anyrange → boolean`
Мультидиапазон содержится в диапазоне?
`'{[2,4)}'::int4multirange <@ int4range(1,7) → t`

`anyrange <@ anymultirange → boolean`
Диапазон содержится в мультидиапазоне?
`int4range(2,4) <@ '{[1,7)}'::int4multirange → t`

`anyelement <@ anymultirange → boolean`
Заданный элемент содержится в мультидиапазоне?
`4 <@ '{[1,7)}'::int4multirange → t`

`anymultirange && anymultirange → boolean`
Мультидиапазоны пересекаются (у них есть общие элементы)?
`'{[3,7)}'::int8multirange && '{[4,12)}'::int8multirange → t`

`anymultirange && anyrange → boolean`
Мультидиапазон пересекает заданный диапазон?
`'{[3,7)}'::int8multirange && int8range(4,12) → t`

`anyrange && anymultirange → boolean`
Диапазон пересекает заданный мультидиапазон?
`int8range(3,7) && '{[4,12)}'::int8multirange → t`

`anymultirange << anymultirange → boolean`
Первый мультидиапазон располагается строго слева от второго?
`'{[1,10)}'::int8multirange << '{[100,110)}'::int8multirange → t`

`anymultirange << anyrange → boolean`
Мультидиапазон располагается строго слева от заданного диапазона?
`'{[1,10)}'::int8multirange << int8range(100,110) → t`

`anyrange << anymultirange → boolean` Диапазон располагается строго слева от заданного мультидиапазона?

Оператор **Описание** **Пример(ы)**
`int8range(1,10) << '{[100,110)}'::int8multirange` → `t`

`anymultirange >> anymultirange` → `boolean`
Первый мультидиапазон располагается строго справа от второго?
`'{[50,60)}'::int8multirange >> '{[20,30)}'::int8multirange` → `t`

`anymultirange >> anyrange` → `boolean`
Мультидиапазон располагается строго справа от заданного диапазона?
`'{[50,60)}'::int8multirange >> int8range(20,30)` → `t`

`anyrange >> anymultirange` → `boolean`
Диапазон располагается строго справа от заданного мультидиапазона?
`int8range(50,60) >> '{[20,30)}'::int8multirange` → `t`

`anymultirange &< anymultirange` → `boolean`
Первый мультидиапазон не простирается правее второго?
`'{[1,20)}'::int8multirange &< '{[18,20)}'::int8multirange` → `t`

`anymultirange &< anyrange` → `boolean`
Мультидиапазон не простирается правее заданного диапазона?
`'{[1,20)}'::int8multirange &< int8range(18,20)` → `t`

`anyrange &< anymultirange` → `boolean`
Диапазон не простирается правее заданного мультидиапазона?
`int8range(1,20) &< '{[18,20)}'::int8multirange` → `t`

`anymultirange &> anymultirange` → `boolean`
Первый мультидиапазон не простирается левее второго?
`'{[7,20)}'::int8multirange &> '{[5,10)}'::int8multirange` → `t`

`anymultirange &> anyrange` → `boolean`
Мультидиапазон не простирается левее заданного диапазона?
`'{[7,20)}'::int8multirange &> int8range(5,10)` → `t`

`anyrange &> anymultirange` → `boolean`
Диапазон не простирается левее заданного мультидиапазона?
`int8range(7,20) &> '{[5,10)}'::int8multirange` → `t`

`anymultirange -|- anymultirange` → `boolean`
Мультидиапазоны примыкают друг к другу?
`'{[1.1,2.2)}'::nummultirange -|- '{[2.2,3.3)}'::nummultirange` → `t`

`anymultirange -|- anyrange` → `boolean`
Мультидиапазон примыкает к заданному диапазону?
`'{[1.1,2.2)}'::nummultirange -|- numrange(2.2,3.3)` → `t`

`anyrange -|- anymultirange` → `boolean`
Диапазон примыкает к заданному мультидиапазону?
`numrange(1.1,2.2) -|- '{[2.2,3.3)}'::nummultirange` → `t`

`anymultirange + anymultirange` → `anymultirange`
Вычисляет объединение мультидиапазонов. Мультидиапазоны могут не пересекаться и не примыкать друг к другу.
`'{[5,10)}'::nummultirange + '{[15,20)}'::nummultirange` → `{[5,10), [15,20)}`

`anymultirange * anymultirange` → `anymultirange`

Оператор **Описание** **Пример(ы)**
Вычисляет пересечение мультидиапазонов. `'{[5,15)}'::int8multirange * '{[10,20)}'::int8multirange` → `{[10,15)}`

`anymultirange - anymultirange` → `anymultirange`
Вычисляет разность мультидиапазонов.
`'{[5,20)}'::int8multirange - '{[10,15)}'::int8multirange` → `{[5,10), [15,20)}`

Операторы слева/справа/примыкает всегда возвращают false, если один из диапазонов или мультидиапазонов пуст; то есть, считается, что пустой диапазон находится не слева и не справа от какого-либо другого диапазона.

В других случаях пустые диапазоны и мультидиапазоны рассматриваются как нулевой элемент: всё, что объединяется с пустым значением, остаётся неизменным. Все, от чего отнимается пустое значение, остаётся неизменным. Пустой мультидиапазон обладает теми же свойствами, что и пустой диапазон. Каждый диапазон содержит пустой диапазон. Каждый мультидиапазон содержит сколько угодно пустых диапазонов.

При вычислении операторов объединения и вычитания диапазонов возникнет ошибка, если в результате получатся два непересекающихся поддиапазона, поскольку такой результат нельзя представить в виде диапазона. Существуют отдельные операторы для объединения и вычитания, которые принимают мультидиапазонные параметры и возвращают мультидиапазон, и они не вызовут ошибку, даже если их аргументы не пересекаются. Поэтому, если вы хотите избежать ошибок при объединении или вычитании диапазонов, когда могут получиться непересекающиеся поддиапазоны, сначала преобразуйте входные диапазоны в мультидиапазоны.

В Таблице 9.55 перечислены функции, предназначенные для работы с диапазонными типами. Функции, предназначенные для использования с мультидиапазонными типами, приведены в Таблице 9.59.

Таблица 9.58. Диапазонные функции

Функция **Описание** **Пример(ы)**

`lower ( anyrange )` → `anyelement`
Выдаёт нижнюю границу диапазона (`NULL`, если диапазон пуст или не имеет нижней границы).
`lower(numrange(1.1, 2.2))` → `1.1`

`upper ( anyrange )` → `anyelement`
Выдаёт верхнюю границу диапазона (`NULL`, если диапазон пуст или не имеет верхней границы).
`upper(numrange(1.1, 2.2))` → `2.2`

`isempty ( anyrange )` → `boolean`
Диапазон пуст?
`isempty(numrange(1.1,2.2))` → `f`

`lower_inc ( anyrange )` → `boolean`
Нижняя граница диапазона включается в него?
`lower_inc(numrange(1.1, 2.2))` → `t`

`upper_inc ( anyrange )` → `boolean`
Верхняя граница диапазона включается в него?
`upper_inc(numrange(1.1, 2.2))` → `f`

Функция **Описание** **Пример(ы)**

`lower_inf` (`anyrange`) → `boolean`
У диапазона нет нижней границы? (При указании нижней границы `-Infinity` возвращается false.)
`lower_inf('(,)'::daterange)` → `t`

`upper_inf` (`anyrange`) → `boolean`
У диапазона нет верхней границы? (При указании верхней границы `Infinity` возвращается false.)
`upper_inf('(,)'::daterange)` → `t`

`range_merge` (`anyrange`, `anyrange`) → `anyrange`
Вычисляет наименьший диапазон, включающий оба заданных диапазона.
`range_merge('[1,2)'::int4range, '[3,4)'::int4range)` → `[1,4)`

Таблица 9.59. Мультидиапазонные функции

Функция **Описание** **Пример(ы)**

`lower` (`anymultirange`) → `anyelement`
Выдаёт нижнюю границу мультидиапазона (`NULL`, если мультидиапазон пуст или не имеет нижней границы).
`lower('{[1.1,2.2)}'::nummultirange)` → `1.1`

`upper` (`anymultirange`) → `anyelement`
Выдаёт верхнюю границу мультидиапазона (`NULL`, если мультидиапазон пуст или не имеет верхней границы).
`upper('{[1.1,2.2)}'::nummultirange)` → `2.2`

`isempty` (`anymultirange`) → `boolean`
Мультидиапазон пуст?
`isempty('{[1.1,2.2)}'::nummultirange)` → `f`

`lower_inc` (`anymultirange`) → `boolean`
Нижняя граница мультидиапазона включается в него?
`lower_inc('{[1.1,2.2)}'::nummultirange)` → `t`

`upper_inc` (`anymultirange`) → `boolean`
Верхняя граница мультидиапазона включается в него?
`upper_inc('{[1.1,2.2)}'::nummultirange)` → `f`

`lower_inf` (`anymultirange`) → `boolean`
У мультидиапазона нет нижней границы? (При указании нижней границы `-Infinity` возвращается false.)
`lower_inf('{(,)}'::datemultirange)` → `t`

`upper_inf` (`anymultirange`) → `boolean`
У мультидиапазона нет верхней границы? (При указании верхней границы `Infinity` возвращается false.)
`upper_inf('{(,)}'::datemultirange)` → `t`

`range_merge` (`anymultirange`) → `anyrange`
Вычисляет наименьший диапазон, включающий весь заданный мультидиапазон.
`range_merge('{[1,2), [3,4)}'::int4multirange)` → `[1,4)`

Функция **Описание** **Пример(ы)**

`multirange` (`anyrange`) → `anymultirange`
Возвращает мультидиапазон, содержащий только заданный диапазон.
`multirange('[1,2)'::int4range)` → `{[1,2)}`

`unnest` (`anymultirange`) → `set of anyrange`
Разворачивает мультидиапазон в набор диапазонов в восходящем порядке.
`unnest('{[1,2), [3,4)}'::int4multirange)` →
`[1,2)`
`[3,4)`

Если функциям `lower_inc`, `upper_inc`, `lower_inf`, `upper_inf` передаётся пустой диапазон или мультидиапазон, они возвращают false.

9.21. Агрегатные функции

Агрегатные функции получают единственный результат из набора входных значений. Встроенные агрегатные функции общего назначения перечислены в Таблице 9.60, а статистические агрегатные функции — в Таблице 9.61. Встроенные внутригрупповые сортирующие агрегатные функции перечислены в Таблице 9.62, встроенные внутригрупповые гипотезирующие — в Таблице 9.63. Группирующие операторы, тесно связанные с агрегатными функциями, перечислены в Таблице 9.64. Особенности синтаксиса агрегатных функцией разъясняются в Подразделе 4.2.7. За дополнительной вводной информацией обратитесь к Разделу 2.7.

Агрегатные функции, поддерживающие *частичный режим*, являются кандидатами на участие в различных оптимизациях, например, в параллельном агрегировании.

Хотя все агрегатные функции ниже принимают необязательное предложение `ORDER BY` (как описано в Подразделе 4.2.7), это предложение было добавлено только для тех функций, вывод которых зависит от упорядочения.

Таблица 9.60. Агрегатные функции общего назначения

Функция **Описание**	**Частичный режим**
`any_value` (`anyelement`) → *`тот же тип, что на входе`* Возвращает произвольное значение из входных значений, отличных от NULL.	Да
`array_agg` (`anynonarray` `ORDER BY` `input_sort_columns`) → `anyarray` Собирает все входные значения, включая NULL, в массив.	Да
`array_agg` (`anyarray` `ORDER BY` `input_sort_columns`) → `anyarray` Собирает все входные массивы в массив с размерностью больше на один. (Входные массивы должны иметь одинаковую размерность и не могут быть пустыми или равны NULL).	Да

`avg` (`smallint`) → `numeric`
`avg` (`integer`) → `numeric`
`avg` (`bigint`) → `numeric`
`avg` (`numeric`) → `numeric`
`avg` (`real`) → `double precision`
`avg` (`double precision`) → `double precision`
`avg` (`interval`) → `interval`
	Да

Функция **Описание**	**Частичный режим**
Вычисляет арифметическое среднее для всех входных значений, отличных от NULL.	

`bit_and ( smallint ) → smallint`
`bit_and ( integer ) → integer`
`bit_and ( bigint ) → bigint`
`bit_and ( bit ) → bit`
Вычисляет побитовое И для всех входных значений, отличных от NULL.
	Да

`bit_or ( smallint ) → smallint`
`bit_or ( integer ) → integer`
`bit_or ( bigint ) → bigint`
`bit_or ( bit ) → bit`
Вычисляет побитовое ИЛИ для всех входных значений, отличных от NULL.
	Да

`bit_xor ( smallint ) → smallint`
`bit_xor ( integer ) → integer`
`bit_xor ( bigint ) → bigint`
`bit_xor ( bit ) → bit`
Вычисляет побитовое исключающее ИЛИ для всех входных значений, отличных от NULL. Может быть полезна для вычисления контрольной суммы неупорядоченного набора значений.
	Да

`bool_and ( boolean ) → boolean` Возвращает true, если все входные значения, отличные от NULL, равны true, и false в противном случае.	Да
`bool_or ( boolean ) → boolean` Возвращает true, если хотя бы одно входное значение, отличное от NULL, равно true, и false в противном случае.	Да
`count ( * ) → bigint` Выдаёт количество входных строк.	Да
`count ( "any" ) → bigint` Выдаёт количество входных строк, в которых входное значение отлично от NULL.	Да
`every ( boolean ) → boolean` Это соответствующий стандарту SQL аналог `bool_and`.	Да

`json_agg ( anyelement ORDER BY input_sort_columns ) → json`
`jsonb_agg ( anyelement ORDER BY input_sort_columns ) → jsonb`
Собирает все входные значения, включая NULL, в JSON-массив. Значения преобразуются в JSON методом `to_json` или `to_jsonb`.
	Нет

`json_agg_strict ( anyelement ) → json`
`jsonb_agg_strict ( anyelement ) → jsonb`
Собирает все входные значения, исключая NULL, в JSON-массив. Значения преобразуются в JSON методом `to_json` или `to_jsonb`.
	Нет

`json_arrayagg ( [`*`выражение_значения`*` ] [ORDER BY `*`выражение_сортировки`*` ] [ { NULL | ABSENT } ON NULL] [RETURNING `*`тип_данных`*` [FORMAT JSON [ENCODING UTF8]]])` Ведёт себя так же, как функция `json_array`, но агрегатная, поэтому принимает только один параметр *`выражение_значения`*. С указанием `ABSENT ON NULL`	Нет

Функция **Описание**	**Частичный режим**
любые значения NULL пропускаются. С указанием `ORDER BY` элементы добавляются в массив в заданном порядке, а не в том порядке, в котором они вводятся. `SELECT json_arrayagg(v) FROM (VALUES(2),(1)) t(v)` → `[2, 1]`	
`json_objectagg` ([{ *выражение_ключа* { `VALUE` | ':' } *выражение_значения* }] [{ `NULL` | `ABSENT` } `ON NULL`] [{ `WITH` | `WITHOUT` } `UNIQUE` [`KEYS`]] [`RETURNING` *тип_данных* [`FORMAT JSON` [`ENCODING UTF8`]]]) Ведёт себя так же, как описанная выше функция `json_object`, но является агрегатной, поэтому принимает только один аргумент *выражение_ключа* и один аргумент *выражение_значения*. `SELECT json_objectagg(k:v) FROM (VALUES ('a'::text,current_date), ('b',current_date + 1)) AS t(k,v)` → `{ "a" : "2022-05-10", "b" : "2022-05-11" }`	Нет

`json_object_agg` (*key* "any", *value* "any" `ORDER BY` input_sort_columns) → `json`
`jsonb_object_agg` (*key* "any", *value* "any" `ORDER BY` input_sort_columns) → `jsonb`
Собирает все пары ключ/значение в JSON-объект. Аргументы-ключи приводятся к текстовому типу, а значения преобразуются методом `to_json` или `to_jsonb`. В качестве значений, но не ключей, могут передаваться null.
	Нет

`json_object_agg_strict` (*key* "any", *value* "any") → `json`
`jsonb_object_agg_strict` (*key* "any", *value* "any") → `jsonb`
Собирает все пары ключ/значение в JSON-объект. Аргументы-ключи приводятся к текстовому типу, а значения преобразуются методом `to_json` или `to_jsonb`. Аргумент *key* должен быть отличен от NULL. Если *value* равно NULL, запись пропускается.
	Нет

`json_object_agg_unique` (*key* "any", *value* "any") → `json`
`jsonb_object_agg_unique` (*key* "any", *value* "any") → `jsonb`
Собирает все пары ключ/значение в JSON-объект. Аргументы-ключи приводятся к текстовому типу, а значения преобразуются методом `to_json` или `to_jsonb`. В качестве значений, но не ключей, могут передаваться null. При наличии повторяющихся ключей выдаётся ошибка.
	Нет

`json_object_agg_unique_strict` (*key* "any", *value* "any") → `json`
`jsonb_object_agg_unique_strict` (*key* "any", *value* "any") → `jsonb`
Собирает все пары ключ/значение в JSON-объект. Аргументы-ключи приводятся к текстовому типу, а значения преобразуются методом `to_json` или `to_jsonb`. В качестве значений, но не ключей, могут передаваться null. Если *value* равно null, запись пропускается. Если есть повторяющийся ключ, выдаётся ошибка.
	Нет

`max` (*см. описание*) → *тот же тип, что на входе* Вычисляет максимальное из всех значений, отличных от NULL. Имеется для всех числовых и строковых типов, типов-перечислений и даты/времени, а также типов `inet`, `interval`, `money`, `oid`, `pg_lsn`, `tid`, `xid8` и массивов с элементами этих типов.	Да
`min` (*см. описание*) → *тот же тип, что на входе* Вычисляет минимальное из всех значений, отличных от NULL. Имеется для всех числовых и строковых типов, типов-перечислений и даты/времени, а также типов `inet`, `interval`, `money`, `oid`, `pg_lsn`, `tid`, `xid8` и массивов с элементами этих типов.	Да
`range_agg` (*value* `anyrange`) → `anymultirange`	Нет

Функция **Описание**	**Частичный режим**
`range_agg` (*`value`* `anymultirange`) → `anymultirange` Вычисляет объединение всех входных значений, отличных от NULL.	

`range_intersect_agg` (*`value`* `anyrange`) → `anyrange`
`range_intersect_agg` (*`value`* `anymultirange`) → `anymultirange`
Вычисляет пересечение всех входных значений, отличных от NULL.
	Нет

`string_agg` (*`value`* `text`, *`delimiter`* `text`) → `text`
`string_agg` (*`value`* `bytea`, *`delimiter`* `bytea` `ORDER BY` `input_sort_columns`) → `bytea`
Соединяет все входные значения, отличные от NULL, в строку. Перед каждым значением, кроме первого, добавляется соответствующий разделитель, заданный параметром *`delimiter`* (если он отличен от NULL).
	Да

`sum` (`smallint`) → `bigint`
`sum` (`integer`) → `bigint`
`sum` (`bigint`) → `numeric`
`sum` (`numeric`) → `numeric`
`sum` (`real`) → `real`
`sum` (`double precision`) → `double precision`
`sum` (`interval`) → `interval`
`sum` (`money`) → `money`
Вычисляет сумму всех входных значений, отличных от NULL.
	Да

`xmlagg` (`xml` `ORDER BY` `input_sort_columns`) → `xml` Соединяет вместе входные XML-значения, отличные от NULL (см. также Подраздел 9.15.1.8).	Нет

Следует заметить, что за исключением `count`, все эти функции возвращают NULL, если для них не была выбрана ни одна строка. В частности, функция `sum`, не получив строк, возвращает NULL, а не 0, как можно было бы ожидать, и `array_agg` в этом случае возвращает NULL, а не пустой массив. Если необходимо, подставить в результат 0 или пустой массив вместо NULL можно с помощью функции `coalesce`.

Агрегатные функции `array_agg`, `json_agg`, `jsonb_agg`, `json_agg_strict`, `jsonb_agg_strict`, `json_object_agg`, `jsonb_object_agg`, `json_object_agg_strict`, `jsonb_object_agg_strict`, `json_object_agg_unique`, `jsonb_object_agg_unique`, `json_object_agg_unique_strict`, `jsonb_object_agg_unique_strict`, `string_agg` и `xmlagg` так же, как и подобные пользовательские агрегатные функции, выдают разные по содержанию результаты в зависимости от порядка входных значений. По умолчанию порядок не определён, но его можно задать, дополнив вызов агрегатной функции предложением `ORDER BY`, как описано в Подразделе 4.2.7. Обычно нужного результата также можно добиться, передав для агрегирования результат подзапроса с сортировкой. Например:

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

Но учтите, что этот подход может не работать, если на внешнем уровне запроса выполняется дополнительная обработка, например соединение, так как при этом результат подзапроса может быть переупорядочен перед вычислением агрегатной функции.

Примечание

Логические агрегатные функции `bool_and` и `bool_or` равнозначны описанным в стандарте SQL агрегатам `every` и `any` или `some`. Postgres Pro поддерживает `every`, но не `any`/`some`, так как синтаксис, описанный в стандарте, допускает неоднозначность:

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Здесь `ANY` можно рассматривать и как объявление подзапроса, и как агрегатную функцию, если этот подзапрос возвращает одну строку с булевым значением. Таким образом, этим агрегатным функциям нельзя было дать стандартные имена.

Примечание

Пользователи с опытом использования других СУБД SQL могут быть недовольны скоростью агрегатной функции `count`, когда она применяется ко всей таблице. Подобный запрос:

```
SELECT count(*) FROM sometable;
```

потребует затрат в количестве, пропорциональном размеру таблицы: Postgres Pro придётся полностью просканировать либо всю таблицу, либо один из индексов, включающий все её строки.

В Таблице 9.61 перечислены агрегатные функции, обычно применяемые в статистическом анализе. (Они выделены просто для того, чтобы не загромождать список наиболее популярных агрегатных функций.) Функции, показанные как принимающие *`числовой_тип`*, существуют для типов `smallint`, `integer`, `bigint`, `numeric`, `real`, и `double precision`. В их описании под *`N`* подразумевается число входных строк, для которых все входные выражения отличны от NULL. Во всех случаях, когда вычисление не имеет смысла, например, когда *`N`* равно нулю, возвращается null.

Таблица 9.61. Агрегатные функции для статистических вычислений

Функция **Описание**	**Частичный режим**
`corr` (*`Y`* `double precision`, *`X`* `double precision`) → `double precision` Вычисляет коэффициент корреляции.	Да
`covar_pop` (*`Y`* `double precision`, *`X`* `double precision`) → `double precision` Вычисляет ковариацию совокупности.	Да
`covar_samp` (*`Y`* `double precision`, *`X`* `double precision`) → `double precision` Вычисляет ковариацию выборки.	Да
`regr_avgx` (*`Y`* `double precision`, *`X`* `double precision`) → `double precision` Вычисляет среднее для независимой переменной, `sum(` *`X`*`)` / *`N`*.	Да
`regr_avgy` (*`Y`* `double precision`, *`X`* `double precision`) → `double precision` Вычисляет среднее для зависимой переменной, `sum(` *`Y`*`)` / *`N`*.	Да
`regr_count` (*`Y`* `double precision`, *`X`* `double precision`) → `bigint` Вычисляет число входных строк, в которых оба выражения отличны от NULL.	Да
`regr_intercept` (*`Y`* `double precision`, *`X`* `double precision`) → `double precision` Вычисляет пересечение с осью OY линии, полученной методом наименьших квадратов по парам (*`X`*, *`Y`*).	Да
`regr_r2` (*`Y`* `double precision`, *`X`* `double precision`) → `double precision` Вычисляет квадрат коэффициента корреляции.	Да
`regr_slope` (*`Y`* `double precision`, *`X`* `double precision`) → `double precision` Вычисляет наклон линии, полученной методом наименьших квадратов по парам (*`X`*, *`Y`*).	Да

Функция **Описание**	**Частичный режим**
`regr_sxx` (*Y* `double precision`, *X* `double precision`) → `double precision` Вычисляет «сумму квадратов» независимой переменной, `sum(`*X*`^2) - sum(`*X*`)^2/`*N*.	Да
`regr_sxy` (*Y* `double precision`, *X* `double precision`) → `double precision` Вычисляет «сумму произведений» независимой и зависимой переменных, `sum(`*X*`*`*Y*`) - sum(`*X*`) * sum(`*Y*`)/`*N*.	Да
`regr_syy` (*Y* `double precision`, *X* `double precision`) → `double precision` Вычисляет «сумму квадратов» зависимой переменной, `sum(`*Y*`^2) - sum(`*Y*`)^2/`*N*.	Да
`stddev` (*числовой_тип*) → `double precision` для `real` или `double precision`, иначе `numeric` Сохранившийся синоним `stddev_samp`.	Да
`stddev_pop` (*numeric_type*) → `double precision` для `real` или `double precision`, иначе `numeric` Вычисляет стандартное отклонение по генеральной совокупности входных значений.	Да
`stddev_samp` (*числовой_тип*) → `double precision` для `real` или `double precision`, иначе `numeric` Вычисляет стандартное отклонение по выборке входных значений.	Да
`variance` (*numeric_type*) → `double precision` для `real` или `double precision`, иначе `numeric` Сохранившийся синоним `var_samp`.	Да
`var_pop` (*numeric_type*) → `double precision` для `real` или `double precision`, иначе `numeric` Вычисляет дисперсию для генеральной совокупности входных значений (квадрат стандартного отклонения).	Да
`var_samp` (*числовой_тип*) → `double precision` для `real` или `double precision`, иначе `numeric` Вычисляет дисперсию по выборке для входных значений (квадрат отклонения по выборке).	Да

В Таблица 9.62 показаны некоторые агрегатные функции, использующие синтаксис *сортирующих агрегатных функций*. Эти функции иногда называются функциями «обратного распределения». Их агрегированные входные данные формируются указанием `ORDER BY`, а кроме того они могут принимать не агрегируемый *непосредственный аргумент*, который вычисляется только один раз. Все эти функции игнорируют значения NULL в агрегируемых данных. Для тех функций, которые принимают параметр *`fraction`*, его значение должно быть между 0 и 1; в противном случае выдаётся ошибка. Однако если значение *`fraction`* — NULL, они выдают NULL в результате.

Таблица 9.62. Сортирующие агрегатные функции

Функция **Описание**	**Частичный режим**
`mode` () `WITHIN GROUP` (`ORDER BY anyelement`) → `anyelement`	Нет

Функция **Описание**	**Частичный режим**
Вычисляет *моду* — наиболее часто встречающееся в агрегируемом аргументе значение (если одинаково часто встречаются несколько значений, произвольно выбирается первое из них). Агрегируемый аргумент должен быть сортируемого типа.	

`percentile_cont ( fraction double precision ) WITHIN GROUP ( ORDER BY double precision )` → `double precision`
`percentile_cont ( fraction double precision ) WITHIN GROUP ( ORDER BY interval )` → `interval`
Вычисляет *непрерывный процентиль* — значение, соответствующее дроби, заданной параметром *`fraction`*, в отсортированном множестве значений агрегатного аргумента. При этом в случае необходимости соседние входные значения будут интерполироваться.
	Нет

`percentile_cont ( fractions double precision[] ) WITHIN GROUP ( ORDER BY double precision )` → `double precision[]`
`percentile_cont ( fractions double precision[] ) WITHIN GROUP ( ORDER BY interval )` → `interval[]`
Вычисляет множественные непрерывные процентили. Возвращает массив той же размерности, что имеет параметр *`fractions`*, в котором каждый отличный от NULL элемент заменяется соответствующим данному перцентилю значением (возможно интерполированным).
	Нет

`percentile_disc ( fraction double precision ) WITHIN GROUP ( ORDER BY anyelement )` → `anyelement` Вычисляет *дискретный процентиль* — первое значение в отсортированном множестве значений агрегатного аргумента, позиция которого в этом множестве равна или больше значения *`fraction`*. Агрегируемый аргумент должен быть сортируемого типа.	Нет
`percentile_disc ( fractions double precision[] ) WITHIN GROUP ( ORDER BY anyelement )` → `anyarray` Вычисляет множественные дискретные процентили. Возвращает массив той же размерности, что имеет параметр *`fractions`*, в котором каждый отличный от NULL элемент заменяется соответствующим данному перцентилю значением. Агрегируемый аргумент должен быть сортируемого типа.	Нет

Все «гипотезирующие» агрегатные функции, перечисленные в Таблице 9.63, связаны с одноимёнными оконными функциями, определёнными в Разделе 9.22. В каждом случае их результат — значение, которое бы вернула связанная оконная функция для «гипотетической» строки, полученной из *`аргументов`*, если бы такая строка была добавлена в сортированную группу строк, которую образуют *`сортирующие_аргументы`*. Для всех этих функций список непосредственных аргументов, переданный в качестве *`аргументов`*, по числу и типу элементов должен соответствовать списку, передаваемому в качестве *`сортирующих_аргументов`*. В отличие от большинства встроенных агрегатов, данные агрегаты не являются строгими, то есть они не игнорируют строки, содержащие NULL. Значения NULL сортируются по правилу, заданному в предложении `ORDER BY`.

Таблица 9.63. Гипотезирующие агрегатные функции

Функция **Описание**	**Частичный режим**
`rank ( аргументы ) WITHIN GROUP ( ORDER BY сортирующие_аргументы )` → `bigint`	Нет

Функция **Описание**	**Частичный режим**
Вычисляет ранг гипотетической строки с пропусками, то есть номер первой родственной ей строки.	
`dense_rank` (*`args`*) `WITHIN GROUP` (`ORDER BY` *`сортируемые_аргументы`*) → `bigint` Вычисляет ранг гипотетической строки без пропусков; по сути эта функция считает группы родственных строк.	Нет
`percent_rank` (*`аргументы`*) `WITHIN GROUP` (`ORDER BY` *`сортируемые_аргументы`*) → `double precision` Вычисляет относительный ранг гипотетической строки, то есть (`rank` - 1) / (общее число строк - 1). Таким образом, результат лежит в интервале от 0 до 1, включая границы.	Нет
`cume_dist` (*`аргументы`*) `WITHIN GROUP` (`ORDER BY` *`сортируемые_аргументы`*) → `double precision` Вычисляет кумулятивное распределение, то есть (число строк, предшествующих или родственных гипотетической строке) / (общее число строк). Результат лежит в интервале от 1/*`N`* до 1.	Нет

Таблица 9.64. Операции группировки

Функция **Описание**
`GROUPING` (*`выражения_group_by`*) → `integer` Выдаёт битовую маску, показывающую, какие выражения `GROUP BY` не вошли в текущий набор группирования. Биты назначаются справа налево, то есть самому правому аргументу соответствует самый младший бит; бит равен 0, если соответствующее выражение вошло в критерий группировки набора группирования, для которого сформирована текущая строка результата, или 1 в противном случае.

Операции группировки, показанные в Таблице 9.64, применяются в сочетании с наборами группирования (см. Подраздел 7.2.4) для различения результирующих строк. Аргументы функции `GROUPING` на самом деле не вычисляются, но они должны в точности соответствовать выражениям, заданным в предложении `GROUP BY` на их уровне запроса. Например:

```
=> SELECT * FROM items_sold;
 make  | model | sales
-------+-------+-------
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM items_sold GROUP BY
 ROLLUP(make,model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
 Foo   |       |        1 | 30
 Bar   |       |        1 | 20
       |       |        3 | 50
```

```
(7 rows)
```

Здесь значение `grouping`, равное `0` в первых четырёх строках, показывает, что эти строки были сгруппированы обычным образом по обоим группирующим столбцам. Значение `1` в двух предпоследних строках показывает, что столбец `model` не был группирующим, а значение `3` в последней строке показывает, что при группировании не учитывались ни `make`, ни `model`, то есть агрегирование выполнялось по всем входным строкам.

9.22. Оконные функции

Оконные функции дают возможность выполнять вычисления с набором строк, каким-либо образом связанным с текущей строкой запроса. Вводную информацию об этом можно получить в Разделе 3.5, а подробнее узнать о синтаксисе можно в Подразделе 4.2.8.

Встроенные оконные функции перечислены в Таблице 9.65. Заметьте, что эти функции *должны* вызываться именно как оконные, т. е. при вызове необходимо использовать предложение `OVER`.

В дополнение к этим функциям в качестве оконных можно использовать любые встроенные или обычные пользовательские агрегатные функции (но не сортирующие и не гипотезирующие); список встроенных агрегатных функций приведён в Разделе 9.21. Агрегатные функции работают как оконные, только когда за их вызовом следует предложение `OVER`; в противном случае они работают как обычные функции и выдают для всего набора единственную строку.

Таблица 9.65. Оконные функции общего назначения

Функция	Описание
`row_number` () → `bigint`	Возвращает номер текущей строки в её разделе, начиная с 1.
`rank` () → `bigint`	Возвращает ранг текущей строки с пропусками; то же, что и `row_number` для первой родственной ей строки.
`dense_rank` () → `bigint`	Возвращает ранг текущей строки без пропусков; по сути эта функция считает группы родственных строк.
`percent_rank` () → `double precision`	Вычисляет относительный ранг текущей строки, то есть (`rank` - 1) / (общее число строк раздела - 1). Таким образом, результат лежит в интервале от 0 до 1, включительно.
`cume_dist` () → `double precision`	Возвращает кумулятивное распределение, то есть (число строк раздела, предшествующих или родственных текущей строке) / (общее число строк раздела). Таким образом, результат лежит в интервале от 1/*N* до 1.
`ntile` (*`num_buckets`* `integer`) → `integer`	Возвращает целое от 1 до значения аргумента для разбиения раздела на части максимально близких размеров.
`lag` (*`value`* `anycompatible` [, *`offset`* `integer` [, *`default`* `anycompatible`]]) → `anycompatible`	Возвращает значение *`value`*, вычисленное для строки, сдвинутой на *`offset`* строк от текущей к началу раздела; если такой строки нет, возвращается значение *`default`* (оно должно быть совместимого с *`value`* типа). Оба аргумента, *`offset`* и *`default`*, вычисляются для текущей строки. Если они не указываются, *`offset`* считается равным 1, а *`default`* — `NULL`.
`lead` (*`value`* `anycompatible` [, *`offset`* `integer` [, *`default`* `anycompatible`]]) → `anycompatible`	Возвращает значение *`value`*, вычисленное для строки, сдвинутой на *`offset`* строк от текущей к концу раздела; если такой строки нет, возвращается значение *`default`* (оно

Функция Описание
должно быть совместимого с *`value`* типа). Оба аргумента, *`offset`* и *`default`*, вычисляются для текущей строки. Если они не указываются, *`offset`* считается равным 1, а *`default`* — `NULL`.
`first_value` (*`value`* `anyelement`) → `anyelement` Возвращает значение (*`value`*), вычисленное для первой строки в рамке окна.
`last_value` (*`value`* `anyelement`) → `anyelement` Возвращает значение (*`value`*), вычисленное для последней строки в рамке окна.
`nth_value` (*`value`* `anyelement`, *`n`* `integer`) → `anyelement` Возвращает значение (*`value`*), вычисленное в *`n`*-ой строке в рамке окна (считая с 1), или `NULL`, если такой строки нет.

Результат всех функций, перечисленных в Таблице 9.65, зависит от порядка сортировки, заданного предложением `ORDER BY` в определении соответствующего окна. Строки, которые являются одинаковыми при рассмотрении только столбцов `ORDER BY`, считаются *родственными*. Четыре функции, вычисляющие ранг (включая `cume_dist`), реализованы так, что их результат будет одинаковым для всех родственных строк.

Заметьте, что функции `first_value`, `last_value` и `nth_value` рассматривают только строки в «рамке окна», которая по умолчанию содержит строки от начала раздела до последней родственной строки для текущей. Поэтому результаты `last_value` и иногда `nth_value` могут быть не очень полезны. В таких случаях можно переопределить рамку, добавив в предложение `OVER` подходящее указание рамки (`RANGE`, `ROWS` или `GROUPS`). Подробнее эти указания описаны в Подразделе 4.2.8.

Когда в качестве оконной функции используется агрегатная, она обрабатывает строки в рамке текущей строки. Агрегатная функция с `ORDER BY` и определением рамки окна по умолчанию будет вычисляться как «бегущая сумма», что может не соответствовать желаемому результату. Чтобы агрегатная функция работала со всем разделом, следует опустить `ORDER BY` или использовать `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`. Используя другие указания в определении рамки, можно получить и другие эффекты.

> **Примечание**
>
> В стандарте SQL определены параметры `RESPECT NULLS` или `IGNORE NULLS` для функций `lead`, `lag`, `first_value`, `last_value` и `nth_value`. В Postgres Pro такие параметры не реализованы: эти функции ведут себя так, как положено в стандарте по умолчанию (или с подразумеваемым параметром `RESPECT NULLS`). Также функция `nth_value` не поддерживает предусмотренные стандартом параметры `FROM FIRST` и `FROM LAST`: реализовано только поведение по умолчанию (с подразумеваемым параметром `FROM FIRST`). (Получить эффект параметра `FROM LAST` можно, изменив порядок `ORDER BY` на обратный.)

9.23. Функции поддержки MERGE

Postgres Pro включает функцию поддержки `MERGE`, которую можно использовать в списке `RETURNING` команды MERGE, чтобы определить действие, выполняемое для каждой строки; см. Таблицу 9.66.

Таблица 9.66. Функции поддержки MERGE

Функция Описание
`merge_action` ( ) → `text` Возвращает действие MERGE, выполненное для текущей строки. Это может быть `'INSERT'`, `'UPDATE'` или `'DELETE'`.

Пример:

```
MERGE INTO products p
  USING stock s ON p.product_id = s.product_id
  WHEN MATCHED AND s.quantity > 0 THEN
    UPDATE SET in_stock = true, quantity = s.quantity
  WHEN MATCHED THEN
    UPDATE SET in_stock = false, quantity = 0
  WHEN NOT MATCHED THEN
    INSERT (product_id, in_stock, quantity)
      VALUES (s.product_id, true, s.quantity)
  RETURNING merge_action(), p.*;

 merge_action | product_id | in_stock | quantity
--------------+------------+----------+----------
 UPDATE       |       1001 | t        |       50
 UPDATE       |       1002 | f        |        0
 INSERT       |       1003 | t        |       10
```

Обратите внимание, что эта функция может использоваться только в списке `RETURNING` команды `MERGE`. Использование её в любой другой части запроса не допускается.

9.24. Выражения подзапросов

В этом разделе описаны выражения подзапросов, которые реализованы в Postgres Pro в соответствии со стандартом SQL. Все рассмотренные здесь формы выражений возвращает булевы значения (true/false).

9.24.1. `EXISTS`

`EXISTS (`*`подзапрос`*`)`

Аргументом `EXISTS` является обычный оператор `SELECT`, т. е. *подзапрос*. Выполнив запрос, система проверяет, возвращает ли он строки в результате. Если он возвращает минимум одну строку, результатом `EXISTS` будет «true», а если не возвращает ни одной — «false».

Подзапрос может обращаться к переменным внешнего запроса, которые в рамках одного вычисления подзапроса считаются константами.

Вообще говоря, подзапрос может выполняться не полностью, а завершаться, как только будет возвращена хотя бы одна строка. Поэтому в подзапросах следует избегать побочных эффектов (например, обращений к генераторам последовательностей); проявление побочного эффекта может быть непредсказуемым.

Так как результат этого выражения зависит только от того, возвращаются строки или нет, но не от их содержимого, список выходных значений подзапроса обычно не имеет значения. Как следствие, широко распространена практика, когда проверки `EXISTS` записываются в форме `EXISTS(SELECT 1 WHERE ...)`. Однако из этого правила есть и исключения, например с подзапросами с предложением `INTERSECT`.

Этот простой пример похож на внутреннее соединение по столбцу `col2`, но он выдаёт максимум одну строку для каждой строки в `tab1`, даже если в `tab2` ей соответствуют несколько строк:

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.24.2. `IN`

`выражение`` IN (`*`подзапрос`*`)`

В правой стороне этого выражения в скобках задаётся подзапрос, который должен возвращать ровно один столбец. Вычисленное значение левого выражения сравнивается со значениями во всех строках, возвращённых подзапросом. Результатом всего выражения `IN` будет «true», если строка с таким значением находится, и «false» в противном случае (в том числе, когда подзапрос вообще не возвращает строк).

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция `IN` возвращает NULL, а не false. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Так же, как и с `EXISTS`, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

`*конструктор_строки* IN (*подзапрос*)`

В левой части этой формы `IN` записывается конструктор строки (подробнее они рассматриваются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен вернуть ровно столько столбцов, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращённых подзапросом. Результатом всего выражения `IN` будет «true», если строка с такими значениями находится, и «false» в противном случае (в том числе, когда подзапрос вообще не возвращает строк).

Как обычно, значения NULL в строках обрабатываются при этом по принятым в SQL правилам сравнения. Две строки считаются равными, если все их соответствующие элементы не равны NULL, но равны между собой; неравными они считаются, когда в них находятся элементы, не равные NULL, и не равные друг другу; в противном случае результат сравнения строк не определён (равен NULL). Если в результатах сравнения строк нет ни одного положительного, но есть хотя бы один NULL, результатом `IN` будет NULL.

9.24.3. `NOT IN`

`*выражение* NOT IN (*подзапрос*)`

Справа в скобках записывается подзапрос, который должен возвращать ровно один столбец. Вычисленное значение левого выражения сравнивается со значением во всех строках, возвращённых подзапросом. Результатом всего выражения `NOT IN` будет «true», если находятся только несовпадающие строки (в том числе, когда подзапрос вообще не возвращает строк). Если же находится хотя бы одна подходящая строка, результатом будет «false».

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция `NOT IN` возвращает NULL, а не true. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Так же, как и с `EXISTS`, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

`*конструктор_строки* NOT IN (*подзапрос*)`

В левой части этой формы `NOT IN` записывается конструктор строки (подробнее они описываются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен вернуть ровно столько столбцов, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращённых подзапросом. Результатом всего выражения `NOT IN` будет «true», если равных строк не найдётся (в том числе, и когда подзапрос не возвращает строк), и «false», если такие строки есть.

Как обычно, значения NULL в строках обрабатываются при этом по принятым в SQL правилам сравнения. Две строки считаются равными, если все их соответствующие элементы не равны NULL, но равны между собой; неравными они считаются, когда в них находятся элементы, не равные NULL, и не равные друг другу; в противном случае результат сравнения строк не определён (равен NULL). Если в результатах сравнения строк нет ни одного положительного, но есть хотя бы один NULL, результатом `NOT IN` будет NULL.

9.24.4. `ANY`/`SOME`

```
выражение оператор ANY (подзапрос)
выражение оператор SOME (подзапрос)
```

В правой части конструкции в скобках записывается подзапрос, который должен возвращать ровно один столбец. Вычисленное значение левого выражения сравнивается со значением в каждой строке результата подзапроса с помощью заданного *оператора* условия, который должен выдавать логическое значение. Результатом `ANY` будет «true», если хотя бы для одной строки условие истинно, и «false» в противном случае (в том числе, и когда подзапрос не возвращает строк).

Ключевое слово `SOME` является синонимом `ANY`. Конструкцию `IN` можно также записать как `= ANY`.

Заметьте, что если условие не выполняется ни для одной из строк, а хотя бы для одной строки условный оператор выдаёт NULL, конструкция `ANY` возвращает NULL, а не false. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Так же, как и с `EXISTS`, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

```
конструктор_строки оператор ANY (подзапрос)
конструктор_строки оператор SOME (подзапрос)
```

В левой части этой формы `ANY` записывается конструктор строки (подробнее они описываются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен возвращать ровно столько столбцов, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращённых подзапросом, с применением заданного *оператора*. Результатом всего выражения `ANY` будет «true», если для какой-либо из строк подзапроса результатом сравнения будет true, или «false», если для всех строк результатом сравнения оказывается false (в том числе, и когда подзапрос не возвращает строк). Результатом выражения будет NULL, если ни для одной из строк подзапроса результат сравнения не равен true, а минимум для одной равен NULL.

Подробнее логика сравнения конструкторов строк описана в Подразделе 9.25.5.

9.24.5. `ALL`

```
выражение оператор ALL (подзапрос)
```

В правой части конструкции в скобках записывается подзапрос, который должен возвращать ровно один столбец. Вычисленное значение левого выражения сравнивается со значением в каждой строке результата подзапроса с помощью заданного *оператора* условия, который должен выдавать логическое значение. Результатом `ALL` будет «true», если условие истинно для всех строк (и когда подзапрос не возвращает строк), или «false», если находятся строки, для которых оно ложно. Результатом выражения будет NULL, если ни для одной из строк подзапроса результат сравнения не равен true, а минимум для одной равен NULL.

Конструкция `NOT IN` равнозначна `<> ALL`.

Так же, как и с `EXISTS`, здесь не следует рассчитывать на то, что подзапрос будет всегда выполняться полностью.

```
конструктор_строки оператор ALL (подзапрос)
```

В левой части этой формы `ALL` записывается конструктор строки (подробнее они описываются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен возвращать ровно столько столбцов, сколько содержит строка в выражении слева. Вычисленные значения левого выражения сравниваются построчно со значениями во всех строках, возвращённых подзапросом, с применением заданного *оператора*. Результатом всего выражения `ALL` будет «true», если для всех строк подзапроса результатом сравнения будет true (или если подзапрос не возвращает строк), либо «false», если результат сравнения равен false для любой из строк подзапроса. Результатом выражения будет NULL, если ни для одной из строк подзапроса результат сравнения не равен true, а минимум для одной равен NULL.

Подробнее логика сравнения конструкторов строк описана в Подразделе 9.25.5.

9.24.6. Сравнение единичных строк

```
конструктор_строки оператор (подзапрос)
```

В левой части конструкции записывается конструктор строки (подробнее они описываются в Подразделе 4.2.13). Справа в скобках записывается подзапрос, который должен возвращать ровно столько столбцов, сколько содержит строка в выражении слева. Более того, подзапрос может вернуть максимум одну строку. (Если он не вернёт строк, результатом будет NULL.) Конструкция возвращает результат сравнения строки слева с этой одной строкой результата подзапроса.

Подробнее логика сравнения конструкторов строк описана в Подразделе 9.25.5.

9.25. Сравнение табличных строк и массивов

В этом разделе описываются несколько специальных конструкций, позволяющих сравнивать группы значений. Синтаксис этих конструкций связан с формами выражений с подзапросами, описанными в предыдущем разделе, а отличаются они отсутствием подзапросов. Конструкции, в которых в качестве подвыражений используются массивы, являются расширениями Postgres Pro; все остальные формы соответствуют стандарту SQL. Все описанные здесь выражения возвращают логические значения (true/false).

9.25.1. `IN`

```
выражение IN (значение [, ...])
```

Справа в скобках записывается список выражений. Результатом будет «true», если значение левого выражения равняется одному из значений выражений в правой части. Эту конструкцию можно считать краткой записью условия

```
выражение = значение1
OR
выражение = значение2
OR
...
```

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция `IN` возвращает NULL, а не false. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

9.25.2. `NOT IN`

```
выражение NOT IN (значение [, ...])
```

Справа в скобках записывается список выражений. Результатом будет «true», если значение левого выражения не равно ни одному из значений выражений в правой части. Эту конструкцию можно считать краткой записью условия

```
выражение <> значение1
AND
выражение <> значение2
AND
...
```

Заметьте, что если результатом выражения слева оказывается NULL или равных значений справа не находится, а хотя бы одно из значений справа равно NULL, конструкция `NOT IN` возвращает NULL, а не true, как можно было бы наивно полагать. Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Подсказка

Выражения `x NOT IN y` и `NOT (x IN y)` полностью равнозначны. Учитывая, что значения NULL могут ввести в заблуждение начинающих скорее в конструкции `NOT IN`, чем в `IN`, лучше формулировать условия так, чтобы в них было как можно меньше отрицаний.

9.25.3. `ANY`/`SOME` (с массивом)

```
выражение оператор ANY (выражение массива)
выражение оператор SOME (выражение массива)
```

Справа в скобках записывается выражение, результатом которого является массив. Вычисленное значение левого выражения сравнивается с каждым элементом этого массива с применением заданного *оператора* условия, который должен выдавать логическое значение. Результатом `ANY` будет «true», если для какого-либо элемента условие истинно, и «false» в противном случае (в том числе, и когда массив оказывается пустым).

Если значением массива оказывается NULL, результатом `ANY` также будет NULL. Если NULL получен в левой части, результатом `ANY` обычно тоже будет NULL (хотя оператор нестрогого сравнения может выдать другой результат). Кроме того, если массив в правой части содержит элементы NULL и ни с одним из элементов условие не выполняется, результатом `ANY` будет NULL, а не false (опять же, если используется оператор строгого сравнения). Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

Ключевое слово `SOME` является синонимом `ANY`.

9.25.4. `ALL` (с массивом)

```
выражение оператор ALL (выражение массива)
```

Справа в скобках записывается выражение, результатом которого является массив. Вычисленное значение левого выражения сравнивается с каждым элементом этого массива с применением заданного *оператора* условия, который должен выдавать логическое значение. Результатом `ALL` будет «true», если для всех элементов условие истинно (или массив не содержит элементов), и «false», если находятся строки, для которых оно ложно.

Если значением массива оказывается NULL, результатом `ALL` также будет NULL. Если NULL получен в левой части, результатом `ALL` обычно тоже будет NULL (хотя оператор нестрогого сравнения может выдать другой результат). Кроме того, если массив в правой части содержит элементы NULL и при этом нет элементов, с которыми условие не выполняется, результатом `ALL` будет NULL, а не true (опять же, если используется оператор строгого сравнения). Это соответствует принятым в SQL правилам сравнения переменных со значениями NULL.

9.25.5. Сравнение конструкторов строк

```
конструктор_строки оператор конструктор_строки
```

С обеих сторон представлены конструкторы строк (они описываются в Подразделе 4.2.13). При этом данные конструкторы должны содержать одинаковое число полей. Заданный *оператор* применяется к каждой паре соответствующих полей. (Поскольку поля могут быть разных типов, для каждой пары может быть выбран отдельный конкретный оператор.) Все выбранные операторы должны быть членами некоторого класса операторов B-дерева или быть обратными члену класса операторов B-дерева `=`. Сравнения конструкторов строк возможны с *оператором* `=`, `<>`, `<`, `<=`, `>` или `>=`, или имеющим аналогичную семантику.

Сравнения `=` и `<>` несколько отличаются от других. С этими операторами две строки считаются равными, если все их соответствующие поля не равны NULL и равны между собой, и неравными, если какие-либо соответствующие их поля не NULL и не равны между собой. В противном случае результатом сравнения будет неопределённость (NULL).

С операторами `<`, `<=`, `>` и `>=` элементы строк сравниваются слева направо до тех пор, пока не будет найдена пара неравных элементов или значений NULL. Если любым из элементов пары оказывается NULL, результатом сравнения будет неопределённость (NULL), в противном случае результат всего выражения определяется результатом сравнения этих двух элементов. Например, результатом `ROW(1,2,NULL) < ROW(1,3,0)` будет true, а не NULL, так как третья пара элементов не принимается в рассмотрение.

конструктор_строки `IS DISTINCT FROM` *конструктор_строки*

Эта конструкция похожа на сравнение строк с оператором `<>`, но со значениями NULL она выдаёт не NULL. Любое значение NULL для неё считается неравным (отличным от) любому значению не NULL, а два NULL считаются равными (не различными). Таким образом, результатом такого выражения будет true или false, но не NULL.

конструктор_строки `IS NOT DISTINCT FROM` *конструктор_строки*

Эта конструкция похожа на сравнение строк с оператором `=`, но со значениями NULL она выдаёт не NULL. Любое значение NULL для неё считается неравным (отличным от) любому значению не NULL, а два NULL считаются равными (не различными). Таким образом, результатом такого выражения всегда будет true или false, но не NULL.

9.25.6. Сравнение составных типов

запись оператор запись

Стандарт SQL требует, чтобы при сравнении строк возвращался NULL, если результат зависит от сравнения двух значений NULL или значения NULL и не NULL. Postgres Pro выполняет это требование только при сравнении строк, созданных конструкторами (как описано в Подразделе 9.25.5), или строки, созданной конструктором, со строкой результата подзапроса (как было описано в Разделе 9.24). В других контекстах при сравнении полей составных типов два значения NULL считаются равными, а любое значение не NULL полагается меньшим NULL. Это отклонение от правила необходимо для полноценной реализации сортировки и индексирования составных типов.

После вычисления каждой стороны они сравниваются по строкам. Сравнения составных типов возможны с *оператором* `=`, `<>`, `<`, `<=`, `>` или `>=`, либо другим подобным. (Точнее, оператором сравнения строк может быть любой оператор, входящий в класс операторов B-дерева, либо обратный к оператору `=`, входящему в класс операторов B-дерева.) По умолчанию вышеперечисленные операторы действуют так же, как выражение `IS [ NOT ] DISTINCT FROM` для конструкторов строк (см. Подраздел 9.25.5).

Для поддержки сравнения строк с элементами, для которых не определён класс операторов B-дерева по умолчанию, введены следующие операторы: `*=`, `*<>`, `*<`, `*<=`, `*>` и `*>=`. Эти операторы сравнивают внутреннее двоичное представление двух строк. Учтите, что две строки могут иметь различное двоичное представление, даже когда при сравнении оператором равенства считаются равными. Порядок строк с такими операторами детерминирован, но не несёт смысловой нагрузки. Данные операторы не предназначены для обычных запросов; они применяются внутри системы для материализованных представлений и могут быть полезны для других специальных целей, например для репликации или исключения дубликатов в B-дереве (см. Подраздел 62.1.4.3).

9.26. Функции, возвращающие множества

В этом разделе описаны функции, которые могут возвращать не одну, а множество строк. Чаще всего из их числа используются функции, генерирующие ряды значений, которые перечислены в Таблице 9.67 и Таблице 9.68. Другие, более специализированные функции множеств описаны в других разделах этой документации. Варианты комбинирования нескольких функций, возвращающих множества строк, описаны в Подразделе 7.2.1.4.

Таблица 9.67. Функции, генерирующие ряды значений

Функция **Описание**
`generate_series` (*`start`* `integer`, *`stop`* `integer` [, *`step`* `integer`]) → `setof integer`

Функция **Описание**

`generate_series` (*start* `bigint`, *stop* `bigint` [, *step* `bigint`]) → `setof bigint`
`generate_series` (*start* `numeric`, *stop* `numeric` [, *step* `numeric`]) → `setof numeric`
Выдаёт ряд значений от *start* до *stop* с заданным шагом (*step*), по умолчанию равным 1.

`generate_series` (*start* `timestamp`, *stop* `timestamp`, *step* `interval`) → `setof timestamp`
`generate_series` (*start* `timestamp with time zone`, *stop* `timestamp with time zone`, *step* `interval` [, *timezone* `text`]) → `setof timestamp with time zone`
Выдаёт ряд значений от *start* до *stop* с заданным шагом (*step*). В форме с учётом часового пояса, время суток и поправки на летнее время вычисляются в соответствии с часовым поясом, указанным в аргументе *timezone*, или текущим значением TimeZone при отсутствии аргумента.

Если заданный шаг (*step*) положительный, а *start* оказывается больше *stop*, эти функции возвращают 0 строк. Тот же результат будет, если *step* меньше 0, а *start* меньше *stop*, или если в каком-либо аргументе передаётся `NULL`. Если же *step* будет равен 0, произойдёт ошибка. Несколько примеров:

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
(3 rows)

-- в этом примере задействуется оператор прибавления к дате целого числа:
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)
```

```
SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)

(9 rows) -- в этом примере предполагается, что для TimeZone установлен часовой пояс
 UTC; обратите внимание на переход на летнее время:
SELECT * FROM generate_series('2001-10-22 00:00 -04:00'::timestamptz,
                              '2001-11-01 00:00 -05:00'::timestamptz,
                              '1 day'::interval, 'America/New_York');
    generate_series
------------------------
 2001-10-22 04:00:00+00
 2001-10-23 04:00:00+00
 2001-10-24 04:00:00+00
 2001-10-25 04:00:00+00
 2001-10-26 04:00:00+00
 2001-10-27 04:00:00+00
 2001-10-28 04:00:00+00
 2001-10-29 05:00:00+00
 2001-10-30 05:00:00+00
 2001-10-31 05:00:00+00
 2001-11-01 05:00:00+00
(11 rows)
```

Таблица 9.68. Функции, генерирующие индексы массивов

Функция **Описание**
`generate_subscripts` (*`array`* anyarray, *`dim`* integer) → `setof integer` Выдаёт ряд значений, содержащий все допустимые индексы в размерности *`dim`* данного массива.
`generate_subscripts` (*`array`* anyarray, *`dim`* integer, *`reverse`* boolean) → `setof integer` Выдаёт ряд значений, содержащий все допустимые индексы в размерности *`dim`* данного массива. Если параметр *`reverse`* равен true, значения выдаются от большего к меньшему.

Функция `generate_subscripts` упрощает задачу получения всех допустимых индексов для указанной размерности данного массива. Она выдаёт 0 строк, если в массиве нет указанной размерности или любой из аргументов — `NULL`. Взгляните на следующие примеры:

```
-- простой пример использования:
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
```

```
 4
(4 rows)

-- для показанного массива получение индекса и обращение
-- к элементу по индексу выполняется с помощью подзапроса:
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- разворачивание двумерного массива:
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

Когда для функции в предложении `FROM` указывается `WITH ORDINALITY`, к столбцам результата функции добавляется столбец типа `bigint`, числа в котором начинаются с 1 и увеличиваются на 1 для каждой строки, выданной функцией. В первую очередь это полезно для функций, возвращающих множества, например, `unnest()`.

```
-- функция, возвращающая множество, с нумерацией
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls        | n
-----------------+----
 pg_serial       |  1
 pg_twophase     |  2
 postmaster.opts |  3
 pg_notify       |  4
 postgresql.conf |  5
 pg_tblspc       |  6
 logfile         |  7
 base            |  8
 postmaster.pid  |  9
```

```
 pg_ident.conf    | 10
 global           | 11
 pg_xact          | 12
 pg_snapshots     | 13
 pg_multixact     | 14
 PG_VERSION       | 15
 pg_wal           | 16
 pg_hba.conf      | 17
 pg_stat_tmp      | 18
 pg_subtrans      | 19
(19 rows)
```

9.27. Системные информационные функции и операторы

Функции, описанные в этом разделе, предназначены для получения различной информации о сервере Postgres Pro.

9.27.1. Функции получения информации о сеансе

В Таблице 9.69 перечислен ряд функций, предназначенных для получения информации о текущем сеансе и системе.

В дополнение к перечисленным здесь функциям существуют также функции, связанные с подсистемой статистики, которые тоже предоставляют системную информацию. Подробнее они рассматриваются в Подразделе 26.2.2.

Таблица 9.69. Функции получения информации о сеансе

Функция **Описание**

`current_catalog` → `name`
`current_database` () → `name`
Выдаёт имя текущей базы данных. (В стандарте SQL базы данных называются «каталогами», поэтому стандарту соответствует написание `current_catalog` .)

`current_query` () → `text` Выдаёт текст запроса, выполняемого в данный момент, в том виде, в каком его передал клиент (может состоять из нескольких операторов).
`current_role` → `name` Аналог `current_user` .

`current_schema` → `name`
`current_schema` () → `name`
Выдаёт имя первой схемы в пути поиска (или значение NULL, если путь поиска пустой). В эту схему будут добавляться таблицы и другие объекты, при создании которых схема не указывается явно.

`current_schemas` (*`include_implicit`* `boolean`) → `name[]` Выдаёт массив имён всех схем, в настоящее время представленных в действующем пути поиска, по порядку их приоритета. (Элементы текущего значения search_path, которым не соответствуют существующие схемы, опускаются). Если логический аргумент равен `true`, в результат включаются системные схемы, неявно просматриваемые при поиске, как например, `pg_catalog` .
`current_user` → `name` Выдаёт имя пользователя в текущем контексте выполнения.
`inet_client_addr` () → `inet`

Функция **Описание**
Выдаёт IP-адрес текущего клиента либо `NULL`, если текущее соединение установлено через Unix-сокет.
`inet_client_port` () → `integer` Выдаёт номер TCP-порта текущего клиента либо `NULL`, если текущее подключение установлено через Unix-сокет.
`inet_server_addr` () → `inet` Выдаёт IP-адрес, через который сервер принял текущее подключение, либо NULL, если текущее подключение установлено через Unix-сокет.
`inet_server_port` () → `integer` Выдаёт номер TCP-порта, через который сервер принял текущее подключение, либо `NULL`, если текущее подключение установлено через Unix-сокет.
`pg_backend_pid` () → `integer` Выдаёт идентификатор серверного процесса, обслуживающего текущий сеанс.
`pg_blocking_pids` (`integer`) → `integer[]` Выдаёт массив с идентификаторами процессов сеансов, которые блокируют серверный процесс с указанным идентификатором, либо пустой массив, если указанный серверный процесс не найден или не заблокирован. Один серверный процесс блокирует другой, если он либо удерживает блокировку, конфликтующую с блокировкой, запрашиваемой вторым (жёсткая блокировка), либо ожидает блокировку, которая вызвала бы конфликт с запросом блокировки заблокированного процесса и находится перед ней в очереди ожидания (мягкая блокировка). При распараллеливании запросов эта функция всегда выдаёт видимые клиентом идентификаторы процессов (то есть, результаты `pg_backend_pid`), даже если фактическая блокировка удерживается или ожидается дочерним рабочим процессом. Вследствие этого в результатах могут оказаться дублирующиеся PID. Также заметьте, что когда конфликтующую блокировку удерживает подготовленная транзакция, в выводе этой функции она будет представлена нулевым ID процесса. Частые вызовы этой функции могут отразиться на производительности базы данных, так как ей нужен монопольный доступ к общему состоянию менеджера блокировок, пусть и на короткое время.
`pg_conf_load_time` () → `timestamp with time zone` Выдаёт время, когда в последний раз сервер загружал файлы конфигурации. (Если текущий сеанс начался раньше, она возвращает время, когда эти файлы были перезагружены для данного сеанса, так что в разных сеансах это значение может немного различаться. В противном случае это будет время, когда файлы конфигурации считал главный процесс.)
`pg_current_logfile` ([`text`]) → `text` Выдаёт путь к файлу журнала, в настоящее время используемому сборщиком сообщений. Этот путь состоит из пути каталога log_directory и имени конкретного файла. Если сборщик сообщений отключён, выдаётся значение `NULL`. Если ведутся несколько журналов в разных форматах, при вызове функции `pg_current_logfile` без аргументов возвращается путь первого существующего файла следующего формата (в порядке приоритета): stderr, csvlog, `jsonlog`. Если файл журнала в этих форматах отсутствует, выдаётся `NULL`. Чтобы запросить информацию о файле в определённом формате, передайте csvlog, `jsonlog` или stderr в качестве значения необязательного параметра типа `text`. Если запрошенный формат не включён в log_destination, результатом будет `NULL`. Результат функции отражает содержимое файла `current_logfiles` . По умолчанию доступ к этой функции имеют только суперпользователи или роли с правами роли `pg_monitor` , но право на её выполнение (EXECUTE) можно дать и другим пользователям.

Функция **Описание**
`pg_my_temp_schema` () → `oid` Выдаёт OID временной схемы текущего сеанса или 0, если такой схемы нет (в рамках сеанса не создавались временные таблицы).
`pg_is_other_temp_schema` (`oid`) → `boolean` Выдаёт true, если заданный OID относится к временной схеме другого сеанса. (Это может быть полезно, например для исключения временных таблиц других сеансов из общего списка при просмотре таблиц базы данных.)
`pg_jit_available` () → `boolean` Выдаёт true, если в данном сеансе доступна JIT-компиляция (см. Главу 29) и параметр конфигурации jit включён.
`pg_listening_channels` () → `setof text` Выдаёт список имён каналов асинхронных уведомлений, по которым текущий сеанс принимает сигналы.
`pg_notification_queue_usage` () → `double precision` Выдаёт долю (0–1) максимального размера очереди асинхронных уведомлений, в настоящее время занятую уведомлениями, ожидающими обработки. За дополнительными сведениями обратитесь к LISTEN и NOTIFY.
`pg_postmaster_start_time` () → `timestamp with time zone` Выдаёт время, когда был запущен сервер.
`pg_safe_snapshot_blocking_pids` (`integer`) → `integer[]` Выдаёт массив с идентификаторами процессов сеансов, которые блокируют серверный процесс с указанным идентификатором, не давая ему получить безопасный снимок, либо выдаёт пустой массив, если такой серверный процесс не найден или он не заблокирован. Сеанс, выполняющий транзакцию уровня `SERIALIZABLE`, блокирует транзакцию `SERIALIZABLE READ ONLY DEFERRABLE`, не давая ей получить снимок, пока она не определит, что можно безопасно избежать установления предикатных блокировок. За дополнительными сведениями о сериализуемых и откладываемых транзакциях обратитесь к Подразделу 13.2.3. Частые вызовы этой функции могут отразиться на производительности сервера, так как ей нужен доступ к общему состоянию менеджера предикатных блокировок, пусть и на короткое время.
`pg_trigger_depth` () → `integer` Выдаёт текущий уровень вложенности в триггерах Postgres Pro (0, если эта функция вызывается не из тела триггера, непосредственно или косвенно).
`session_user` → `name` Выдаёт имя пользователя сеанса.
`system_user` → `text` Возвращает метод аутентификации и идентификационную информацию (если есть), которые пользователь представил во время цикла аутентификации до того, как ему была назначена роль базы данных. Он представлен как `auth_method:identity` или `NULL`, если пользователь не прошел аутентификацию (например, если была использована аутентификация trust).
`user` → `name` Аналог `current_user`.

Примечание

Функции `current_catalog`, `current_role`, `current_schema`, `current_user`, `session_user` и `user` имеют особый синтаксический статус в SQL: они должны вызываться без скобок после имени. Postgres Pro позволяет добавить скобки в вызове `current_schema`, но не других функций.

Функция `session_user` обычно возвращает имя пользователя, установившего текущее соединение с базой данных, но суперпользователи могут изменить это имя, выполнив команду SET SESSION AUTHORIZATION. Функция `current_user` возвращает идентификатор пользователя, по которому будут проверяться его права. Обычно это тот же пользователь, что и пользователь сеанса, но его можно сменить с помощью SET ROLE. Этот идентификатор также меняется при выполнении функций с атрибутом `SECURITY DEFINER`. На языке Unix пользователь сеанса называется «реальным», а текущий — «эффективным». Имена `current_role` и `user` являются синонимами `current_user`. (В стандарте SQL `current_role` и `current_user` имеют разное значение, но в Postgres Pro они не различаются, так как пользователи и роли объединены в единую сущность.)

9.27.2. Функции для проверки прав доступа

В Таблице 9.70 показаны функции, позволяющие программно проверить права доступа к объектам. (Подробнее о правах рассказывается в Разделе 5.8.) Этим функциям в качестве идентификатора пользователя, для которого запрашиваются права, можно передать его имя или OID (`pg_authid.oid`), а также можно передать `public` (это будет указывать на псевдороль PUBLIC). Кроме того, аргумент *`user`* можно полностью опустить, тогда будет подразумеваться `current_user`. Объект, права для доступа к которому запрашиваются, также можно задать по имени или по OID. Когда указывается имя объекта, его можно дополнить именем схемы, если это уместно. Запрашиваемое право записывается текстовой строкой, которая должна задавать одно из прав доступа, соответствующих типу объекта (например, `SELECT`). Дополнительно к названию права можно добавить `WITH GRANT OPTION` и проверить, разрешено ли пользователю передавать это право другим. Кроме того, в одном параметре можно перечислить несколько прав через запятую, и тогда функция возвратит `true`, если пользователь имеет какие-либо из них. (Регистр в названии прав не имеет значения, а между ними (но не внутри) разрешены пробельные символы.) Пара примеров:

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

Таблица 9.70. Функции для проверки прав доступа

Функция **Описание**
`has_any_column_privilege` ([*`user`* `name` или `oid`,] *`table`* `text` или `oid`, *`privilege`* `text`) → `boolean` Имеет ли пользователь указанное право для какого-либо столбца таблицы? Ответ положительный, когда он имеет это право для всей таблицы или ему дано соответствующее право на уровне столбцов хотя бы для одного столбца. Возможные права: `SELECT`, `INSERT`, `UPDATE` и `REFERENCES`.
`has_column_privilege` ([*`user`* `name` или `oid`,] *`table`* `text` или `oid`, *`column`* `text` или `smallint`, *`privilege`* `text`) → `boolean` Имеет ли пользователь указанное право для заданного столбца таблицы? Ответ положительный, когда он имеет это право для всей таблицы или ему дано соответствующее право на уровне столбца. Столбец можно задать по имени или номеру атрибута (`pg_attribute .attnum`). Возможные права: `SELECT`, `INSERT`, `UPDATE` и `REFERENCES`.
`has_database_privilege` ([*`user`* `name` или `oid`,] *`database`* `text` или `oid`, *`privilege`* `text`) → `boolean` Имеет ли пользователь указанное право в базе данных? Возможные права: `CREATE`, `CONNECT`, `TEMPORARY` и `TEMP` (синоним `TEMPORARY`).

Функция	**Описание**
`has_foreign_data_wrapper_privilege` ([*user* `name` **или** `oid`,] *fdw* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для обёртки сторонних данных? На данный момент возможно только одно право: `USAGE`.
`has_function_privilege` ([*user* `name` **или** `oid`,] *function* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для функции? Возможно единственное право: `EXECUTE`. При указании функции по имени, а не по OID, допускаются те же входные значения, что и для типа `regprocedure` (см. Раздел 8.19). Например: `SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');`
`has_language_privilege` ([*user* `name` **или** `oid`,] *language* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для языка? Возможно единственное право: `USAGE`.
`has_parameter_privilege` ([*user* `name` **или** `oid`,] *parameter* `text`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для параметра конфигурации? Регистр в имени параметра игнорируется. Возможные права: `SET` и `ALTER SYSTEM`.
`has_schema_privilege` ([*user* `name` **или** `oid`,] *schema* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для схемы? Возможные права: `CREATE` и `USAGE`.
`has_sequence_privilege` ([*user* `name` **или** `oid`,] *sequence* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для последовательности? Возможные права: `USAGE`, `SELECT` и `UPDATE`.
`has_server_privilege` ([*user* `name` **или** `oid`,] *server* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для стороннего сервера? На данный момент возможно только одно право: `USAGE`.
`has_table_privilege` ([*user* `name` **или** `oid`,] *table* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для таблицы? Возможные права: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER` и `MAINTAIN`.
`has_tablespace_privilege` ([*user* `name` **или** `oid`,] *tablespace* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для табличного пространства? Возможно единственное право: `CREATE`.
`has_type_privilege` ([*user* `name` **или** `oid`,] *type* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для типа данных? Возможно единственное право: `USAGE`. При указании типа по имени, а не по OID, допускаются те же входные значения, что и для типа данных `regtype` (см. Раздел 8.19).
`pg_has_role` ([*user* `name` **или** `oid`,] *role* `text` **или** `oid`, *privilege* `text`) → `boolean`	Имеет ли пользователь право для роли? Возможные права: `MEMBER`, `USAGE` и `SET`. Под `MEMBER` подразумевается непосредственное или косвенное членство в роли независимо от того, какие особые права могут быть предоставлены. `USAGE` означает, что пользователь получает права этой роли автоматически, не выполняя `SET ROLE`, а `SET` — можно ли изменить роль командой `SET ROLE`. К любому из этих типов прав можно добавить `WITH ADMIN OPTION` или `WITH GRANT OPTION`, чтобы проверить, имеется ли право `ADMIN` (все шесть вариантов написания проверяют одно и то же). Эта функция не поддерживает особый случай установки значения `public` для *user*, так как псевдороль PUBLIC не может быть членом обычных ролей.
`row_security_active` (*table* `text` **или** `oid`) → `boolean`	

Функция **Описание**
Действует ли защита на уровне строк для заданной таблицы в контексте текущего пользователя в текущем окружении?

В Таблице 9.71 показаны операторы, предназначенные для работы с типом `aclitem`, который представляет в системном каталоге права доступа. О содержании значений этого типа рассказывается в Разделе 5.8.

Таблица 9.71. Операторы для типа `aclitem`

Оператор **Описание** **Пример(ы)**

`aclitem` = `aclitem` → `boolean`
Значения `aclitem` равны? (Заметьте, что для типа `aclitem` не определён обычный набор операторов сравнения, поддерживается только проверка равенства. Массивы `aclitem`, в свою очередь, также могут сравниваться только на равенство.)
`'calvin=r*w/hobbes'::aclitem = 'calvin=r*w*/hobbes'::aclitem` → `f`

`aclitem[] @> aclitem` → `boolean`
Содержит ли массив заданное право? (Ответ положительный, если в массиве есть запись, соответствующая правообладателю и праводателю из `aclitem` и содержащая как минимум указанный набор прав.)
`'{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] @> 'calvin=r*/hobbes'::aclitem` → `t`

`aclitem[] ~ aclitem` → `boolean`
Это устаревший синоним для `@>`.
`'{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] ~ 'calvin=r*/hobbes'::aclitem` → `t`

В Таблице 9.72 приведены дополнительные функции, предназначенные для работы с типом `aclitem`.

Таблица 9.72. Функции для типа `aclitem`

Функция **Описание**
`acldefault` (*`type`* `"char"`, *`ownerId`* `oid`) → `aclitem[]` Выдаёт массив с элементами `aclitem`, содержащий действующий по умолчанию набор прав доступа для объектов типа *`type`*, принадлежащих роли *`ownerId`*. Результат показывает, какие права доступа подразумеваются, когда список ACL определённого объекта пуст. (Права доступа, действующие по умолчанию, описываются в Разделе 5.8.) Параметр *`type`* может принимать одно из следующих значений: 'c' —столбец (`COLUMN`), 'r' — таблица или подобный таблице объект (`TABLE`), 's' — последовательность (`SEQUENCE`), 'd' — база данных (`DATABASE`), 'f' — функция (`FUNCTION`) или процедура (`PROCEDURE`), 'l' — язык (`LANGUAGE`), 'L' — большой объект (`LARGE OBJECT`), 'n' — схема (`SCHEMA`), 'p' — параметр (`PARAMETER`), 't' — табличное пространство (`TABLESPACE`), 'F' — обёртка сторонних данных (`FOREIGN DATA WRAPPER`), 'S' — сторонний сервер (`FOREIGN SERVER`) и 'T' — тип (`TYPE`) или домен (`DOMAIN`).
`aclexplode` (`aclitem[]`) → `setof record` (*`grantor`* `oid`, *`grantee`* `oid`, *`privilege_type`* `text`, *`is_grantable`* `boolean`) Возвращает массив `aclitem` в виде набора табличных строк. Если праводателем является псевдороль PUBLIC, она представляется нулём в столбце *`grantee`*. Назначенные права представляются ключевыми словами `SELECT`, `INSERT` и т. д. (полный список находится в

Функция Описание
Таблице 5.1). Заметьте, что каждое право выводится в отдельной строке, так что в столбце *privilege_type* содержится только одно ключевое слово.
`makeaclitem` (*grantee* `oid`, *grantor* `oid`, *privileges* `text`, *is_grantable* `boolean`) → `aclitem` Конструирует значение `aclitem` с заданными свойствами. В аргументе *privileges* передаётся список имён прав, разделённых запятыми, таких как `SELECT`, `INSERT` и т. д., которые устанавливаются в результате. (Регистр строки права не имеет значения, дополнительные пробелы допускаются между именами прав, но не внутри них.)

9.27.3. Функции для определения видимости

В Таблице 9.73 показаны функции, определяющие *видимость* объекта с текущим путём поиска схем. К примеру, таблица считается видимой, если содержащая её схема включена в путь поиска и нет другой таблицы с тем же именем, которая была бы найдена в этом пути раньше. Другими словами, к этой таблице можно будет обратиться просто по её имени, без явного указания схемы. Просмотреть список всех видимых таблиц можно так:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Объект, представляющий функцию и оператор, считаются видимым в пути поиска, если при просмотре пути не находится предшествующий ему другой объект с тем же именем *и типами аргументов*. Для семейств и классов операторов помимо имени объекта во внимание принимается также связанный метод доступа к индексу.

Таблица 9.73. Функции для определения видимости

Функция Описание
`pg_collation_is_visible` (*collation* `oid`) → `boolean` Видимо ли правило сортировки в пути поиска?
`pg_conversion_is_visible` (*conversion* `oid`) → `boolean` Видимо ли преобразование в пути поиска?
`pg_function_is_visible` (*function* `oid`) → `boolean` Видима ли функция в пути поиска? (Также работает для процедур и агрегатных функций.)
`pg_opclass_is_visible` (*opclass* `oid`) → `boolean` Виден ли класс операторов в пути поиска?
`pg_operator_is_visible` (*operator* `oid`) → `boolean` Видим ли оператор в пути поиска?
`pg_opfamily_is_visible` (*opclass* `oid`) → `boolean` Видимо ли семейство операторов в пути поиска?
`pg_statistics_obj_is_visible` (*stat* `oid`) → `boolean` Видим ли объект статистики в пути поиска?
`pg_table_is_visible` (*table* `oid`) → `boolean` Видима ли таблица в пути поиска? (Эта проверка работает со всеми типами отношений, включая представления, матпредставления, индексы, последовательности и сторонние таблицы.)
`pg_ts_config_is_visible` (*config* `oid`) → `boolean` Видима ли конфигурация текстового поиска?
`pg_ts_dict_is_visible` (*dict* `oid`) → `boolean` Видим ли словарь текстового поиска?

Функция **Описание**
`pg_ts_parser_is_visible` (*`parser`* `oid`) → `boolean` Видим ли анализатор текстового поиска?
`pg_ts_template_is_visible` (*`template`* `oid`) → `boolean` Видим ли шаблон текстового поиска?
`pg_type_is_visible` (*`type`* `oid`) → `boolean` Видим ли тип (или домен) в пути поиска?

Всем этим функциям должен передаваться OID проверяемого объекта. Если вы хотите проверить объект по имени, удобнее использовать типы-псевдонимы OID (`regclass`, `regtype`, `regprocedure`, `regoperator`, `regconfig` или `regdictionary`), например:

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Заметьте, что проверять таким способом имена без указания схемы не имеет большого смысла — если имя удастся распознать, значит и объект будет видимым.

9.27.4. Функции для обращения к системным каталогам

В Таблице 9.74 перечислены функции, извлекающие информацию из системных каталогов.

Таблица 9.74. Функции для обращения к системным каталогам

Функция **Описание**
`format_type` (*`type`* `oid`, *`typemod`* `integer`) → `text` Выдаёт в формате SQL имя типа данных, определяемого по OID и, возможно, модификатору типа. Если модификатор неизвестен, вместо него можно передать NULL.

`pg_basetype` (`regtype`) → `regtype`
Возвращает OID базового типа домена, идентифицируемого его OID типа. Если в аргументе передаётся OID недоменного типа, он возвращается как есть. Возвращает NULL, если аргумент не является допустимым OID типа. Если есть цепочка доменных зависимостей, функция будет рекурсивно выполняться до тех пор, пока не будет найден базовый тип.
Предположим, для `CREATE DOMAIN mytext AS text`:
`pg_basetype('mytext'::regtype)` → `text`

`pg_char_to_encoding` (*`encoding`* `name`) → `integer` Преобразует предоставленное имя кодировки в целое число, представляющее внутренний идентификатор, используемый в некоторых таблицах системного каталога. Возвращает `-1`, если указано неизвестное имя кодировки.
`pg_encoding_to_char` (*`encoding`* `integer`) → `name` Преобразует целое число, используемое в качестве внутреннего идентификатора кодировки в некоторых таблицах системного каталога, в понятное имя. Возвращает пустую строку, если указан неправильный номер кодировки.
`pg_get_catalog_foreign_keys` () → `setof record` (*`fktable`* `regclass`, *`fkcols`* `text[]`, *`pktable`* `regclass`, *`pkcols`* `text[]`, *`is_array`* `boolean`, *`is_opt`* `boolean`) Возвращает набор записей, описывающих отношения внешних ключей, существующие в системных каталогах Postgres Pro. Столбец *`fktable`* содержит имя ссылающегося каталога, а столбец *`fkcols`* содержит имена ссылающихся столбцов. Точно так же столбец *`pktable`* содержит имя целевого каталога, а столбец *`pkcols`* содержит имена целевых столбцов. Если столбец *`is_array`* содержит значение true, последний ссылающийся столбец является массивом, каждый из элементов которого должен соответствовать некоторой записи в целевом каталоге. Если столбец *`is_opt`* содержит значение true, ссылающиеся столбцы могут содержать нули вместо действительной ссылки.

Функция **Описание**
`pg_get_constraintdef` (*constraint* `oid` [, *pretty* `boolean`]) → `text` Восстанавливает команду, создающую ограничение. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.)
`pg_get_expr` (*expr* `pg_node_tree` , *relation* `oid` [, *pretty* `boolean`]) → `text` Декомпилирует внутреннюю форму выражений, находящихся в системных каталогах, например, выражений, задающих значения по умолчанию. Если выражение может содержать Vars (переменные), передайте во втором параметре OID отношения, к которому обращается данное выражение; в противном случае достаточно передать 0.
`pg_get_functiondef` (*func* `oid`) → `text` Восстанавливает команду, создающую функцию или процедуру. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.) В результате выдаётся полноценный оператор `CREATE OR REPLACE FUNCTION` или `CREATE OR REPLACE PROCEDURE`.
`pg_get_function_arguments` (*func* `oid`) → `text` Восстанавливает список аргументов для функции или процедуры в том виде, в каком он должен задаваться в команде `CREATE FUNCTION` (включая значения по умолчанию).
`pg_get_function_identity_arguments` (*func* `oid`) → `text` Восстанавливает список аргументов, необходимый для однозначной идентификации функции или процедуры, в том виде, в каком он должен задаваться, например в `ALTER FUNCTION`. В этом виде значения по умолчанию не указываются.
`pg_get_function_result` (*func* `oid`) → `text` Восстанавливает для функции предложение `RETURNS` в том виде, в котором оно должно указываться в команде `CREATE FUNCTION`. Для процедуры возвращается `NULL`.
`pg_get_indexdef` (*index* `oid` [, *column* `integer`, *pretty* `boolean`]) → `text` Восстанавливает команду, создающую индекс. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.) Если передаётся значение *column* и оно отлично от нуля, восстанавливается только определение этого столбца.
`pg_get_keywords` () → `setof record` (*word* `text`, *catcode* `"char"`, *barelabel* `boolean`, *catdesc* `text`, *baredesc* `text`) Выдаёт набор записей, описывающих ключевые слова, которые воспринимает сервер. Столбец *word* содержит слово, а столбец *catcode* — код категории: `U` — не зарезервировано, `C` — слово, которое может быть именем столбца, `T` — слово, которое может быть именем типа или функции, `R` — полностью зарезервированное слово. Столбец *barelabel* содержит `true`, если ключевое слово можно использовать в качестве «открытой» метки столбца в списках `SELECT`, или `false`, если его можно использовать только после `AS`. Столбец `catdesc` содержит описание категории слова, возможно локализованное. Столбец *baredesc* содержит возможно локализованную строку, описывающую статус метки столбца, содержащего слово.
`pg_get_partkeydef` (*table* `oid`) → `text` Восстанавливает определение ключа секционирования секционированной таблицы в той форме, которую он имел бы в предложении `PARTITION BY` команды `CREATE TABLE`. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.)
`pg_get_ruledef` (*rule* `oid` [, *pretty* `boolean`]) → `text` Восстанавливает команду, создающую правило. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.)
`pg_get_serial_sequence` (*table* `text`, *column* `text`) → `text` Выдаёт имя последовательности, связанной со столбцом, либо NULL, если такой последовательности нет. Для столбца идентификации это будет последовательность, созданная для него неявно внутри. Для столбцов, имеющих один из последовательных типов (

Функция	**Описание**
	`serial`, `smallserial`, `bigserial`), это последовательность, созданная объявлением данного столбца. В последнем случае эту связь можно изменить или разорвать, воспользовавшись командой `ALTER SEQUENCE OWNED BY`. (Возможно, эту функцию стоило назвать `pg_get_owned_sequence`; её существующее имя отражает тот факт, что изначально она использовалась со столбцами последовательных типов.) В первом параметре функции указывается имя таблицы (возможно, дополненное схемой), а во втором — имя столбца. Так как первый параметр может содержать имя схемы и таблицы, он воспринимается не как идентификатор в кавычках и поэтому по умолчанию приводится к нижнему регистру, тогда как имя столбца во втором параметре воспринимается как заключённое в кавычки и в нём регистр символов сохраняется. Эта функция возвращает имя в виде, пригодном для передачи функциям, работающим с последовательностями (см. Раздел 9.17). Обычно эта функция применяется для получения текущего значения последовательности для столбца идентификации или последовательного столбца, например так: `SELECT currval(pg_get_serial_sequence('sometable', 'id'));`
`pg_get_statisticsobjdef` (*statobj* `oid`) → `text`	Восстанавливает команду, создающую объект расширенной статистики. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.)
`pg_get_triggerdef` (*trigger* `oid` [, *pretty* `boolean`]) → `text`	Восстанавливает команду, создающую триггер. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.)
`pg_get_userbyid` (*role* `oid`) → `name`	Выдаёт имя роли по заданному OID.
`pg_get_viewdef` (*view* `oid` [, *pretty* `boolean`]) → `text`	Восстанавливает команду `SELECT`, определяющую представление или матпредставление. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.)
`pg_get_viewdef` (*view* `oid`, *wrap_column* `integer`) → `text`	Восстанавливает команду `SELECT`, определяющую представление или матпредставление. (Текст команды не является изначальным, он восстанавливается путём декомпиляции.) В этой вариации функции всегда применяется визуальное улучшение и длинные строки переносятся, чтобы текст умещался по ширине в заданное число столбцов.
`pg_get_viewdef` (*view* `text` [, *pretty* `boolean`]) → `text`	Восстанавливает команду `SELECT`, определяющую представление или матпредставление, заданное по имени, а не по OID. (Это устаревшая вариация, используйте вместо неё вариацию с OID).
`pg_index_column_has_property` (*index* `regclass`, *column* `integer`, *property* `text`) → `boolean`	Проверяет, имеет ли столбец индекса заданное свойство. Типичные свойства столбцов описаны в Таблице 9.75. (Заметьте, что расширенные методы доступа могут определять для своих индексов дополнительные свойства.) Если заданное имя свойства не известно или не применимо к конкретному объекту, либо если OID или номер столбца не указывают на существующий объект, эта функция возвращает `NULL`.
`pg_index_has_property` (*index* `regclass`, *property* `text`) → `boolean`	Проверяет, имеет ли индекс заданное свойство. Типичные свойства индексов описаны в Таблице 9.75. (Заметьте, что расширенные методы доступа могут определять для своих индексов дополнительные свойства.) Если заданное имя свойства не известно или не применимо к конкретному объекту, либо если OID не указывает на существующий объект, эта функция возвращает `NULL`.
`pg_indexam_has_property` (*am* `oid`, *property* `text`) → `boolean`	

Функция **Описание**
Проверяет, имеет ли метод доступа индекса заданное свойство. Свойства методов доступа описаны в Таблице 9.77. Если заданное имя свойства неизвестно или не применимо к конкретному объекту, либо если OID не указывает на существующий объект, эта функция возвращает `NULL`.
`pg_options_to_table` (*options_array* `text[]`) → `setof record` (*option_name* `text`, *option_value* `text`) Выдаёт набор параметров хранилища, представленных значением из `pg_class.reloptions` или `pg_attribute.attoptions`.
`pg_settings_get_flags` (*guc* `text`) → `text[]` Возвращает массив флагов, связанных с данным параметром GUC, или `NULL`, если указанный параметр не существует. Если же он существует, но у него нет флагов для отображения, возвращается пустой массив. Показываются только самые полезные флаги, перечисленные в Таблице 9.78.
`pg_tablespace_databases` (*tablespace* `oid`) → `setof oid` Выдаёт набор OID баз данных, объекты которых размещены в заданном табличном пространстве. Если эта функция возвращает строки, это означает, что табличное пространство не пустое и удалить его нельзя. Какие именно объекты находятся в табличном пространстве, можно определить, подключаясь к базам данных, OID которых сообщила `pg_tablespace_databases`, и анализируя их каталоги `pg_class`.
`pg_tablespace_location` (*tablespace* `oid`) → `text` Выдаёт путь в файловой системе к местоположению заданного табличного пространства.

`pg_typeof` (`"any"`) → `regtype`
Выдаёт OID типа данных для переданного значения. Это может быть полезно для разрешения проблем или динамического создания SQL-запросов. Эта функция объявлена как возвращающая тип `regtype`, который является псевдонимом типа OID (см. Раздел 8.19); это означает, что значение этого типа можно сравнивать как OID, но выводится оно как название типа.
`pg_typeof(33)` → `integer`

`COLLATION FOR` (`"any"`) → `text`
Выдаёт имя правила сортировки для переданного значения. Это значение может быть заключено в кавычки и дополнено схемой. Если с выражением аргумента не связано правило сортировки, возвращается `NULL`. Если же правила сортировки не применимы для типа аргумента, происходит ошибка.
`collation for ('foo'::text)` → `"default"`
`collation for ('foo' COLLATE "de_DE")` → `"de_DE"`

`to_regclass` (`text`) → `regclass` Переводит в OID текстовое имя отношения. Подобный результат можно получить, приведя строку с именем к типу `regclass` (см. Раздел 8.19); однако если имя не будет распознано, эта функция возвратит `NULL`, а не выдаст ошибку.
`to_regcollation` (`text`) → `regcollation` Переводит в OID текстовое имя правила сортировки. Подобный результат можно получить, приведя строку с именем к типу `regcollation` (см. Раздел 8.19); однако если имя не будет распознано, эта функция возвратит `NULL`, а не выдаст ошибку.
`to_regnamespace` (`text`) → `regnamespace` Переводит в OID текстовое имя схемы. Подобный результат можно получить, приведя строку с именем к типу `regnamespace` (см. Раздел 8.19); однако если имя не будет распознано, эта функция возвратит `NULL`, а не выдаст ошибку.

Функция **Описание**
`to_regoper ( text ) → regoper` Переводит в OID текстовое имя оператора. Подобный результат можно получить, приведя строку с именем к типу `regoper` (см. Раздел 8.19); однако если имя не будет распознано или окажется неоднозначным, эта функция возвратит `NULL`, а не выдаст ошибку.
`to_regoperator ( text ) → regoperator` Переводит в OID текстовое имя оператора с указанием типов параметров. Подобный результат можно получить, приведя строку с именем к типу `regoperator` (см Раздел 8.19); однако если имя не будет распознано, эта функция возвратит `NULL`, а не выдаст ошибку.
`to_regproc ( text ) → regproc` Переводит в OID текстовое имя функции или процедуры. Подобный результат можно получить, приведя строку с именем к типу `regoperator` (см. Раздел 8.19); однако если имя не будет распознано или окажется неоднозначным, эта функция возвратит `NULL`, а не выдаст ошибку.
`to_regprocedure ( text ) → regprocedure` Переводит в OID текстовое имя функции или процедуры с указанием типов аргументов. Подобный результат можно получить, приведя строку с именем к типу `regprocedure` (см. Раздел 8.19); однако если имя не будет распознано, эта функция возвратит `NULL`, а не выдаст ошибку.
`to_regprofile ( text ) → regprofile` Переводит в OID текстовое имя профиля. Подобный результат можно получить, приведя строку с именем к типу `regprofile` (см. Раздел 8.19); однако если имя не будет распознано, эта функция возвратит `NULL`, а не выдаст ошибку.
`to_regrole ( text ) → regrole` Переводит в OID текстовое имя роли. Подобный результат можно получить, приведя строку с именем к типу `regrole` (см. Раздел 8.19); однако если имя не будет распознано, эта функция возвратит `NULL`, а не выдаст ошибку.
`to_regtype ( text ) → regtype` Анализирует строку текста, извлекает из неё потенциальное имя типа и переводит это имя в OID типа. Синтаксическая ошибка в строке приведёт к ошибке, но если строка является синтаксически допустимым именем типа, которое не найдено в каталогах, результатом будет `NULL`. Похожий результат получается при приведении строки к типу `regtype` (см. Раздел 8.19), только при этом будет выведена ошибка о том, что имя не было найдено.

`to_regtypemod ( text ) → integer`
Разбирает строку текста, извлекает из неё потенциальное имя типа и переводит модификатор типа при его наличии. Синтаксическая ошибка в строке приведёт к ошибке, но если строка является синтаксически допустимым именем типа, которое не найдено в каталогах, результатом будет `NULL`. Если модификатор типа отсутствует, результатом будет `-1`.
Функцию `to_regtypemod` можно использовать совместно с функцией to_regtype для создания соответствующих входных данных для format_type, что позволяет нормализовать строку, представляющую имя типа.
`format_type(to_regtype('varchar(32)'), to_regtypemod('varchar(32)'))` → `character varying(32)`

Многие из функций, восстанавливающих (декомпилирующих) объекты БД, принимают дополнительный флаг *pretty*, в котором можно передать `true` для визуального улучшения вывода. При этом будут убраны лишние скобки и добавлены пробельные символы, чтобы текст был более разборчивым. Преобразованный вывод легче читается, но формат по умолчанию с большей вероятностью будет успешно восприниматься будущими версиями Postgres Pro, поэтому для выгрузки содержимого БД рекомендуется избегать использования визуально улучшенного формата. Если параметр *`pretty`* равен `false`, это равнозначно его отсутствию.

Таблица 9.75. Свойства столбца индекса

Имя	Описание
`asc`	Сортируется ли столбец по возрастанию при сканировании вперёд?
`desc`	Сортируется ли столбец по убыванию при сканировании вперёд?
`nulls_first`	Выдаются ли NULL в начале при сканирования вперёд?
`nulls_last`	Выдаются ли NULL в конце при сканировании вперёд?
`orderable`	Связан ли со столбцом некоторый порядок сортировки?
`distance_orderable`	Может ли столбец сканироваться по порядку оператором «расстояния», например, `ORDER BY столбец <-> константа` ?
`returnable`	Может ли значение столбца быть получено при сканировании только индекса?
`search_array`	Поддерживает ли столбец внутренними средствами поиск `столбец = ANY(массив)` ?
`search_nulls`	Поддерживает ли столбец поиск `IS NULL` и `IS NOT NULL`?

Таблица 9.76. Свойства индекса

Имя	Описание
`clusterable`	Может ли индекс использоваться в команде `CLUSTER`?
`index_scan`	Поддерживает ли индекс простое сканирование (не по битовой карте)?
`bitmap_scan`	Поддерживает ли индекс сканирование по битовой карте?
`backward_scan`	Может ли в процессе сканирования меняться направление (для поддержки перемещения курсора `FETCH BACKWARD` без необходимости материализации)?

Таблица 9.77. Свойства метода доступа индекса

Имя	Описание
`can_order`	Поддерживает ли метод доступа `ASC`, `DESC` и связанные ключевые слова в `CREATE INDEX`?
`can_unique`	Поддерживает ли метод доступа уникальные индексы?
`can_multi_col`	Поддерживает ли метод доступа индексы по нескольким столбцам?
`can_exclude`	Поддерживает ли метод доступа ограничения-исключения?

Имя	**Описание**
`can_include`	Поддерживает ли метод доступа предложение `INCLUDE` команды `CREATE INDEX`?

Таблица 9.78. Флаги параметров GUC

Флаг	**Описание**
`EXPLAIN`	Параметры с этим флагом включаются в команды `EXPLAIN (SETTINGS)` .
`NO_SHOW_ALL`	Параметры с этим флагом исключаются из команд `SHOW ALL`.
`NO_RESET`	Параметры с этим флагом не поддерживают команды `RESET`.
`NO_RESET_ALL`	Параметры с этим флагом исключаются из команд `RESET ALL`.
`NOT_IN_SAMPLE`	Параметры с этим флагом не включаются в `postgresql.conf` по умолчанию.
`RUNTIME_COMPUTED`	Параметры с этим флагом вычисляются во время выполнения.

9.27.5. Функции получения информации и адресации объектов

В Таблице 9.79 перечислены функции, связанные с идентификацией и адресацией объектов баз данных.

Таблица 9.79. Функции получения информации и адресации объектов

Функция **Описание**
`pg_describe_object` (*`classid`* `oid`, *`objid`* `oid`, *`objsubid`* `integer`) → `text` Выдаёт текстовое описание объекта БД, идентифицируемого по OID каталога, OID объекта и ID подобъекта (например, номер столбца в таблице; ID подобъекта равен нулю для объекта в целом). Это описание предназначено для человека и может переводиться, в зависимости от конфигурации сервера. Данная функция в особенности полезна, когда требуется узнать, что за объект хранится в каталоге `pg_depend` . Эта функция возвращает значение `NULL` для неопределённых объектов.
`pg_identify_object` (*`classid`* `oid`, *`objid`* `oid`, *`objsubid`* `integer`) → `record` (*`type`* `text`, *`schema`* `text`, *`name`* `text`, *`identity`* `text`) Выдаёт запись, содержащую достаточно информации для однозначной идентификации объекта БД по OID каталога, OID объекта и ID подобъекта. Эта информация предназначена для машины и поэтому никогда не переводится. Столбец *`type`* содержит тип объекта БД; *`schema`* — имя схемы, к которой относится объект (либо `NULL` для объектов, не относящихся к схемам); *`name`* — имя объекта, при необходимости в кавычках, которое присутствует, только если оно (возможно, вместе со схемой) однозначно идентифицирует объект (в противном случае `NULL`); *`identity`* — полный идентификатор объекта, точный формат которого зависит от типа объекта, а каждая его часть дополняется схемой и заключается в кавычки, если требуется. Для неопределённых объектов эти столбцы содержат значение `NULL`.
`pg_identify_object_as_address` (*`classid`* `oid`, *`objid`* `oid`, *`objsubid`* `integer`) → `record` (*`type`* `text`, *`object_names`* `text[]`, *`object_args`* `text[]`) Выдаёт запись, содержащую достаточно информации для однозначной идентификации объекта БД по OID каталога, OID объекта и ID подобъекта. Выдаваемая информация не зависит от текущего сервера, то есть по ней можно идентифицировать одноимённый объект на другом сервере. Поле *`type`* содержит тип объекта БД, а *`object_names`* и *`object_args`* — текстовые массивы, в совокупности формирующие ссылку на объект.

Функция **Описание**
Эти три значения можно передать функции `pg_get_object_address`, чтобы получить внутренний адрес объекта.
`pg_get_object_address` (*type* `text`, *object_names* `text[]`, *object_args* `text[]`) → `record` (*classid* `oid`, *objid* `oid`, *objsubid* `integer`) Выдаёт запись, содержащую достаточно информации для уникальной идентификации объекта БД по коду типа и массивам имён и аргументов. В ней возвращаются значения, которые используются в системных каталогах, например `pg_depend`, и могут передаваться в другие системные функции, например `pg_describe_object` или `pg_identify_object`. Поле *classid* содержит OID системного каталога, к которому относится объект; *objid* — OID самого объекта, а *objsubid* — идентификатор подобъекта или 0 в случае его отсутствия. Эта функция является обратной к `pg_identify_object_as_address`. Для неопределённых объектов эти поля содержат значение `NULL`.

9.27.6. Функции получения комментариев

Функции, перечисленные в Таблице 9.80, извлекают комментарии, заданные для объектов с помощью команды COMMENT. Если найти комментарий для заданных параметров не удаётся, они возвращают NULL.

Таблица 9.80. Функции получения комментариев

Функция **Описание**
`col_description` (*table* `oid`, *column* `integer`) → `text` Выдаёт комментарий для столбца с заданным номером в таблице с указанным OID. (`obj_description` нельзя использовать для столбцов таблицы, так столбцы не имеют собственных OID.)
`obj_description` (*object* `oid`, *catalog* `name`) → `text` Выдаёт комментарий для объекта, имеющего заданный OID и находящегося в указанном системном каталоге. Например, в результате вызова `obj_description(123456,'pg_class')` будет получен комментарий для таблицы с OID 123456.
`obj_description` (*object* `oid`) → `text` Выдаёт комментарий для объекта, заданного только по OID. Эта вариация считается *устаревшей*, так как нет гарантии, что OID будет уникальным в разных системных каталогах, и поэтому может быть получен неправильный комментарий.
`shobj_description` (*object* `oid`, *catalog* `name`) → `text` Выдаёт комментарий для общего объекта баз данных, имеющего заданный OID и находящегося в указанном системном каталоге. Эта функция аналогична `obj_description`, за исключением того, что она извлекает комментарий для общих объектов (то есть, баз данных, ролей и табличных пространств). Некоторые системные каталоги являются глобальными для всех баз данных в кластере и описания объектов в них также хранятся глобально.

9.27.7. Функции проверки допустимости данных

Функции, показанные в Таблица 9.81, могут быть полезны для проверки допустимости предлагаемых входных данных.

Таблица 9.81. Функции проверки допустимости данных

Функция **Описание** **Пример(ы)**
`pg_input_is_valid` (*string* `text`, *type* `text`) → `boolean`

<table>
<tr><th>Функция
Описание
Пример(ы)</th></tr>
<tr><td>Проверяет, является ли заданный аргумент <i>string</i> допустимым входным значением для указанного типа данных, и возвращает true или false.
Эта функция будет работать должным образом только в том случае, если функция ввода типа данных была изменена, чтобы сообщать о недопустимом вводе как о «мягкой» ошибке. В противном случае передача недопустимых входных данных приведёт к отмене транзакции, как если бы строка была непосредственно приведена к типу.
<code>pg_input_is_valid('42', 'integer')</code> → <code>t</code>
<code>pg_input_is_valid('42000000000', 'integer')</code> → <code>f</code>
<code>pg_input_is_valid('1234.567', 'numeric(7,4)')</code> → <code>f</code></td></tr>
<tr><td><code>pg_input_error_info</code> (<i>string</i> <code>text</code>, <i>type</i> <code>text</code>) → <code>record</code> (<i>message</i> <code>text</code>, <i>detail</i> <code>text</code>, <i>hint</i> <code>text</code>, <i>sql_error_code</i> <code>text</code>)
Проверяет, является ли заданный аргумент <i>string</i> допустимым входным значением для указанного типа данных; если нет, возвращает подробную информацию о ошибке, которая возникла бы в таком случае. Если ввод допустим, выдаётся NULL. Принимаются те же входные значения, что и для <code>pg_input_is_valid</code>.
Эта функция будет работать должным образом только в том случае, если функция ввода типа данных была изменена, чтобы сообщать о недопустимом вводе как о «мягкой» ошибке. В противном случае передача недопустимых входных данных приведёт к отмене транзакции, как если бы строка была непосредственно приведена к типу.
<code>SELECT * FROM pg_input_error_info('42000000000', 'integer')</code> →
<pre>
                       message | detail | hint | sql_error_code
--+--------+------+----------------
 value "42000000000" is out of range for type integer | | | 22003
</pre></td></tr>
</table>

9.27.8. Функции получения информации об идентификаторах транзакций и снимках состояния

Функции, показанные в Таблице 9.82, выдают информацию о транзакциях сервера в виде, подходящем для экспорта. В основном эти функции используются, чтобы определить, какие транзакции были зафиксированы между двумя снимками состояния.

Таблица 9.82. Функции получения информации об идентификаторах транзакций и снимках состояния

Функция Описание
`age` (`xid`) → `integer` Возвращает количество транзакций между указанным идентификатором транзакции и текущим значением счётчика транзакций.
`mxid_age` (`xid`) → `integer` Возвращает количество идентификаторов мультитранзакций между указанным идентификатором мультитранзакций и текущим значением счётчика мультитранзакций.
`pg_current_xact_id` () → `xid8` Выдаёт идентификатор текущей транзакции. Если у текущей транзакции ещё нет идентификатора (она не успела выполнить какие-либо изменения в базе), он будет ей назначен; подробнее см. Раздел 64.1. Если выполняется в подтранзакции, выдаёт идентификатор транзакции верхнего уровня; подробнее см. Раздел 64.3.
`pg_current_xact_id_if_assigned` () → `xid8`

Функция **Описание**
Выдаёт идентификатор текущей транзакции или `NULL`, если она ещё не имеет идентификатора. (Если транзакция может оставаться только читающей, лучше использовать эту вариацию, чтобы избежать излишнего потребления XID.) Если выполняется в подтранзакции, возвращает идентификатор транзакции верхнего уровня.
`pg_xact_status` (`xid8`) → `text` Выдаёт состояние фиксации транзакции. Результатом будет одно из значений: `in progress` (выполняется), `committed` (зафиксирована) и `aborted` (прервана), если только транзакция не настолько давняя, чтобы система удалила информацию о её состоянии. Если же упоминаний о транзакции в системе не осталось и информация о её фиксации потеряна, эта функция возвращает `NULL`. Приложения могут использовать эту функцию, например, чтобы определить, была ли транзакция зафиксирована или прервана, после потери подключения к серверу в процессе выполнения `COMMIT`. Заметьте, что для подготовленных транзакций возвращается состояние `in progress`; приложения должны проверять `pg_prepared_xacts`, если им нужно определить, является ли транзакция с заданным ID подготовленной.
`pg_current_snapshot` () → `pg_snapshot` Возвращает текущий *снимок состояния* — структуру данных, показывающую идентификаторы транзакций, которые выполняются в момент снимка. В снимок включаются только идентификаторы транзакций верхнего уровня; идентификаторы подтранзакций не отображаются; за подробностями обратитесь к Разделу 64.3.
`pg_snapshot_xip` (`pg_snapshot`) → `setof xid8` Выдаёт набор идентификаторов выполняющихся транзакций, содержащихся в снимке.
`pg_snapshot_xmax` (`pg_snapshot`) → `xid8` Выдаёт значение `xmax` для заданного снимка.
`pg_snapshot_xmin` (`pg_snapshot`) → `xid8` Выдаёт `xmin` для заданного снимка.
`pg_visible_in_snapshot` (`xid8`, `pg_snapshot`) → `boolean` *Видна* ли транзакция с указанным идентификатором в данном снимке (то есть, была ли она завершена до момента получения снимка)? Обратите внимание, что эта функция не выдаст правильный ответ, если ей передать идентификатор подтранзакции (subxid); за подробностями обратитесь к Разделу 64.3.
`pg_get_multixact_members` (*`multixid`* `xid`) → `setof record` (*`xid`* `xid`, *`mode`* `text`) Выдаёт идентификатор транзакции и режима блокировки для каждого члена с заданным идентификатором мультитранзакции. Режимы блокировок `forupd`, `fornokeyupd`, `sh` и `keysh` соответствуют блокировкам уровня строк `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` и `FOR KEY SHARE` соответственно, более подробная информация предоставлена в Подразделе 13.3.2. Ещё два режима относятся только к мультитранзакциям: `nokeyupd`, используется для обновлений, не меняющий ключевые столбцы, и `upd`, используется для обновлений и удалений, которые затрагивают ключевые столбцы.

Внутренний тип идентификаторов транзакций, `xid`, имеет размер 32 бита и значения в нём повторяются через каждые 4 миллиарда транзакций. Однако функции, показанные в Таблице 9.82, за исключением `age`, `mxid_age` и `pg_get_multixact_members`, используют 64-битный тип `xid8`, значения которого не повторяются на протяжении всей жизни сервера и могут быть приведены к типу `xid`, если требуется (подробное описание находится в Разделе 64.1). Тип данных `pg_snapshot` содержит информацию о видимости транзакций в определённый момент времени. Его компоненты описаны в Таблице 9.83. В текстовом виде `pg_snapshot` представляется как *`xmin`*`:`*`xmax`*`:`*`список_xip`*. Например, запись `10:20:10,14,15` обозначает `xmin=10, xmax=20, xip_list=10, 14, 15`.

Таблица 9.83. Состав информации о снимке

Имя	Описание
`xmin`	Минимальный идентификатор транзакции среди всех активных. Все транзакции, идентификаторы которых меньше `xmin`, уже либо зафиксированы и видимы, либо отменены и «мертвы».
`xmax`	Идентификатор на один больше идентификатора последней завершённой транзакции. Все транзакции, идентификаторы которых больше или равны `xmax`, на момент получения снимка ещё не завершены и не являются видимыми.
`xip_list`	Транзакции, выполняющиеся в момент получения снимка. Транзакции с такими идентификаторами, для которых `xmin <= ид < xmax`, не попавшие в этот список, были уже завершены на момент получения снимка, и поэтому либо видимы, либо «мертвы», в зависимости от варианта завершения. Идентификаторы подтранзакций (subxid) в этот список не включаются.

До 13 версии в Postgres Pro отсутствовал тип `xid8`, поэтому предлагались вариации этих функций, в которых 64-битные XID представлялись в типе `bigint`, а для информации о снимке использовался отдельный тип `txid_snapshot`. В названиях этих старых функций фигурирует `txid`. Они по-прежнему поддерживаются для обратной совместимости, но могут быть удалены в будущем. См. Таблицу 9.84.

Таблица 9.84. Устаревшие функции получения информации об идентификаторах транзакций и снимках состояния

Функция Описание
`txid_current` () → `bigint` См. `pg_current_xact_id()` .
`txid_current_if_assigned` () → `bigint` См. `pg_current_xact_id_if_assigned()` .
`txid_current_snapshot` () → `txid_snapshot` См. `pg_current_snapshot()` .
`txid_snapshot_xip` (`txid_snapshot`) → `setof bigint` См. `pg_snapshot_xip()` .
`txid_snapshot_xmax` (`txid_snapshot`) → `bigint` См. `pg_snapshot_xmax()` .
`txid_snapshot_xmin` (`txid_snapshot`) → `bigint` См. `pg_snapshot_xmin()` .
`txid_visible_in_snapshot` (`bigint`, `txid_snapshot`) → `boolean` См. `pg_visible_in_snapshot()` .
`txid_status` (`bigint`) → `text` См. `pg_xact_status()` .

9.27.9. Функции получения информации о фиксации транзакций

Функции, показанные в Таблице 9.85, выдают информацию о времени фиксации уже завершённых транзакций. Результат этих функций будет полезным, только когда включён параметр конфигурации track_commit_timestamp и только для транзакций, зафиксированных после его включения. Информация о временной метке фиксации удаляется во время очистки.

Таблица 9.85. Функции получения информации о фиксации транзакций

Функция **Описание**
`pg_xact_commit_timestamp` (`xid`) → `timestamp with time zone` Выдаёт время фиксации транзакции.
`pg_xact_commit_timestamp_origin` (`xid`) → `record` (*`timestamp`* `timestamp with time zone`, *`roident`* `oid`) Выдаёт время фиксации и источник репликации транзакции.
`pg_last_committed_xact` () → `record` (*`xid`* `xid`, *`timestamp`* `timestamp with time zone`, *`roident`* `oid`) Выдаёт идентификатор, время фиксации и источник репликации транзакции, зафиксированной последней.

9.27.10. Функции управления данными

Функции, показанные в Таблице 9.86, выдают информацию, записываемую во время `initdb`, например, версию каталога. Они также выводят сведения о журнале предзаписи и контрольных точках. Эта информация относится ко всему кластеру, а не к отдельной базе. Данные функции выдают практически ту же информацию, и из того же источника, что и pg_controldata.

Таблица 9.86. Функции управления данными

Функция **Описание**
`pg_control_checkpoint` () → `record` Выдаёт информацию о текущем состоянии контрольных точек, показанную в Таблице 9.87.
`pg_control_system` () → `record` Выдаёт информацию о текущем состоянии управляющего файла, показанную в Таблице 9.88.
`pg_control_init` () → `record` Выдаёт информацию о состоянии инициализации кластера, показанную в Таблице 9.89.
`pg_control_recovery` () → `record` Выдаёт информацию о состоянии восстановления, как показано в Таблице 9.90.

Таблица 9.87. Столбцы результата `pg_control_checkpoint`

Имя столбца	**Тип данных**
`checkpoint_lsn`	`pg_lsn`
`redo_lsn`	`pg_lsn`
`redo_wal_file`	`text`
`timeline_id`	`integer`
`prev_timeline_id`	`integer`
`full_page_writes`	`boolean`
`next_xid`	`text`
`next_oid`	`oid`
`next_multixact_id`	`xid`

Имя столбца	Тип данных
`next_multi_offset`	`xid`
`oldest_xid`	`xid`
`oldest_xid_dbid`	`oid`
`oldest_active_xid`	`xid`
`oldest_multi_xid`	`xid`
`oldest_multi_dbid`	`oid`
`oldest_commit_ts_xid`	`xid`
`newest_commit_ts_xid`	`xid`
`checkpoint_time`	`timestamp with time zone`

Таблица 9.88. Столбцы результата `pg_control_system`

Имя столбца	Тип данных
`pg_control_version`	`integer`
`catalog_version_no`	`integer`
`system_identifier`	`bigint`
`pg_control_last_modified`	`timestamp with time zone`
`pg_control_edition`	`text`

Таблица 9.89. Столбцы результата `pg_control_init`

Имя столбца	Тип данных
`max_data_alignment`	`integer`
`database_block_size`	`integer`
`blocks_per_segment`	`integer`
`wal_block_size`	`integer`
`bytes_per_wal_segment`	`integer`
`max_identifier_length`	`integer`
`max_index_columns`	`integer`
`max_toast_chunk_size`	`integer`
`large_object_chunk_size`	`integer`
`float8_pass_by_value`	`boolean`
`data_page_checksum_version`	`integer`

Таблица 9.90. Столбцы результата `pg_control_recovery`

Имя столбца	Тип данных
`min_recovery_end_lsn`	`pg_lsn`
`min_recovery_end_timeline`	`integer`
`backup_start_lsn`	`pg_lsn`
`backup_end_lsn`	`pg_lsn`
`end_of_backup_record_required`	`boolean`

9.27.11. Функции получения информации о версии

Функции, перечисленные в Таблице 9.91, выводят информацию о версии.

Таблица 9.91. Функции получения информации о версии

Функция Описание
`version` () → `text` Выдаёт текстовую строку, описывающую версию сервера PostgreSQL. Эту информацию также можно получить из переменной server_version или, в более машинно-ориентированном формате, из переменной server_version_num. При разработке программ следует использовать функцию `server_version_num` (появившуюся в версии 8.2) или `PQserverVersion`, а не разбирать текстовую версию.
`unicode_version` () → `text` Возвращает строку с версией Unicode, используемой Postgres Pro.
`icu_unicode_version` () → `text` Возвращает строку с версией Unicode, используемой ICU, если сервер был создан с поддержкой ICU, и `NULL` в противном случае
`pgpro_version` () → `text` Выдаёт текстовую строку, описывающую версию сервера Postgres Pro. Эта функция устарела и будет удалена в будущих выпусках. Вместо неё используйте параметр конфигурации pgpro_version.
`pgpro_edition` () → `text` Выдаёт текстовую строку, описывающую редакцию Postgres Pro, например `standard` или `enterprise`. Эта функция устарела и будет удалена в будущих выпусках. Вместо неё используйте параметр конфигурации pgpro_edition.
`pgpro_build` () → `text` Выдаёт идентификатор состояния исходного кода, из которого скомпилирован Postgres Pro. Эта функция устарела и будет удалена в будущих выпусках. Вместо неё используйте параметр конфигурации pgpro_build.

9.27.12. Функции получения информации о сводках WAL

Функции, перечисленные в Таблице 9.92, выводят информацию о создании сводок WAL. См. summarize_wal.

Таблица 9.92. Функции получения информации о сводках WAL

Функция Описание
`pg_available_wal_summaries` () → `setof record` (*`tli`* `bigint`, *`start_lsn`* `pg_lsn` , *`end_lsn`* `pg_lsn`) Выдаёт информацию о файлах сводок WAL, имеющихся в каталоге данных `pg_wal/summaries`. Функция возвращает по одной строке на каждый файл сводки WAL. Каждый файл содержит сводку WAL по указанному TLI в указанном диапазоне LSN. Эта функция может быть полезна для определения того, достаточно ли на сервере сводок WAL для создания инкрементальной резервной копии на основе одной из предыдущих копий, начальный LSN которой известен.
`pg_wal_summary_contents` (*`tli`* `bigint`, *`start_lsn`* `pg_lsn` , *`end_lsn`* `pg_lsn`) → `setof record` (*`relfilenode`* `oid`, *`reltablespace`* `oid`, *`reldatabase`* `oid`, *`relforknumber`* `smallint`, *`relblocknumber`* `bigint`, *`is_limit_block`* `boolean`) Выдаёт информацию о содержимом одного файла сводок WAL, идентифицируемого по TLI и начальному и конечному LSN. Каждая строка с `is_limit_block` =false указывает, что блок, идентифицированный оставшимися выходными столбцами, был изменён по крайней мере одной записью WAL в диапазоне записей, попавших в этот файл. Каждая строка с `is_limit_block` =true указывает, что либо (а) слой отношения был усечён до длины, заданной `relblocknumber` в соответствующем диапазоне записей WAL, либо (b)

Функция **Описание**
слой отношения был создан или удалён в соответствующем диапазоне записей WAL, в таких случаях `relblocknumber` будет равен нулю.
`pg_get_wal_summarizer_state` () → `record` (*`summarized_tli`* `bigint`, *`summarized_lsn`* `pg_lsn` , *`pending_lsn`* `pg_lsn` , *`summarizer_pid`* `int`) Выдаёт информацию о ходе создания сводок WAL. Если процесс создания сводок WAL ни разу не запускался с момента запуска экземпляра, то `summarized_tli` и `summarized_lsn` будут равны `0` и `0/0` соответственно, в противном случае в этих полях выводятся значения TLI и конечного LSN последнего файла сводки WAL, записанного на диск. Если процесс создания сводок WAL в данный момент запущен, то в `pending_lsn` возвращается конечный LSN последней использованной записи, который всегда должен быть больше или равен `summarized_lsn` . Если процесс создания сводок WAL не запущен, LSN будет равен `summarized_lsn` . `summarizer_pid` — это PID процесса создания сводок WAL, если он запущен, в противном случае — NULL. В качестве особого исключения, процесс создания сводок WAL не будет генерировать файлы сводок, если запущен для журнала, сгенерированного при `wal_level=minimal` , поскольку такие сводки было бы небезопасно использовать в качестве основы для инкрементального резервного копирования. В этом случае вышеуказанные поля продолжат обновляться, как если бы сводки генерировались, но при этом запись на диск не происходит. Как только `wal_level` изменится на `replica` или выше, процесс создания сводок возобновит запись сводок на диск.

9.28. Функции для системного администрирования

Функции, описанные в этом разделе, предназначены для контроля и управления сервером Postgres Pro.

9.28.1. Функции для управления конфигурацией

В Таблице 9.93 показаны функции, позволяющие получить и изменить значения параметров конфигурации выполнения.

Таблица 9.93. Функции для управления конфигурацией

Функция **Описание** **Пример(ы)**

`current_setting` (*`setting_name`* `text` [, *`missing_ok`* `boolean`]) → `text`
Выдаёт текущее значение параметра *`setting_name`* . Если такого параметра нет, `current_setting` выдаёт ошибку, если только дополнительно не передан параметр *`missing_ok`* со значением `true` (в этом случае выдаётся NULL). Эта функция соответствует SQL-команде SHOW.
`current_setting('datestyle')` → `ISO, MDY`

`set_config` (*`setting_name`* `text`, *`new_value`* `text`, *`is_local`* `boolean`) → `text`
Устанавливает для параметра *`setting_name`* значение *`new_value`* и возвращает это значение. Если параметр *`is_local`* равен `true`, новое значение будет действовать только в рамках текущей транзакции. Чтобы это значение действовало на протяжении текущего сеанса, присвойте этому параметру `false`. Эта функция соответствует SQL-команде SET.
`set_config('log_statement_stats', 'off', false)` → `off`

9.28.2. Функции для передачи сигналов серверу

Функции, перечисленные в Таблице 9.94, позволяют передавать управляющие сигналы другим серверным процессам. Вызывать эти функции по умолчанию разрешено только суперпользователям, но доступ к ним можно дать и другим пользователям командой `GRANT`, кроме явно отмеченных исключений.

Все эти функции возвращают `true`, если сигнал успешно отправлен, и `false`, если отправить сигнал не удалось.

Таблица 9.94. Функции для передачи сигналов серверу

Функция **Описание**
`pg_cancel_backend` (*`pid`* `integer`) → `boolean` Отменяет текущий запрос в сеансе, который обслуживается процессом с заданным PID. Это действие разрешается и ролям, являющимся членами роли, запрос которой отменяется, и ролям с правами роли `pg_signal_backend` ; однако только суперпользователям разрешено отменять запросы других суперпользователей.
`pg_log_backend_memory_contexts` (*`pid`* `integer`) → `boolean` Запрашивает вывод в журнал информации о контекстах памяти обслуживающего процесса с указанным PID. Эта функция может отправлять запрос обслуживающим и вспомогательным процессам, кроме процесса протоколирования. Запрошенная информация будет выведена в сообщениях уровня `LOG`, которые появятся в журнале сервера в зависимости от заданной конфигурации журнала (за дополнительными сведениями обратитесь к Разделу 18.8), но не будут передаваться клиенту независимо от client_min_messages.
`pg_reload_conf` () → `boolean` Даёт всем процессам сервера Postgres Pro команду перезагрузить файлы конфигурации. (Для этого посылается сигнал SIGHUP главному процессу, который, в свой очередь, посылает SIGHUP всем своим дочерним процессам.) Вы можете прочитать представления `pg_file_settings` , `pg_hba_file_rules` и `pg_ident_file_mappings` , чтобы проверить файлы конфигурации на предмет возможных ошибок перед перезагрузкой.
`pg_rotate_logfile` () → `boolean` Указывает менеджеру журнала сообщений немедленно переключиться на новый файл. Это имеет смысл, только когда работает встроенный сборщик сообщений, так как без него подпроцесс менеджера журнала не запускается.
`pg_terminate_backend` (*`pid`* `integer`, *`timeout`* `bigint DEFAULT 0`) → `boolean` Завершает сеанс, который обслуживается процессом с заданным PID. Это действие разрешается и ролям, являющимся членами роли, процесс которой прерывается, и ролям с правами роли `pg_signal_backend` ; однако только суперпользователям разрешено прерывать обслуживающие процессы других суперпользователей. Если параметр *`timeout`* не задан или равен нулю, данная функция возвращает `true` независимо от того, действительно ли завершается процесс; то есть результат показывает только, что сигнал был отправлен успешно. Если параметр *`timeout`* задан (в миллисекундах) и больше нуля, функция ожидает, пока процесс не будет фактически завершён или пока не пройдёт заданное время. Если процесс завершён, функция возвращает `true`. В случае тайм-аута выдаётся предупреждение и возвращается `false`.

`pg_cancel_backend` и `pg_terminate_backend` передают сигналы (SIGINT и SIGTERM, соответственно) серверному процессу с заданным кодом PID. Код активного процесса можно получить из столбца `pid` представления `pg_stat_activity` или просмотрев на сервере процессы с именем `postgres` (используя ps в Unix или Диспетчер задач в Windows). Роль пользователя активного процесса можно узнать в столбце `usename` представления `pg_stat_activity`.

`pg_log_backend_memory_contexts` может использоваться для получения в журнале сервера информации о контекстах памяти. Например:

```
postgres=# SELECT pg_log_backend_memory_contexts(pg_backend_pid());
 pg_log_backend_memory_contexts
--------------------------------
```

```
 t
(1 row)
```

Для каждого контекста памяти будет выводиться одно сообщение. Например:

```
LOG:  logging memory contexts of PID 10377
STATEMENT:  SELECT pg_log_backend_memory_contexts(pg_backend_pid());
LOG:  level: 0; TopMemoryContext: 80800 total in 6 blocks; 14432 free (5 chunks); 66368
 used
LOG:  level: 1; pgstat TabStatusArray lookup hash table: 8192 total in 1 blocks; 1408
 free (0 chunks); 6784 used
LOG:  level: 1; TopTransactionContext: 8192 total in 1 blocks; 7720 free (1 chunks);
 472 used
LOG:  level: 1; RowDescriptionContext: 8192 total in 1 blocks; 6880 free (0 chunks);
 1312 used
LOG:  level: 1; MessageContext: 16384 total in 2 blocks; 5152 free (0 chunks); 11232
 used
LOG:  level: 1; Operator class cache: 8192 total in 1 blocks; 512 free (0 chunks); 7680
 used
LOG:  level: 1; smgr relation table: 16384 total in 2 blocks; 4544 free (3 chunks);
 11840 used
LOG:  level: 1; TransactionAbortContext: 32768 total in 1 blocks; 32504 free (0
 chunks); 264 used
...
LOG:  level: 1; ErrorContext: 8192 total in 1 blocks; 7928 free (3 chunks); 264 used
LOG:  Grand total: 1651920 bytes in 201 blocks; 622360 free (88 chunks); 1029560 used
```

Если у одного родителя более 100 дочерних контекстов, в журнале описываются первые 100 дочерних контекстов, а по остальным выдаётся краткая сводка. Обратите внимание, что частые вызовы этой функции могут создавать значительную нагрузку, так как она может записывать в журнал массу сообщений.

9.28.3. Функции управления резервным копированием

Функции, перечисленные в Таблице 9.95, предназначены для выполнения резервного копирования «на ходу». Эти функции нельзя выполнять во время восстановления (за исключением `pg_backup_start`, `pg_backup_stop` и `pg_wal_lsn_diff`).

Подробнее практическое применение этих функций описывается в Разделе 24.3.

Таблица 9.95. Функции управления резервным копированием

Функция **Описание**
`pg_create_restore_point` (*name* `text`) → `pg_lsn` Создаёт в журнале предзаписи именованный маркер, который можно использовать как цель при восстановлении, и возвращает соответствующую ему позицию в журнале. Полученное имя затем можно указать в параметре recovery_target_name, определив тем самым точку, до которой будет выполняться восстановление. Учтите, что если будет создано несколько точек восстановления с одним именем, восстановление остановится на первой точке с этим именем. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_current_wal_flush_lsn` () → `pg_lsn` Выдаёт текущую позицию сброса данных в журнале предзаписи (см. примечания ниже).
`pg_current_wal_insert_lsn` () → `pg_lsn` Выдаёт текущую позицию добавления в журнале предзаписи (см. примечания ниже).
`pg_current_wal_lsn` () → `pg_lsn` Выдаёт текущую позицию записи в журнале предзаписи (см. примечания ниже).

Функция **Описание**
`pg_backup_start` (*`label`* `text` [, *`fast`* `boolean`]) → `pg_lsn` Подготавливает сервер к резервному копированию «на лету». Единственный обязательный параметр задаёт произвольную пользовательскую метку резервной копии. (Обычно это имя, которое получит файл резервной копии.) Если необязательный второй параметр задан и имеет значение `true`, функция `pg_backup_start` должна выполниться максимально быстро. Это означает, что принудительно будет выполнена контрольная точка, вследствие чего кратковременно увеличится нагрузка на ввод-вывод и параллельно выполняемые запросы могут замедлиться. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_backup_stop` ([*`wait_for_archive`* `boolean`]) → `record` (*`lsn`* `pg_lsn` , *`labelfile`* `text`, *`spcmapfile`* `text`) Завершает выполнение резервного копирования. В результате возвращается ожидаемое содержимое файла метки резервной копии и файла карты табличных пространств, которое нужно сохранить в файлы в архиве (не в каталоге данных). Эти файлы нельзя записывать в каталог работающей базы данных (в противном случае Postgres Pro не сможет перезапуститься, если произойдёт сбой). У этой функции есть также необязательный параметр типа `boolean`. Если он равен false, `pg_backup_stop` завершится сразу после окончания резервного копирования, не дожидаясь архивации WAL. Это поведение полезно только для программ резервного копирования, которые осуществляют архивацию WAL независимо. Если же WAL не будет заархивирован вовсе, резервная копия может оказаться неполной, и, как следствие, непригодной для восстановления. Когда он равен true (по умолчанию), `pg_backup_stop` будет ждать выполнения архивации WAL, если архивирование включено. Для резервного сервера это означает, что ожидание возможно только при условии `archive_mode = always`. Если активность записи на ведущем сервере невысока, может иметь смысл выполнить на нём `pg_switch_wal` для немедленного переключения сегмента. При выполнении на ведущем эта функция также создаёт файл истории резервного копирования в области архива журнала предзаписи. В этом файле для данной резервной копии сохраняется метка, заданная при вызове `pg_backup_start` , начальная и конечная позиция в журнале предзаписи, а также время начала и окончания копирования. После записи конечной позиции текущая позиция автоматически перемещается к следующему файлу журнала предзаписи, чтобы файл конечной позиции можно было немедленно архивировать для завершения резервного копирования. В результате эта функция выдаёт единственную запись. Столбец *`lsn`* в ней содержит позицию завершения копирования в журнале предзаписи (её также можно игнорировать). Во втором столбце передаётся содержимое файла метки резервной копии, а в третьем — содержимое файла карты табличных пространств. Эту информацию нужно сохранить в составе резервной копии, она потребуется в процессе восстановления. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_switch_wal` () → `pg_lsn` Производит принудительное переключение журнала предзаписи на новый файл, что позволяет архивировать текущий (в предположении, что выполняется непрерывная архивация). Результат функции — конечная позиция в только что законченном файле журнала предзаписи + 1. Если с момента последнего переключения файлов не было активности, отражающейся в журнале предзаписи, `pg_switch_wal` не делает ничего и возвращает начальную позицию в файле журнала предзаписи, используемом в данный момент. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_walfile_name` (*`lsn`* `pg_lsn`) → `text`

Функция Описание
Выдаёт для заданной позиции в журнале предзаписи имя соответствующего файла WAL.
`pg_walfile_name_offset` (*`lsn`* `pg_lsn`) → `record` (*`file_name`* `text`, *`file_offset`* `integer`) Выдаёт для заданной позиции в журнале предзаписи имя соответствующего файла и байтовое смещение в нём.
`pg_split_walfile_name` (*`file_name`* `text`) → `record` (*`segment_number`* `numeric`, *`timeline_id`* `bigint`) Извлекает последовательный номер и идентификатор линии времени из имени файла WAL.
`pg_wal_lsn_diff` (*`lsn1`* `pg_lsn` , *`lsn2`* `pg_lsn`) → `numeric` Вычисляет разницу в байтах (*`lsn1`* - *`lsn2`*) между двумя позициями в журнале предзаписи. Полученный результат можно использовать с `pg_stat_replication` или с некоторыми функциями, перечисленными в Таблица 9.95, для определения задержки репликации.

`pg_current_wal_lsn` выводит текущую позицию записи в журнале предзаписи в том же формате, что и вышеописанные функции. `pg_current_wal_insert_lsn` подобным образом выводит текущую позицию добавления, а `pg_current_wal_flush_lsn` — позицию сброса данных журнала. Позицией добавления называется «логический» конец журнала предзаписи в любой момент времени, тогда как позиция записи указывает на конец данных, фактически вынесённых из внутренних буферов сервера, а позиция сброса показывает, до какого места данные считаются сохранёнными в надёжном хранилище. Позиция записи отмечает конец данных, которые может видеть снаружи внешний процесс, и именно она представляет интерес при копировании частично заполненных файлов журнала. Позиция добавления и позиция сброса выводятся в основном для отладки серверной части. Все эти функции выполняются в режиме «только чтение» и не требуют прав суперпользователя.

Используя функцию `pg_walfile_name_offset`, из значения `pg_lsn` можно получить имя соответствующего ему файла журнала предзаписи и байтовое смещение в этом файле. Например:

```
postgres=# SELECT * FROM pg_walfile_name_offset((pg_backup_stop()).lsn);
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

Родственная ей функция `pg_walfile_name` извлекает только имя файла журнала предзаписи.

`pg_split_walfile_name` полезна для вычисления LSN на основе смещения файла и имени файла WAL, например:

```
postgres=# \set file_name '000000010000000100C000AB'
postgres=# \set offset 256
postgres=# SELECT '0/0'::pg_lsn + pd.segment_number * ps.setting::int + :offset AS lsn
  FROM pg_split_walfile_name(:'file_name') pd,
       pg_show_all_settings() ps
  WHERE ps.name = 'wal_segment_size';
      lsn
---------------
 C001/AB000100
(1 row)
```

9.28.4. Функции управления восстановлением

Функции, показанные в Таблице 9.96, предоставляют сведения о текущем состоянии ведомого сервера. Эти функции могут выполняться как во время восстановления, так и в обычном режиме работы.

Таблица 9.96. Функции для получения информации о восстановлении

Функция	Описание
`pg_is_in_recovery` () → `boolean`	Возвращает true, если в данный момент выполняется процедура восстановления.
`pg_last_wal_receive_lsn` () → `pg_lsn`	Выдаёт позицию последней записи в журнале предзаписи, которая была получена и записана на диск в процессе потоковой репликации. Пока выполняется потоковая репликация, эта позиция постоянно увеличивается. По окончании восстановления она остаётся на записи WAL, полученной и записанной на диск последней. Если потоковая репликация отключена или ещё не запускалась, функция возвращает `NULL`.
`pg_last_wal_replay_lsn` () → `pg_lsn`	Выдаёт позицию последней записи в журнале предзаписи, которая была воспроизведёна при восстановлении. В процессе восстановления эта позиция постоянно увеличивается. По окончании этого процесса она остаётся на записи WAL, которая была восстановлена последней. Если сервер при запуске не выполнял процедуру восстановления, эта функция выдаёт `NULL`.
`pg_last_xact_replay_timestamp` () → `timestamp with time zone`	Выдаёт отметку времени последней транзакции, воспроизведённой при восстановлении. Это время, когда на главном сервере произошла фиксация или откат записи WAL для этой транзакции. Если в процессе восстановления не была воспроизведена ни одна транзакция, эта функция выдаёт `NULL`. В противном случае возвращаемое значение постоянно увеличивается в процессе восстановления. По окончании восстановления в нём остаётся время транзакции, которая была восстановлена последней. Если сервер при запуске не выполнял процедуру восстановления, эта функция выдаёт `NULL`.
`pg_get_wal_resource_managers` () → `setof record` (*`rm_id`* `integer`, *`rm_name`* `text`, *`rm_builtin`* `boolean`)	Выдаёт загруженные на данный момент менеджеры ресурсов WAL в системе. Столбец *`rm_builtin`* показывает, является ли данный менеджер ресурсов встроенным или пользовательским, то есть создаваемым расширением.

Функции, перечисленные в Таблице 9.97 управляют процессом восстановления. Вызывать их в другое время нельзя.

Таблица 9.97. Функции управления восстановлением

Функция	Описание
`pg_is_wal_replay_paused` () → `boolean`	Возвращает true, если запрошена приостановка восстановления.
`pg_get_wal_replay_pause_state` () → `text`	Возвращает состояние приостановки восстановления. Возвращаемые значения: `not paused` — приостановка не запрашивалась, `pause requested` — получен запрос на приостановку, но восстановление ещё не приостановлено, и `paused`, если восстановление действительно приостановлено.
`pg_promote` (*`wait`* boolean DEFAULT `true`, *`wait_seconds`* integer DEFAULT `60`) → `boolean`	Повышает статус ведомого сервера до ведущего. Если параметр *`wait`* равен `true` (по умолчанию), эта функция дожидается завершения операции повышения в течение *`wait_seconds`* секунд и возвращает `true` в случае успешного повышения или `false` в противном случае. Если параметр *`wait`* равен `false`, функция возвращает `true` сразу после передачи процессу postmaster сигнала `SIGUSR1`, инициирующего повышение. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.

Функция **Описание**
`pg_wal_replay_pause` () → `void` Запрашивает приостановку восстановления. Запрос приостановки может быть выполнен не сразу. Если вы хотите убедиться, что восстановление действительно приостановлено, проверьте состояние приостановки восстановления с помощью функции `pg_get_wal_replay_pause_state()`. Заметьте, что `pg_get_wal_replay_paused` показывает только был ли сделан запрос. Когда восстановление приостановлено, запись изменений в базу не производится. В режиме «горячего резерва» все последующие запросы будут видеть один согласованный снимок базы данных, что исключает конфликты запросов до продолжения восстановления. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_wal_replay_resume` () → `void` Возобновляет восстановление, если оно было приостановлено. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.

Функции `pg_wal_replay_pause` и `pg_wal_replay_resume` нельзя выполнять в процессе повышения. Если повышение запрашивается, когда восстановление приостановлено, сервер выходит из состояния паузы и продолжает процедуру повышения.

Когда потоковая репликация отключена, пауза при восстановлении может длиться сколь угодно долго без каких-либо проблем. Если же потоковая репликация осуществляется, новые записи WAL продолжат поступать и заполнят весь диск рано или поздно, в зависимости от длительности паузы, интенсивности записи в WAL и объёма свободного пространства.

9.28.5. Функции синхронизации снимков

Postgres Pro позволяет синхронизировать снимки состояния между сеансами баз данных. *Снимок состояния* определяет, какие данные видны транзакции, работающей с этим снимком. Синхронизация снимков необходима, когда в двух или более сеансах нужно видеть одно и то же содержимое базы данных. Если в двух сеансах транзакции запускаются независимо, всегда есть вероятность, что некая третья транзакция будет зафиксирована между командами `START TRANSACTION` для первых двух, и в результате в одном сеансе будет виден результат третьей, а в другом — нет.

Для решения этой проблемы Postgres Pro позволяет транзакции *экспортировать* снимок состояния, с которым она работает. Пока экспортирующая этот снимок транзакция выполняется, другие транзакции могут *импортировать* его и, таким образом, увидеть абсолютно то же состояние базы данных, что видит первая транзакция. Но учтите, что любые изменения, произведённые этими транзакциями, будут не видны для других, как это и должно быть с изменениями в незафиксированных транзакциях. Таким образом, транзакции синхронизируют только начальное состояние данных, а последующие производимые в них изменения изолируются как обычно.

Снимки состояния экспортируются с помощью функции `pg_export_snapshot`, показанной в Таблице 9.98, и импортируются командой SET TRANSACTION.

Таблица 9.98. Функции синхронизации снимков

Функция **Описание**
`pg_export_snapshot` () → `text` Сохраняет текущий снимок состояния транзакции и возвращает строку типа `text` с идентификатором этого снимка. Эта строка должна передаваться внешними средствами клиентам, которые будут импортировать этот снимок. Снимок может быть импортирован только при условии, что экспортировавшая его транзакция ещё не завершена. Если необходимо, транзакция может экспортировать несколько снимков. Заметьте, что это имеет смысл только для транзакций уровня `READ COMMITTED`, так как транзакции

Функция Описание	
	уровня изоляции `REPEATABLE READ` и выше работают с одним снимком состояния на протяжении всего своего существования. После того как транзакция экспортировала снимок, её нельзя сделать подготовленной с помощью PREPARE TRANSACTION.
`pg_log_standby_snapshot` () → `pg_lsn`	Сделать снимок выполняющихся транзакций и записать его в WAL, не дожидаясь, пока процессы bgwriter или checkpointer запишут его. Это полезно для логического декодирования на резервном сервере, поскольку создание логического слота должно откладываться до тех пор, пока такая запись не будет воспроизведена на резервном сервере.

9.28.6. Функции управления репликацией

В Таблице 9.99 показаны функции, предназначенные для управления механизмом репликации и взаимодействия с ним. Чтобы изучить этот механизм подробнее, обратитесь к Подразделу 25.2.5, Подразделу 25.2.6 и Главе 48. По умолчанию только суперпользователи имеют разрешение выполнять функции с источниками репликации, но его можно дать и другим, воспользовавшись командой `GRANT`. Использование данных функций для слотов репликации разрешено только суперпользователям и пользователям, имеющим право `REPLICATION`.

Многие из этих функций соответствуют командам в протоколе репликации; см. Раздел 53.4.

Функции, описанные в Подразделе 9.28.3, Подразделе 9.28.4 и Подразделе 9.28.5, также имеют отношение к репликации.

Таблица 9.99. Функции управления репликацией

Функция Описание	
`pg_create_physical_replication_slot` (*slot_name* `name` [, *immediately_reserve* `boolean`, *temporary* `boolean`]) → `record` (*slot_name* `name`, *lsn* `pg_lsn`)	Создаёт новый физический слот репликации с именем *slot_name*. Необязательный второй параметр, когда он равен `true`, указывает, что LSN для этого слота репликации должен быть зарезервирован немедленно; в противном случае LSN резервируется при первом подключении клиента потоковой репликации. Передача изменений из физического слота возможна только по протоколу потоковой репликации — см. Раздел 53.4. Необязательный третий параметр, *temporary*, когда он равен true, указывает, что этот слот не должен постоянно храниться на диске, так как он предназначен только для текущего сеанса. Временные слоты также освобождаются при любой ошибке. Эта функция соответствует команде протокола репликации `CREATE_REPLICATION_SLOT ... PHYSICAL`.
`pg_drop_replication_slot` (*slot_name* `name`) → `void`	Удаляет физический или логический слот репликации с именем *slot_name*. Соответствует команде протокола репликации `DROP_REPLICATION_SLOT`.
`pg_create_logical_replication_slot` (*slot_name* `name`, *plugin* `name` [, *temporary* `boolean`, *twophase* `boolean`, *failover* `boolean`]) → `record` (*slot_name* `name`, *lsn* `pg_lsn`)	Создаёт новый логический (декодирующий) слот репликации с именем *slot_name*, используя модуль вывода *plugin*. Необязательный третий параметр, *temporary*, когда равен true, указывает, что этот слот не должен постоянно храниться на диске, так как предназначен только для текущего сеанса. Временные слоты также освобождаются при любой ошибке. Необязательный четвёртый параметр, *twophase*, когда равен true, указывает, что для данного слота включается декодирование подготовленных транзакций. Необязательный пятый параметр *failover*, когда равен true, указывает, что этот слот может быть синхронизирован с резервными серверами, чтобы логическая репликация могла быть возобновлена после отработки отказа. Вызов этой функции равнозначен выполнению команды протокола репликации `CREATE_REPLICATION_SLOT ... LOGICAL`.

Функция **Описание**
pg_copy_physical_replication_slot (*src_slot_name* name, *dst_slot_name* name [, *temporary* boolean]) → record (*slot_name* name, *lsn* pg_lsn) Копирует существующий слот физической репликации с именем *src_slot_name* в слот с именем *dst_slot_name* . Скопированный слот физической репликации начинает резервировать WAL с того же последовательного номера LSN, что и исходный слот. Параметр *temporary* является необязательным и позволяет указать, будет ли слот временным. Если параметр *temporary* опущен, сохраняется то же свойство, что имеет исходный слот. Копирование аннулированного слота не допускается.
pg_copy_logical_replication_slot (*src_slot_name* name, *dst_slot_name* name [, *temporary* boolean [, *plugin* name]]) → record (*slot_name* name, *lsn* pg_lsn) Копирует существующий слот логической репликации с именем *src_slot_name* в слот с именем *dst_slot_name* , с возможностью смены модуля вывода и свойства временности. Скопированный логический слот начинает передачу с того же последовательного номера LSN, что и исходный слот. Параметры *temporary* и *plugin* являются необязательными; если они опущены, сохраняются свойства исходного слота. Параметр failover логического слота источника не копируется, и по умолчанию для него устанавливается значение false. Это позволяет устранить риск невозможности продолжить логическую репликацию после аварийного переключения на резервный сервер, с которым синхронизируется слот. Копирование аннулированного слота не допускается.
pg_logical_slot_get_changes (*slot_name* name, *upto_lsn* pg_lsn , *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn , *xid* xid, *data* text) Возвращает изменения в слоте *slot_name* с позиции, до которой ранее были получены изменения. Если параметры *upto_lsn* и *upto_nchanges* равны NULL, логическое декодирование продолжится до конца журнала транзакций. Если *upto_lsn* не NULL, декодироваться будут только транзакции, зафиксированные до заданного LSN. Если *upto_nchanges* не NULL, декодирование остановится, когда число строк, полученных при декодировании, превысит заданное значение. Заметьте однако, что фактическое число возвращённых строк может быть больше, так как это ограничение проверяется только после добавления строк, декодированных для очередной транзакции. Если указанный слот является слотом для отработки отказа логической репликации, то функция не выдаст результат, пока все физические слоты, указанные в synchronized_standby_slots , не подтвердят получение WAL.
pg_logical_slot_peek_changes (*slot_name* name, *upto_lsn* pg_lsn , *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn , *xid* xid, *data* text) Работает так же, как функция pg_logical_slot_get_changes() , но не забирает изменения; то есть, они будут получены снова при следующих вызовах.
pg_logical_slot_get_binary_changes (*slot_name* name, *upto_lsn* pg_lsn , *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn , *xid* xid, *data* bytea) Работает так же, как функция pg_logical_slot_get_changes() , но выдаёт изменения в типе bytea.
pg_logical_slot_peek_binary_changes (*slot_name* name, *upto_lsn* pg_lsn , *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn , *xid* xid, *data* bytea) Работает аналогично функции pg_logical_slot_peek_changes() , но выдаёт изменения в значении bytea.
pg_replication_slot_advance (*slot_name* name, *upto_lsn* pg_lsn) → record (*slot_name* name, *end_lsn* pg_lsn) Продвигает текущую подтверждённую позицию слота репликации с именем *slot_name* . Слот нельзя переместить назад, а также вперёд за текущую позицию добавления. Воз-

Функция **Описание**
вращает имя слота и позицию, до которой он фактически продвинулся. Информация об изменившемся положении слота, если он продвинулся, сохраняется только при ближайшей контрольной точке. Поэтому в случае прерывания работы слот может оказаться в предыдущем положении. Если указанный слот является слотом для отработки отказа логической репликации, то функция не выдаст результат, пока все физические слоты, указанные в `synchronized_standby_slots`, не подтвердят получение WAL.
`pg_replication_origin_create` (*`node_name`* `text`) → `oid` Создаёт источник репликации с заданным внешним именем и возвращает назначенный ему внутренний идентификатор.
`pg_replication_origin_drop` (*`node_name`* `text`) → `void` Удаляет ранее созданный источник репликации, в том числе связанную информацию о воспроизведении.
`pg_replication_origin_oid` (*`node_name`* `text`) → `oid` Ищет источник репликации по имени и возвращает внутренний идентификатор. Если такой источник не находится, возвращает `NULL`.
`pg_replication_origin_session_setup` (*`node_name`* `text`) → `void` Помечает текущий сеанс как воспроизводящий изменения из указанного источника, что позволяет отслеживать процесс воспроизведения. Может применяться, только если в текущий момент источник ещё не выбран. Для отмены этого действия вызовите `pg_replication_origin_session_reset`.
`pg_replication_origin_session_reset` () → `void` Отменяет действие `pg_replication_origin_session_setup()`.
`pg_replication_origin_session_is_setup` () → `boolean` Возвращает true, если в текущем сеансе выбран источник репликации.
`pg_replication_origin_session_progress` (*`flush`* `boolean`) → `pg_lsn` Возвращает позицию воспроизведения для источника репликации, выбранного в текущем сеансе. Параметр *`flush`* определяет, будет ли гарантироваться сохранение локальной транзакции на диске.
`pg_replication_origin_xact_setup` (*`origin_lsn`* `pg_lsn`, *`origin_timestamp`* `timestamp with time zone`) → `void` Помечает текущую транзакцию как воспроизводящую транзакцию, зафиксированную с указанным LSN и временем. Может вызываться только после того, как источник репликации был выбран в результате вызова `pg_replication_origin_session_setup()`.
`pg_replication_origin_xact_reset` () → `void` Отменяет действие `pg_replication_origin_xact_setup()`.
`pg_replication_origin_advance` (*`node_name`* `text`, *`lsn`* `pg_lsn`) → `void` Устанавливает положение репликации для заданного узла в указанную позицию. Это в основном полезно для установки начальной позиции или новой позиции после изменения конфигурации и подобных действий. Но учтите, что неосторожное использование этой функции может привести к несогласованности реплицированных данных.
`pg_replication_origin_progress` (*`node_name`* `text`, *`flush`* `boolean`) → `pg_lsn` Выдаёт позицию воспроизведения для заданного источника репликации. Параметр *`flush`* определяет, будет ли гарантироваться сохранение локальной транзакции на диске.
`pg_logical_emit_message` (*`transactional`* `boolean`, *`prefix`* `text`, *`content`* `text` [, *`flush`* `boolean DEFAULT false`]) → `pg_lsn`

Функция **Описание**
`pg_logical_emit_message` (*`transactional`* `boolean`, *`prefix`* `text`, *`content`* `bytea` [, *`flush`* `boolean DEFAULT false`]) → `pg_lsn` Генерирует сообщение логического декодирования. Её можно использовать для передачи через WAL произвольных сообщений модулям логического декодирования. Параметр *`transactional`* определяет, должно ли сообщение относиться к текущей транзакции или оно должно записываться немедленно и декодироваться сразу, как только эта запись будет прочитана при логическом декодировании. Параметр *`prefix`* задаёт текстовый префикс, по которому модуль логического декодирования может легко распознать интересующие именно его сообщения. В параметре *`content`* передаётся содержимое сообщения, в текстовом или двоичном виде. Параметр *`flush`* (по умолчанию имеет значение `false`) определяет, будут ли сообщения немедленно сохраняться в WAL или нет. Параметр *`flush`* не работает вместе с параметром *`transactional`*, поскольку запись WAL сообщения сохраняется вместе с его транзакцией.
`pg_sync_replication_slots` () → `void` Синхронизирует слоты для отработки отказа логической репликации с ведущего сервера на резервный. Эта функция может выполняться только на резервном сервере. Временные синхронизированные слоты, если таковые имеются, не могут использоваться для логического декодирования и должны быть удалены после повышения. За подробностями обратитесь к Подразделу 47.2.3. Обратите внимание, что эта функция в первую очередь используется в целях тестирования и отладки, а потому должна быть использована с осторожностью. Кроме того, она не может быть выполнена, если включён параметр `sync_replication_slots` и рабочий процесс slotsync уже запущен для выполнения синхронизации слотов. **Внимание** Если после выполнения функции отключён параметр `hot_standby_feedback` на резервном сервере или удалён физический слот, настроенный в `primary_slot_name`, необходимые строки синхронизированного слота могут быть удалены процессом VACUUM на ведущем сервере, в результате чего синхронизированный слот будет аннулирован.

9.28.7. Функции управления объектами баз данных

Функции, показанные в Таблице 9.100, вычисляют объём, который занимают на диске объекты базы данных, и помогают воспринять и проанализировать информацию об используемом объёме. Результаты типа `bignit` представляют значения в байтах. Если им передаётся OID, не соответствующий существующему объекту, эти функции возвращают `NULL`.

Таблица 9.100. Функции получения размера объектов БД

Функция **Описание**
`pg_column_size` (`"any"`) → `integer` Показывает, сколько байт занимает при хранении отдельное значение. Если эта функция применяется непосредственно к значению в столбце таблицы, в результате отражается эффект возможного сжатия.
`pg_column_compression` (`"any"`) → `text` Показывает алгоритм сжатия, который использовался для сжатия отдельного значения переменной длины. Возвращает `NULL`, если значение не сжато.
`pg_column_toast_chunk_id` (`"any"`) → `oid`

Функция **Описание**
Показывает `chunk_id` значения в формате TOAST на диске. Возвращает `NULL`, если значение не в формате TOAST или не хранится на диске. За более подробным описанием TOAST обратитесь к Разделу 63.2.

`pg_database_size ( name ) → bigint`
`pg_database_size ( oid ) → bigint`
Вычисляет общий объём, который занимает на диске база данных с указанным именем или OID. Для использования этой функции необходимо иметь право `CONNECT` для заданной базы (оно даётся по умолчанию) или иметь права роли `pg_read_all_stats`.

`pg_indexes_size ( regclass ) → bigint` Вычисляет общий объём, который занимают на диске индексы, связанные с указанной таблицей.
`pg_relation_size (` *`relation`* `regclass [,` *`fork`* `text] ) → bigint` Вычисляет объём, который занимает на диске один «слой» заданного отношения. (Заметьте, что в большинстве случаев удобнее использовать более высокоуровневые функции `pg_total_relation_size` и `pg_table_size`, которые суммируют размер всех слоёв.) С одним аргументом она выдаёт размер основного слоя с данными отношения. Название другого интересующего слоя можно передать во втором аргументе: • `main` выдаёт размер основного слоя данных заданного отношения. • `fsm` выдаёт размер карты свободного места (см. Раздел 63.3), связанной с заданным отношением. • `vm` выдаёт размер карты видимости (см. Раздел 63.4), связанной с заданным отношением. • `init` выдаёт размер слоя инициализации для заданного отношения, если он имеется.
`pg_size_bytes ( text ) → bigint` Преобразует размер в понятном человеку формате (который выдаёт `pg_size_pretty`) в число байт. Возможные единицы: `bytes`, `B`, `kB`, `MB`, `GB`, `TB` и `PB`.

`pg_size_pretty ( bigint ) → text`
`pg_size_pretty ( numeric ) → text`
Преобразует размер в байтах в более понятный человеку формат с единицами измерения размера (bytes, kB, MB, GB, TB или PB, в зависимости от значения). Заметьте, что эти единицы определяются как степени 2, а не 10, так что 1kB — это 1024 байта, 1MB — 1024^2 = 1048576 байт и т. д.

`pg_table_size ( regclass ) → bigint` Вычисляет объём, который занимает на диске данная таблица, за исключением индексов (но включая её TOAST-таблицу (если она есть), карту свободного места и карту видимости).

`pg_tablespace_size ( name ) → bigint`
`pg_tablespace_size ( oid ) → bigint`
Вычисляет общий объём, который занимает на диске табличное пространство с заданным именем или OID. Для использования этой функции требуется иметь право `CREATE` для указанного табличного пространства или иметь права роли `pg_read_all_stats`, если только это не табличное пространство по умолчанию для текущей базы данных.

`pg_total_relation_size ( regclass ) → bigint` Вычисляет общий объём, который занимает на диске заданная таблица, включая все её индексы и данные TOAST. Результат этой функции равен значению `pg_table_size` `+` `pg_indexes_size`.

Вышеописанные функции, работающие с таблицами или индексами, принимают аргумент типа `regclass`, который представляет собой просто OID таблицы или индекса в системном каталоге `pg_class`. Однако вам не нужно вручную вычислять OID, так как процедура ввода значения `regclass` может сделать это за вас. За подробностями обратитесь к Разделу 8.19.

Функции, перечисленные в Таблице 9.101, помогают определить, в каких файлах на диске хранятся объекты базы данных.

Таблица 9.101. Функции определения расположения объектов

Функция Описание
`pg_relation_filenode` (*`relation`* `regclass`) → `oid` Выдаёт номер «файлового узла», связанного с заданным объектом. Файловым узлом называется основной компонент имени файла, используемого для хранения данных (подробнее это описано в Разделе 63.1). Для большинства отношений этот номер совпадает со значением `pg_class .relfilenode`, но для некоторых системных каталогов `relfilenode` равен 0, и нужно использовать эту функцию, чтобы узнать действительное значение. Если указанное отношение не хранится на диске, как например представление, данная функция возвращает NULL.
`pg_relation_filepath` (*`relation`* `regclass`) → `text` Выдаёт полный путь к файлу отношения (относительно каталога данных `PGDATA`).
`pg_filenode_relation` (*`tablespace`* `oid`, *`filenode`* `oid`) → `regclass` Выдаёт OID отношения, которое хранится в табличном пространстве, заданном по OID, в указанном файловом узле. По сути эта функция является обратной к `pg_relation_filepath` . Для табличного пространства по умолчанию можно передать нулевое значение OID. Если эта функция не находит в базе данных отношение по заданным значениям, она выдаёт `NULL`.

В Таблица 9.102 перечислены функции, предназначенные для управления правилами сортировки.

Таблица 9.102. Функции управления правилами сортировки

Функция Описание
`pg_collation_actual_version` (`oid`) → `text` Возвращает действующую версию объекта правила сортировки, которая в настоящее время установлена в операционной системе. Если она отличается от значения в `pg_collation .collversion`, может потребоваться перестроить объекты, зависящие от данного правила сортировки. См. также ALTER COLLATION.
`pg_database_collation_actual_version` (`oid`) → `text` Возвращает действующую версию правила сортировки БД, установленную на данный момент в операционной системе. Если она отличается от значения в `pg_database .datcollversion`, может потребоваться перестроить объекты, зависящие от данного правила сортировки. См. также ALTER DATABASE.
`pg_import_system_collations` (*`schema`* `regnamespace`) → `integer` Добавляет правила сортировки в системный каталог `pg_collation` , анализируя все локали, которые она находит в операционной системе. Эту информацию использует `initdb`; за подробностями обратитесь к Подразделу 22.2.2. Если позднее в систему устанавливаются дополнительные локали, эту функцию можно запустить снова, чтобы добавились правила сортировки для новых локалей. Локали, для которых обнаруживаются существующие записи в `pg_collation` , будут пропущены. (Объекты правил сортировки для локалей, которые перестают существовать в операционной системе, никогда не удаляются этой функцией.) В параметре *`schema`* обычно передаётся `pg_catalog` , но это не обязательно; правила сортировки могут быть установлены и в другую схему. Эта функция возвращает число созданных ей объектов правил сортировки; вызывать её разрешено только суперпользователям.

В Таблице 9.103 перечислены функции, предоставляющие информацию о структуре секционированных таблиц.

Таблица 9.103. Функции получения информации о секционировании

Функция	Описание
`pg_partition_tree` (`regclass`) → `setof record` (*`relid`* `regclass`, *`parentrelid`* `regclass`, *`isleaf`* `boolean`, *`level`* `integer`)	Выводит информацию о таблицах и индексах в дереве секционирования для заданной секционированной таблицы или секционированного индекса, отражая каждую секцию в отдельной строке. В этой информации представляется OID секции, OID её непосредственного родителя, логический признак того, что секция является конечной, и целое число, показывающее её уровень в иерархии. На уровне 0 будет находится указанная таблица или индекс, на уровне 1 непосредственные потомки-секции, на уровне 2 секции последних и т. д. Если заданное отношение не существует или не является секцией или секционированным отношением, эта функция выдаёт пустой результат.
`pg_partition_ancestors` (`regclass`) → `setof regclass`	Выводит список вышестоящих отношений для заданной секции, включая её саму. Если заданное отношение не существует или не является секцией или секционированным отношением, эта функция выдаёт пустой результат.
`pg_partition_root` (`regclass`) → `regclass`	Выдаёт самое верхнее отношение в дереве секционирования, к которому относится заданное отношение. Если заданное отношение не существует или не является секцией или секционированным отношением, эта функция выдаёт `NULL`.

Например, чтобы определить общий размер данных, содержащихся в секционированной таблице `measurement`, можно использовать следующий запрос:

```
SELECT pg_size_pretty(sum(pg_relation_size(relid))) AS total_size
  FROM pg_partition_tree('measurement');
```

9.28.8. Функции обслуживания индексов

В Таблице 9.104 перечислены функции, предназначенные для обслуживания индексов. (Заметьте, что задачи обслуживания обычно выполняются автоматически в ходе автоочистки; пользоваться данными функциями требуется только в особых случаях.) Выполнять эти функции во время восстановления нельзя. Использовать их разрешено только суперпользователям и владельцу определённого индекса.

Таблица 9.104. Функции обслуживания индексов

Функция	Описание
`brin_summarize_new_values` (*`index`* `regclass`) → `integer`	Сканирует указанный BRIN-индекс в поисках зон страниц в базовой таблице, ещё не обобщённых в индексе; для каждой такой зоны в результате сканирования этих страниц создаётся новый обобщающий кортеж в индексе. Возвращает эта функция число вставленных в индекс обобщающих записей о зонах страниц.
`brin_summarize_range` (*`index`* `regclass`, *`blockNumber`* `bigint`) → `integer`	Обобщает зону страниц, охватывающую данный блок, если она ещё не была обобщена. Эта функция подобна `brin_summarize_new_values`, но обрабатывает только одну выбранную зону страниц.
`brin_desummarize_range` (*`index`* `regclass`, *`blockNumber`* `bigint`) → `void`	Удаляет из BRIN-индекса кортеж, который обобщает зону страниц, охватывающую данный блок таблицы, если такой кортеж имеется.
`gin_clean_pending_list` (*`index`* `regclass`) → `bigint`	

Функция Описание
Очищает очередь указанного GIN-индекса, массово перемещая элементы из неё в основную структуру данных GIN, и возвращает число страниц, убранных из очереди. Если для обработки ей передаётся индекс GIN, построенный с отключённым параметром `fastupdate`, очистка не производится и возвращается 0, так как у такого индекса нет очереди записей. Подробнее об очереди записей и параметре `fastupdate` рассказывается в Подразделе 62.4.4.1 и Подразделе 62.4.5.

9.28.9. Функции для работы с обычными файлами

Функции, перечисленные в Таблице 9.105, предоставляют прямой доступ к файлам, находящимся на сервере. Обычным пользователям, не включённым в роль `pg_read_server_files`, они позволяют обращаться только к файлам в каталоге кластера баз данных и в каталоге `log_directory`. Для файлов в каталоге кластера этим функциям передаётся относительный путь, а для файлов журнала — путь, соответствующий значению параметра `log_directory`.

Заметьте, что пользователи, обладающие правом EXECUTE для `pg_read_file()` или связанных функций, имеют возможность читать любой файл на сервере, который может прочитать серверный процесс; на эти функции не действуют никакие ограничения доступа внутри базы данных. В частности это означает, что пользователь с такими правами может прочитать содержимое таблицы `pg_authid`, в которой хранятся данные аутентификации, равно как и любой другой файл в базе данных. Таким образом, разрешать доступ к этим функциям следует с большой осторожностью.

При предоставлении прав для этих функций обратите внимание, что записи таблицы, показывающие необязательные параметры, в большинстве случаев реализованы как несколько физических функций с разными списками параметров. Права должны быть предоставлены отдельно для каждой такой функции, если она будет использоваться. Команда psql `\df` может быть полезна для проверки фактических сигнатур функций.

Некоторые из этих функций принимают необязательный параметр *`missing_ok`*, который определяет их поведение в случае отсутствия файла или каталога. Если он равен `true`, функция возвращает `NULL` или пустой набор данных. Если же он равен `false`, в указанном случае выдаётся ошибка. (Условия сбоя, кроме «file not found» (файл не найден), в любом случае считаются ошибками.) Значение по умолчанию — `false`.

Таблица 9.105. Функции для работы с обычными файлами

Функция Описание
`pg_ls_dir` (*`dirname`* `text` [, *`missing_ok`* `boolean`, *`include_dot_dirs`* `boolean`]) → `setof text` Выдаёт имена всех файлов (а также каталогов и других специальных файлов) в заданном каталоге. Параметр *`include_dot_dirs`* определяет, будут ли в результирующий набор включаться каталоги «.» и «..». По умолчанию они не включаются, но их можно включить, чтобы с параметром *`missing_ok`* равным `true`, пустой каталог можно было отличить от несуществующего. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_ls_logdir` () → `setof record` (*`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone`) Выводит имя, размер и время последнего изменения (mtime) всех обычных файлов в каталоге журналов сервера. Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы исключаются из рассмотрения. По умолчанию доступ к этой функции имеют только суперпользователи или роли с правами роли `pg_monitor`, но право на её выполнение (EXECUTE) можно дать и другим пользователям.

Функция **Описание**
`pg_ls_waldir` () → `setof record` (*name* `text`, *size* `bigint`, *modification* `timestamp with time zone`) Выводит имя, размер и время последнего изменения (mtime) всех обычных файлов в каталоге журнала предзаписи (WAL) сервера. Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы исключаются из рассмотрения. По умолчанию доступ к этой функции имеют только суперпользователи или роли с правами роли `pg_monitor`, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_ls_logicalmapdir` () → `setof record` (*name* `text`, *size* `bigint`, *modification* `timestamp with time zone`) Выводит имя, размер и время последнего изменения (mtime) всех обычных файлов в каталоге сервера `pg_logical/mappings`. Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы исключаются из рассмотрения. По умолчанию доступ к этой функции имеют только суперпользователи и члены группы `pg_monitor`, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_ls_logicalsnapdir` () → `setof record` (*name* `text`, *size* `bigint`, *modification* `timestamp with time zone`) Выводит имя, размер и время последнего изменения (mtime) всех обычных файлов в каталоге сервера `pg_logical/snapshots`. Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы исключаются из рассмотрения. По умолчанию доступ к этой функции имеют только суперпользователи и члены группы `pg_monitor`, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_ls_replslotdir` (*slot_name* `text`) → `setof record` (*name* `text`, *size* `bigint`, *modification* `timestamp with time zone`) Выводит имя, размер и время последнего изменения (mtime) всех обычных файлов в каталоге сервера `pg_replslot/slot_name`, где *slot_name* — имя слота репликации, переданное функции. Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы исключаются из рассмотрения. По умолчанию доступ к этой функции имеют только суперпользователи и члены группы `pg_monitor`, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_ls_archive_statusdir` () → `setof record` (*name* `text`, *size* `bigint`, *modification* `timestamp with time zone`) Выводит имя, размер и время последнего изменения (mtime) всех обычных файлов в каталоге состояния архива WAL (`pg_wal/archive_status`). Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы исключаются из рассмотрения. По умолчанию доступ к этой функции имеют только суперпользователи и члены группы `pg_monitor`, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_ls_tmpdir` ([*tablespace* `oid`]) → `setof record` (*name* `text`, *size* `bigint`, *modification* `timestamp with time zone`) Выводит имя, размер и время последнего изменения (mtime) всех обычных файлов во временном каталоге для табличного пространства *tablespace*. Если параметр *tablespace* не задан, подразумевается табличное пространство `pg_default`. Файлы с именами, начинающимися с точки, каталоги и другие специальные файлы исключаются из рассмотрения. По умолчанию доступ к этой функции имеют только суперпользователи и члены группы `pg_monitor`, но право на её выполнение (EXECUTE) можно дать и другим пользователям.

Функция **Описание**
`pg_read_file` (*filename* `text` [, *offset* `bigint`, *length* `bigint`] [, *missing_ok* `boolean`]) → `text` Читает из текстового файла всё содержимое или фрагмент с заданного смещения (*offset*), размером не больше *length* байт (размер может быть меньше, если файл кончится раньше). Если смещение *offset* отрицательно, оно отсчитывается от конца файла. Если параметры *offset* и *length* опущены, возвращается всё содержимое файла. Прочитанные из файла байты обрабатываются как символы в кодировке базы данных; если они оказываются недопустимыми для этой кодировки, возникает ошибка. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.

`pg_read_binary_file` (*filename* `text` [, *offset* `bigint`, *length* `bigint`] [, *missing_ok* `boolean`]) → `bytea`
Читает из текстового файла всё содержимое или заданный фрагмент. Эта функция подобна `pg_read_file`, но может читать произвольные двоичные данные и возвращает результат в значении типа `bytea`, а не `text`; как следствие, никакие проверки кодировок не производятся.
По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
В сочетании с `convert_from` эту функцию можно применять для чтения текста в указанной кодировке и преобразования в кодировку базы данных:
`SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');`

`pg_stat_file` (*filename* `text` [, *missing_ok* `boolean`]) → `record` (*size* `bigint`, *access* `timestamp with time zone`, *modification* `timestamp with time zone`, *change* `timestamp with time zone`, *creation* `timestamp with time zone`, *isdir* `boolean`) Выдаёт запись, содержащую размер файла, время последнего обращения и последнего изменения, а также время последнего изменения состояния (только в Unix-системах), время создания (только в Windows) и флаг, показывающий, что это каталог. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.

9.28.10. Функции управления рекомендательными блокировками

Функции, перечисленные в Таблице 9.106, предназначены для управления рекомендательными блокировками. Подробнее об их использовании можно узнать в Подразделе 13.3.5.

Все эти функции предназначены для оперирования блокировками ресурсов, определяемых приложением и задаваемых одним 64-битным или двумя 32-битными ключами (заметьте, что пространства этих ключей не пересекаются). Если конфликтующую блокировку с тем же идентификатором уже удерживает другой сеанс, эти функции либо дожидаются освобождения ресурса, либо выдают в результате `false`, в зависимости от вида функции. Блокировки могут быть как исключительными, так и разделяемыми — разделяемая блокировка не конфликтует с другими разделяемыми блокировками того же ресурса, но конфликтует с исключительными. Блокировки могут устанавливаться на сеансовом уровне (тогда они удерживаются до момента освобождения или до завершения сеанса) и на транзакционном (тогда они удерживаются до конца текущей транзакции, освободить их вручную нет возможности). Когда поступает несколько запросов на блокировку сеансового уровня, они накапливаются, так что если один идентификатор ресурса был заблокирован три раза, должны поступить три запроса на освобождение блокировки, чтобы ресурс был разблокирован до завершения сеанса.

Таблица 9.106. Функции управления рекомендательными блокировками

Функция **Описание**
`pg_advisory_lock` (*key* `bigint`) → `void`

Функция **Описание**
`pg_advisory_lock` (*key1* `integer`, *key2* `integer`) → `void` Получает исключительную рекомендательную блокировку сеансового уровня, ожидая её, если это необходимо.

`pg_advisory_lock_shared` (*key* `bigint`) → `void`
`pg_advisory_lock_shared` (*key1* `integer`, *key2* `integer`) → `void`
Получает разделяемую рекомендательную блокировку сеансового уровня, ожидая её, если это необходимо.

`pg_advisory_unlock` (*key* `bigint`) → `boolean`
`pg_advisory_unlock` (*key1* `integer`, *key2* `integer`) → `boolean`
Освобождает ранее полученную исключительную рекомендательную блокировку сеансового уровня. Если блокировка освобождена успешно, эта функция возвращает `true`, а если сеанс не владел ей — `false`, при этом сервер выдаёт предупреждение SQL.

`pg_advisory_unlock_all` () → `void` Освобождает все закреплённые за текущим сеансом рекомендательные блокировки сеансового уровня. (Эта функция неявно вызывается в конце любого сеанса, даже при нештатном отключении клиента.)

`pg_advisory_unlock_shared` (*key* `bigint`) → `boolean`
`pg_advisory_unlock_shared` (*key1* `integer`, *key2* `integer`) → `boolean`
Освобождает ранее полученную разделяемую рекомендательную блокировку сеансового уровня. Если она освобождена успешно, эта функция возвращает `true`, а если сеанс не владел этой блокировкой — `false`, при этом сервер выдаёт предупреждение SQL.

`pg_advisory_xact_lock` (*key* `bigint`) → `void`
`pg_advisory_xact_lock` (*key1* `integer`, *key2* `integer`) → `void`
Получает исключительную рекомендательную блокировку транзакционного уровня, ожидая её, если это необходимо.

`pg_advisory_xact_lock_shared` (*key* `bigint`) → `void`
`pg_advisory_xact_lock_shared` (*key1* `integer`, *key2* `integer`) → `void`
Получает разделяемую рекомендательную блокировку транзакционного уровня, ожидая её, если это необходимо.

`pg_try_advisory_lock` (*key* `bigint`) → `boolean`
`pg_try_advisory_lock` (*key1* `integer`, *key2* `integer`) → `boolean`
Получает исключительную рекомендательную блокировку сеансового уровня, если она доступна. То есть эта функция либо немедленно получает блокировку и возвращает `true`, либо сразу возвращает `false`, если получить её нельзя.

`pg_try_advisory_lock_shared` (*key* `bigint`) → `boolean`
`pg_try_advisory_lock_shared` (*key1* `integer`, *key2* `integer`) → `boolean`
Получает разделяемую рекомендательную блокировку сеансового уровня, если она доступна. То есть эта функция либо немедленно получает блокировку и возвращает `true`, либо сразу возвращает `false`, если получить её нельзя.

`pg_try_advisory_xact_lock` (*key* `bigint`) → `boolean`
`pg_try_advisory_xact_lock` (*key1* `integer`, *key2* `integer`) → `boolean`
Получает исключительную рекомендательную блокировку транзакционного уровня, если она доступна. То есть эта функция либо немедленно получает блокировку и возвращает `true`, либо сразу возвращает `false`, если получить её нельзя.

`pg_try_advisory_xact_lock_shared` (*key* `bigint`) → `boolean` `pg_try_advisory_xact_lock_shared` (*key1* `integer`, *key2* `integer`) → `boolean`

Функция **Описание**
Получает разделяемую рекомендательную блокировку транзакционного уровня, если она доступна. То есть эта функция либо немедленно получает блокировку и возвращает `true`, либо сразу возвращает `false`, если получить её нельзя.

9.28.11. Опорные функции журнала операций

В журнале операций хранится информация о критически важных системных событиях, таких как обновление, выполнение pg_resetwal и т. п. Эта информация не представляет особого интереса для обычного пользователя, но важна для осуществления технической поддержки со стороны поставщика. Запись в журнал операций производится только на системном уровне (без каких-либо действий со стороны пользователя), а для его чтения используются функции SQL. Функция, показанная в Таблица 9.107, читает журнал операций.

Таблица 9.107. Опорные функции журнала операций

Функция **Описание**
`pg_operation_log` () → `setof record` (*`event`* `text`, *`edition`* `text`, *`version`* `text`, *`lsn`* `pg_lsn` , *`last`* `timestamptz`, *`count`* `int4`) Здесь: • *`event`* — тип события (операции). Может быть `bootstrap`, `startup`, `pg_resetwal` , `pg_rewind` , `pg_upgrade` или `promoted`. • *`edition`* — редакция Postgres Pro. Может быть `vanilla`, `1c`, `std`, `ent` или `unknown`. • *`version`* — версия Postgres Pro. • *`lsn`* — положение последней контрольной точки, возвращённое утилитой pg_controldata на момент возникновения события. • *`last`* — дата/время возникновения последнего события этого типа, если события накапливаются, или дата/время возникновения события в противном случае. • *`count`* — количество событий этого типа, если события накапливаются, или 1 в противном случае. Эта системная функция создаётся автоматически для новых баз данных, а для существующих баз данных должна создаваться следующим образом:

```
CREATE OR REPLACE FUNCTION
  pg_operation_log(
    OUT event text,
    OUT edition text,
    OUT version text,
    OUT lsn pg_lsn,
    OUT last timestamptz,
    OUT count int4)
 RETURNS SETOF record
 LANGUAGE INTERNAL
 STABLE STRICT PARALLEL SAFE
AS 'pg_operation_log';
```

Ниже представлен пример вывода для функции `pg_operation_log`:

```
select * from pg_operation_log();
   event    | edition | version |    lsn    |          last          | count
------------+---------+---------+-----------+------------------------+-------

 startup    | vanilla | 10.22.0 | 0/8000028 | 2022-10-27 23:06:27+03 |     1
 pg_upgrade | 1c      | 15.0.0  | 0/8000028 | 2022-10-27 23:06:27+03 |     1
```

```
 startup    | 1c       | 15.0.0   | 0/80001F8 | 2022-10-27 23:06:53+03 |     2
(3 rows)
```

> **Примечание**
>
> Если журнал операций пуст, то при обновлении перед записью pg_upgrade сначала записывается событие `startup` с предыдущим значением *`version`*, имеющим следующие особенности: вместо версии патча (третья цифра в номере версии) всегда устанавливается ноль, поскольку в pg_upgrade нет информации о версии патча ни для одной редакции (`1c`, `std`, `ent`). При обновлении с `1c` или `vanilla` невозможно выяснить, к какой из этих редакций относится база данных до обновления, поэтому для записи `startup` используется редакция `vanilla`.

9.28.12. Отладочные функции

Функция, показанная в Таблице 9.108, может быть полезна для низкоуровневых операций, например в целях отладки или исследования испорченных баз данных Postgres Pro.

Таблица 9.108. Функции синхронизации снимков

Имя	Тип результата	Описание
`pg_snapshot_any()`	`void`	Отключает действие правил MVCC в рамках текущей транзакции, что позволяет видеть в ней все версии данных.

Применяйте `pg_snapshot_any` крайне осторожно. Выполнять её следует в транзакции с уровнем изоляции `REPEATABLE READ` или выше, в противном случае следующий же запрос заменит выбранный снимок новым. Выполнять её могут только суперпользователи.

> **Примечание**
>
> Если ваш кластер БД создавался сервером версии, в которой этой функции ещё не было, определите её следующей командой:
>
> ```
> CREATE FUNCTION pg_snapshot_any() RETURNS void AS 'pg_snapshot_any' LANGUAGE
>  internal;
> ```

9.28.13. Функции для устранения неполадок

Функции, перечисленные в Таблице 9.109, предназначены для выявления и диагностики проблем. Подробнее об их использовании можно узнать в Главе 31.

Таблица 9.109. Функции для устранения неполадок

Функция Описание
`pg_diagdump` (*`pid`* `integer`) → `integer` Выгружает состояние серверного процесса с указанным идентификатором. Эта функция обычно используется при устранении неполадок, когда процесс не отвечает или находится в бесконечном цикле. Вызывать эту функцию разрешено только суперпользователям.

9.29. Триггерные функции

Тогда как в большинстве случаев использование триггеров подразумевает написание пользовательских триггерных функций, в Postgres Pro имеется несколько встроенных триггерных функций, которые можно использовать непосредственно в пользовательских триггерах. Эти функции показаны в Таблице 9.110. (Существуют и другие встроенные триггерные функции, реализующие ограничения внешнего ключа и отложенные ограничения по индексам. Однако они не документированы, так как пользователям не нужно использовать их непосредственно.)

Подробнее о создании триггеров можно узнать в описании CREATE TRIGGER.

Таблица 9.110. Встроенные триггерные функции

Функция **Описание** **Пример использования**

`suppress_redundant_updates_trigger` ( ) → `trigger`
Предотвращает изменения, не меняющие данные. Подробнее об этом рассказывается ниже.
`CREATE TRIGGER ... suppress_redundant_updates_trigger()`

`tsvector_update_trigger` ( ) → `trigger`
Автоматически обновляет содержимое столбца `tsvector` из связанных столбцов с обычным текстовым содержимым. Конфигурация текстового поиска, которая будет использоваться, задаётся по имени в аргументе триггера. За подробностями обратитесь к Подразделу 12.4.3.
`CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)`

`tsvector_update_trigger_column` ( ) → `trigger`
Автоматически обновляет содержимое столбца `tsvector` из связанных столбцов с обычным текстовым содержимым. Конфигурация текстового поиска, которая будет использоваться, определяется содержимым столбца `regconfig` целевой таблицы. За подробностями обратитесь к Подразделу 12.4.3.
`CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, tsconfigcol, title, body)`

Функция `suppress_redundant_updates_trigger`, применяемая в качестве триггера `BEFORE UPDATE` на уровне строк, предотвратит внесение изменений, при которых данные в строке фактически не меняются. Тем самым переопределяется обычное поведение, когда изменение физической строки происходит вне зависимости от того, были ли изменены данные. (Обычное поведение не предполагает сравнения данных, поэтому операции изменения выполняются быстрее, и в ряде случаев именно это поведение желательно.)

В идеале следует избегать операций изменения, которые фактически не меняют данные в записях. Подобные ненужные изменения могут обходиться дорого, особенно когда требуется обновлять множество индексов, к тому же впоследствии базу данных придётся очищать от «мёртвых» строк. Однако выявить такие изменения в клиентском коде бывает сложно, если вообще возможно, а при составлении соответствующих проверочных выражений легко допустить ошибку. В качестве альтернативного решения можно использовать функцию `suppress_redundant_updates_trigger`, которая опускает изменения, не меняющие данные. Однако использовать её следует с осторожностью. Данный триггер выполняется для каждой записи довольно быстро, но всё же не мгновенно, так что если большинство затронутых записей фактически изменяется, с этим триггером операция изменения в среднем замедлится.

Функцию `suppress_redundant_updates_trigger` можно привязать к таблице так:

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE FUNCTION suppress_redundant_updates_trigger();
```

В большинстве случаев этот триггер должен вызываться для каждой строки последним, чтобы он не перекрыл другие триггеры, которым может понадобиться изменить строку. С учётом того, что триггеры вызываются по порядку сортировки их имён, имя для него нужно выбирать таким, чтобы оно было последним среди имён всех триггеров, которые могут быть в таблице. (Из этого соображения выбран префикс «z» в данном примере.)

9.30. Функции событийных триггеров

Postgres Pro предоставляет следующие вспомогательные функции для получения информации в событийных триггерах.

Подробнее о событийных триггерах можно узнать в Главе 38.

9.30.1. Получение изменений в конце команды

`pg_event_trigger_ddl_commands () → setof record`

Функция `pg_event_trigger_ddl_commands` возвращает список команд DDL, выполняемых в результате действия пользователя. Вызывать её можно только в функции, реализующей событийный триггер `ddl_command_end`. При попытке вызвать её в любом другом контексте возникнет ошибка. Функция `pg_event_trigger_ddl_commands` возвращает одну строку для каждой базовой команды; для некоторых команд, записываемых в виде одного предложения SQL, может возвращаться несколько строк. Эта функция возвращает следующие столбцы:

Имя	**Тип**	**Описание**
`classid`	`oid`	OID каталога, к которому относится объект
`objid`	`oid`	OID самого объекта
`objsubid`	`integer`	Идентификатор подобъекта (например, номер для столбца)
`command_tag`	`text`	Тег команды
`object_type`	`text`	Тип объекта
`schema_name`	`text`	Имя схемы, к которой относится объект; если объект не относится ни к какой схеме — `NULL`. В кавычки имя не заключается.
`object_identity`	`text`	Текстовое представление идентификатора объекта, включающее схему. При необходимости компоненты этого идентификатора заключаются в кавычки.
`in_extension`	`boolean`	True, если команда является частью скрипта расширения
`command`	`pg_ddl_command`	Полное представление команды, во внутреннем формате. Его нельзя вывести непосредственно, но можно передать другим функциям, чтобы получить различные сведения о команде.

9.30.2. Обработка объектов, удалённых командой DDL

`pg_event_trigger_dropped_objects () → setof record`

Функция `pg_event_trigger_dropped_objects` выдаёт список всех объектов, удалённых командой, для которых вызывалось событие `sql_drop`. При вызове в любом другом контексте происходит ошибка. Эта функция выдаёт следующие столбцы:

Имя	**Тип**	**Описание**
`classid`	`oid`	OID каталога, к которому относился объект
`objid`	`oid`	OID самого объекта
`objsubid`	`integer`	Идентификатор подобъекта (например, номер для столбца)
`original`	`boolean`	True, если это один из корневых удаляемых объектов
`normal`	`boolean`	True, если к этому объекту в графе зависимостей привело отношение обычной зависимости
`is_temporary`	`boolean`	True, если объект был временным
`object_type`	`text`	Тип объекта
`schema_name`	`text`	Имя схемы, к которой относился объект; если объект не относился ни к какой схеме — `NULL`. В кавычки имя не заключается.
`object_name`	`text`	Имя объекта, если сочетание схемы и имени позволяет уникально идентифицировать объект; в противном случае — `NULL`. Имя не заключается в кавычки и не дополняется именем схемы.
`object_identity`	`text`	Текстовое представление идентификатора объекта, включающее схему. При необходимости компоненты этого идентификатора заключаются в кавычки.
`address_names`	`text[]`	Массив, который в сочетании с `object_type` и массивом `address_args` можно передать функции `pg_get_object_address`, чтобы воссоздать адрес объекта на удалённом сервере, содержащем одноимённый объект того же рода.
`address_args`	`text[]`	Дополнение к массиву `address_names`

Функцию `pg_event_trigger_dropped_objects` можно использовать в событийном триггере так:

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
```

```
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END;
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
   ON sql_drop
   EXECUTE FUNCTION test_event_trigger_for_drops();
```

9.30.3. Обработка события перезаписи таблицы

В Таблице 9.111 показаны функции, выдающие информацию о таблице, для которой произошло событие перезаписи таблицы (`table_rewrite`). При попытке вызвать их в другом контексте возникнет ошибка.

Таблица 9.111. Функции получения информации о перезаписи таблицы

Функция **Описание**
`pg_event_trigger_table_rewrite_oid` () → `oid` Выдаёт OID таблицы, которая будет перезаписана.
`pg_event_trigger_table_rewrite_reason` () → `integer` Возвращает код с объяснением причины(причин) перезаписи в виде битовой карты, построенной из следующих значений: `1` (в таблице изменилась характеристика хранения отношений), `2` (в столбце изменилось значение по умолчанию), `4` (у столбца изменился тип данных) и `8` (изменился табличный метод доступа).

Эти функции можно использовать в событийном триггере так:

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
                pg_event_trigger_table_rewrite_oid()::regclass,
                pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
                  ON table_rewrite
   EXECUTE FUNCTION test_event_trigger_table_rewrite_oid();
```

9.31. Системные информационные функции

Postgres Pro предоставляет функцию для исследования сложной статистики, создаваемой командой `CREATE STATISTICS`.

9.31.1. Исследование списков MCV

`pg_mcv_list_items ( pg_mcv_list ) → setof record`

Функция `pg_mcv_list_items` возвращает набор записей, описывающих все элементы в многоколоночном списке MCV. Эти записи имеют следующие столбцы:

Имя	**Тип**	**Описание**
`index`	`integer`	индекс элемента в списке MCV
`values`	`text[]`	значения, сохранённые в элементе списка MCV
`nulls`	`boolean[]`	флаги, помечающие значения `NULL`
`frequency`	`double precision`	частота вхождения этого элемента MCV
`base_frequency`	`double precision`	базовая частота вхождения этого элемента MCV

Использовать функцию `pg_mcv_list_items` можно следующим образом:

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts';
```

Значения типа `pg_mcv_list` можно получить только из столбца `pg_statistic_ext_data.stxdmcv`.

Глава 10. Преобразование типов

SQL-операторы, намеренно или нет, требуют совмещать данные разных типов в одном выражении. Для вычисления подобных выражений со смешанными типами Postgres Pro предоставляет широкий набор возможностей.

Очень часто пользователю не нужно понимать все тонкости механизма преобразования. Однако следует учитывать, что неявные преобразования, производимые Postgres Pro, могут влиять на результат запроса. Поэтому при необходимости нужные результаты можно получить, применив *явное* преобразование типов.

В этой главе описываются общие механизмы преобразования типов и соглашения, принятые в Postgres Pro. За дополнительной информацией о конкретных типах данных и разрешённых для них функциях и операторах обратитесь к соответствующим разделам в Главе 8 и Главе 9.

10.1. Обзор

SQL — язык со строгой типизацией. То есть каждый элемент данных в нём имеет некоторый тип, определяющий его поведение и допустимое использование. Postgres Pro наделён расширяемой системой типов, более универсальной и гибкой по сравнению с другими реализациями SQL. При этом преобразования типов в Postgres Pro в основном подчиняются определённым общим правилам, для их понимания не нужен эвристический анализ. Благодаря этому в выражениях со смешанными типами можно использовать даже типы, определённые пользователями.

Анализатор выражений Postgres Pro разделяет их лексические элементы на пять основных категорий: целые числа, другие числовые значения, текстовые строки, идентификаторы и ключевые слова. Константы большинства не числовых типов сначала классифицируются как строки. В определении языка SQL допускается указывать имена типов в строках и это можно использовать в Postgres Pro, чтобы направить анализатор по верному пути. Например, запрос:

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

содержит две строковых константы, типа `text` и типа `point`. Если для такой константы не указан тип, для неё первоначально предполагается тип `unknown`, который затем может быть уточнён, как описано ниже.

В SQL есть четыре фундаментальных фактора, определяющих правила преобразования типов для анализатора выражений Postgres Pro:

Вызовы функций

> Система типов Postgres Pro во многом построена как дополнение к богатым возможностям функций. Функции могут иметь один или несколько аргументов, и при этом Postgres Pro разрешает перегружать имена функций, так что имя функции само по себе не идентифицирует вызываемую функцию; анализатор выбирает правильную функцию в зависимости от типов переданных аргументов.

Операторы

> Postgres Pro позволяет использовать в выражениях префиксные операторы (с одним аргументом), а также инфиксные операторы (с двумя аргументами). Как и функции, операторы можно перегружать, так что и с ними существует проблема выбора правильного оператора.

Сохранение значений

> SQL-операторы `INSERT` и `UPDATE` помещают результаты выражений в таблицы. При этом получаемые значения должны соответствовать типам целевых столбцов или, возможно, приводиться к ним.

`UNION`, `CASE` и связанные конструкции

Так как все результаты запроса объединяющего оператора `SELECT` должны оказаться в одном наборе столбцов, результаты каждого подзапроса `SELECT` должны приводиться к одному набору типов. Подобным образом, результирующие выражения конструкции `CASE` должны приводиться к общему типу, так как выражение `CASE` в целом должно иметь определённый выходной тип. Подобное определение общего типа для значений нескольких подвыражений требуется и для некоторых других конструкций, например `ARRAY[]`, а также для функций `GREATEST` и `LEAST`.

Информация о существующих преобразованиях или *приведениях* типов, для каких типов они определены и как их выполнять, хранится в системных каталогах. Пользователь также может добавить дополнительные преобразования с помощью команды CREATE CAST. (Обычно это делается, когда определяются новые типы данных. Набор приведений для встроенных типов достаточно хорошо проработан, так что его лучше не менять.)

Дополнительная логика анализа помогает выбрать оптимальное приведение в группах типов, допускающих неявные преобразования. Для этого типы данных разделяются на несколько базовых *категорий*, которые включают: `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network` и пользовательские типы. (Полный список категорий приведён в Таблице 51.67; хотя его тоже можно расширить, определив свои категории.) В каждой категории могут быть выбраны один или несколько *предпочитаемых типов*, которые будут считаться наиболее подходящими при рассмотрении нескольких вариантов. Аккуратно выбирая предпочитаемые типы и допустимые неявные преобразования, можно добиться того, что выражения с неоднозначностями (в которых возможны разные решения задачи преобразования) будут разрешаться наилучшим образом.

Все правила преобразования типов разработаны с учётом следующих принципов:

- Результат неявных преобразованиях всегда должен быть предсказуемым и понятным.
- Если в неявном преобразовании нет нужды, анализатор и исполнитель запроса не должны тратить лишнее время на это. То есть, если запрос хорошо сформулирован и типы значений совпадают, он должен выполняться без дополнительной обработки в анализаторе и без лишних вызовов неявных преобразований.
- Кроме того, если запрос изначально требовал неявного преобразования для функции, а пользователь определил новую функцию с точно совпадающими типами аргументов, анализатор должен переключиться на новую функцию и больше не выполнять преобразование для вызова старой.

10.2. Операторы

При выборе конкретного оператора, задействованного в выражении, PostgreSQL следует описанному ниже алгоритму. Заметьте, что на этот выбор могут неявно влиять приоритеты остальных операторов в данном выражении, так как они определяют, какие подвыражения будут аргументами операторов. Подробнее об этом рассказывается в Подразделе 4.1.6.

Выбор оператора по типу

1. Выбрать операторы для рассмотрения из системного каталога `pg_operator`. Если имя оператора не дополнено именем схемы (обычно это так), будут рассматриваться все операторы с подходящим именем и числом аргументов, видимые в текущем пути поиска (см. Подраздел 5.10.3). Если имя оператора определено полностью, в рассмотрение принимаются только операторы из указанной схемы.

    - (Optional) Если в пути поиска оказывается несколько операторов с одинаковыми типами аргументов, учитываются только те из них, которые находятся в пути раньше. Операторы с разными типами аргументов рассматриваются на равных правах вне зависимости от их положения в пути поиска.

2. Проверить, нет ли среди них оператора с точно совпадающими типами аргументов. Если такой оператор есть (он может быть только одним в отобранном ранее наборе), использовать его. Отсутствие точного совпадения создаёт угрозу вызова с указанием полного имени [1] (нетипичным) любого оператора, который может оказаться в схеме, где могут создавать объекты недоверенные пользователи. В таких ситуациях приведите типы аргументов для получения точного совпадения.
    - a. (Optional) Если один аргумент при вызове бинарного оператора имеет тип `unknown`, для данной проверки предполагается, что он имеет тот же тип, что и второй его аргумент. При вызове бинарного оператора с двумя аргументами `unknown` или префиксного с одним `unknown` оператор не будет выбран на этом шаге.
    - b. (Optional) Если один аргумент при вызове бинарного оператора имеет тип `unknown`, а другой — домен, проверить, есть ли оператор, принимающий базовый тип домена с обеих сторон; если таковой находится, использовать его.
3. Найти самый подходящий.
    - a. Отбросить кандидатов, для которых входные типы не совпадают и не могут быть преобразованы (неявным образом) так, чтобы они совпали. В данном случае считается, что константы типа `unknown` можно преобразовать во что угодно. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.
    - b. Если один из аргументов имеет тип домен, далее считать его типом базовый тип домена. Благодаря этому при поиске неоднозначно заданного оператора домены будут подобны свои базовым типам.
    - c. Просмотреть всех кандидатов и оставить только тех, для которых точно совпадают как можно больше типов аргументов. Оставить всех кандидатов, если точных совпадений нет. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.
    - d. Просмотреть всех кандидатов и оставить только тех, которые принимают предпочитаемые типы (из категории типов входных значений) в наибольшем числе позиций, где требуется преобразование типов. Оставить всех кандидатов, если ни один не принимает предпочитаемые типы. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.
    - e. Если какие-либо значения имеют тип `unknown`, проверить категории типов, принимаемых в данных позициях аргументов оставшимися кандидатами. Для каждой позиции выбрать категорию `string`, если какой-либо кандидат принимает эту категорию. (Эта склонность к строкам объясняется тем, что константа типа unknown выглядит как строка.) Если эта категория не подходит, но все оставшиеся кандидаты принимают одну категорию, выбрать её; в противном случае констатировать неудачу — сделать правильный выбор без дополнительных подсказок нельзя. Затем отбросить кандидатов, которые не принимают типы выбранной категории. Далее, если какой-либо кандидат принимает предпочитаемый тип из этой категории, отбросить кандидатов, принимающих другие, не предпочитаемые типы для данного аргумента. Оставить всех кандидатов, если эти проверки не прошёл ни один. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.
    - f. Если в списке аргументов есть аргументы и типа `unknown`, и известного типа, и этот известный тип один для всех аргументов, предположить, что аргументы типа `unknown` также имеют этот тип, и проверить, какие кандидаты могут принимать этот тип в позиции аргумента `unknown`. Если остаётся только один кандидат, использовать его, в противном случае констатировать неудачу.

Ниже это проиллюстрировано на примерах.

---

[1] Эта угроза неактуальна для имён без схемы, так как путь поиска, содержащий схемы, в которых недоверенные пользователи могут создавать объекты, не соответствует шаблону безопасного использования схем.

Пример 10.1. Разрешение типа для оператора квадратного корня

В стандартном каталоге определён только один оператор квадратного корня (префиксный `|/`) и он принимает аргумент типа `double precision`. При просмотре следующего выражения его аргументу изначально назначается тип `integer`:

```
SELECT |/ 40 AS "square root of 40";
 square root of 40
-------------------
 6.324555320336759
(1 row)
```

Поэтому анализатор преобразует тип этого операнда и запрос становится равносильным такому:

```
SELECT |/ CAST(40 AS double precision) AS "square root of 40";
```

Пример 10.2. Разрешение оператора конкатенации строк

Синтаксис текстовых строк используется как для записи строковых типов, так и для сложных типов расширений. Если тип не указан явно, такие строки сопоставляются по тому же алгоритму с наиболее подходящими операторами.

Пример с одним неопределённым аргументом:

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

В этом случае анализатор смотрит, есть ли оператор, у которого оба аргумента имеют тип `text`. Такой оператор находится, поэтому предполагается, что второй аргумент следует воспринимать как аргумент типа `text`.

Конкатенация двух значений неопределённых типов:

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

В данном случае нет подсказки для выбора типа, так как в данном запросе никакие типы не указаны. Поэтому анализатор просматривает все возможные операторы и находит в них кандидатов, принимающих аргументы категорий string и bit-string. Так как категория string является предпочтительной, выбирается она, а затем для разрешения типа не типизированной константы выбирается предпочтительный тип этой категории, `text`.

Пример 10.3. Разрешение оператора абсолютного значения и отрицания

В каталоге операторов Postgres Pro для префиксного оператора `@` есть несколько записей, описывающих операции получения абсолютного значения для различных числовых типов данных. Одна из записей соответствует типу `float8`, предпочтительного в категории числовых типов. Таким образом, столкнувшись со значением типа `unknown`, Postgres Pro выберет эту запись:

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

Здесь система неявно привела константу неизвестного типа к типу `float8`, прежде чем применять выбранный оператор. Можно убедиться в том, что выбран именно тип `float8`, а не какой-то другой:

```
SELECT @ '-4.5e500' AS "abs";

ОШИБКА:  "-4.5e500" вне диапазона для типа double precision
```

С другой стороны, префиксный оператор ~ (побитовое отрицание) определён только для целочисленных типов данных, но не для `float8`. Поэтому, если попытаться выполнить похожий запрос с ~, мы получаем:

```
SELECT ~ '20' AS "negation";

ОШИБКА: оператор не уникален: ~ "unknown"
ПОДСКАЗКА: Не удалось выбрать лучшую кандидатуру оператора. Возможно, вам следует
добавить явные преобразования типов.
```

Это происходит оттого, что система не может решить, какой оператор предпочесть из нескольких возможных вариантов ~. Мы можем облегчить её задачу, добавив явное преобразование:

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

Пример 10.4. Разрешение оператора включения в массив

Ещё один пример разрешения оператора с одним аргументом известного типа и другим неизвестного:

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

В каталоге операторов Postgres Pro есть несколько записей для инфиксного оператора `<@`, но только два из них могут принять целочисленный массива слева: оператор включения массива (`anyarray<@anyarray`) и оператор включения диапазона (`anyelement<@anyrange`). Так как ни один из этих полиморфных псевдотипов (см. Раздел 8.21) не считается предпочтительным, анализатор не может избавиться от неоднозначности на данном этапе. Однако в Шаг 3.f говорится, что константа неизвестного типа должна рассматриваться как значение типа другого аргумента, в данном случае это целочисленный массив. После этого подходящим считается только один из двух операторов, так что выбирается оператор с целочисленными массивами. (Если бы был выбран оператор включения диапазона, мы получили бы ошибку, так как значение в строке не соответствует формату значений диапазона.)

Пример 10.5. Нестандартный оператор с доменом

Иногда пользователи пытаются ввести операторы, применимые только к определённому домену. Это возможно, но вовсе не так полезно, как может показаться, ведь правила разрешения операторов применяются к базовому типу домена. Взгляните на этот пример:

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean AS ...;
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext, rightarg=text);
CREATE TABLE mytable (val mytext);

SELECT * FROM mytable WHERE val = 'foo';
```

В этом запросе не будет использоваться нововведённый оператор. При разборе запроса сначала будет проверено, есть ли оператор `mytext = mytext` (см. Шаг 2.a), но это не так; затем будет рассмотрен базовый тип домена (`text`) и проверено наличие оператора `text = text` (см. Шаг 2.b), и таковой действительно есть; в итоге строковое значение типа `unknown` будет воспринято как `text` и будет применён оператор `text = text`. Единственный вариант задействовать нововведённый оператор — добавить явное приведение:

```
SELECT * FROM mytable WHERE val = text 'foo';
```

так, чтобы оператор `mytext = text` был найден сразу, согласно правилу точного совпадения. Если дело доходит до правил наибольшего соответствия, они активно дискредитируют операторы доменных типов. Если бы они этого не делали, с таким оператором возникало бы слишком много ошибок разрешения операторов, потому что правила приведения всегда считают домен приводимым к базовому типу и наоборот, так что доменный оператор применялся бы во всех случаях, где применяется одноимённый оператор с базовым типом.

10.3. Функции

При выборе конкретной функции, задействованной в выражении, PostgreSQL следует описанному ниже алгоритму.

Разрешение функции по типу

1. Выбрать функции для рассмотрения из системного каталога `pg_proc`. Если имя функции не дополнено именем схемы, будут рассматриваться все функции с подходящим именем и числом аргументов, видимые в текущем пути поиска (см. Подраздел 5.10.3). Если имя функции определено полностью, в рассмотрение принимаются только функции из указанной схемы.

    a. (Optional) Если в пути поиска оказывается несколько функций с одинаковыми типами аргументов, учитываются только те из них, которые находятся в пути раньше. Функции с разными типами аргументов рассматриваются на равных правах вне зависимости от их положения в пути поиска.

    b. (Optional) Если в числе параметров функции есть массив `VARIADIC` и при вызове не указывается ключевое слово `VARIADIC`, функция обрабатывается, как если бы этот параметр был заменён одним или несколькими параметрами типа элементов массива, по числу аргументов при вызове. После такого расширения по фактическим типам аргументов она может совпасть с некоторой функцией с постоянным числом аргументов. В этом случае используется функция, которая находится в пути раньше, а если они оказываются в одной схеме, предпочитается вариант с постоянными аргументами.

       Это создаёт угрозу безопасности при вызове с полным именем [2] функции с переменным числом аргументов, которая может оказаться в схеме, где могут создавать объекты недоверенные пользователи. Злонамеренный пользователь может перехватывать управление и выполнять произвольные SQL-функции, как будто их выполняете вы. Запись вызова с ключевым словом `VARIADIC` устраняет эту угрозу. Однако для вызовов с передачей параметров `VARIADIC "any"` часто не существует необходимой формулировки с ключом `VARIADIC`. Чтобы такие вызовы были безопасными, создание объектов в схеме функции должно разрешаться только доверенным пользователям.

    c. (Optional) Функции, для которых определены значения параметров по умолчанию, считаются совпадающими с вызовом, в котором опущено ноль или более параметров в соответствующих позициях. Если для вызова подходят несколько функций, используется та, что обнаруживается в пути поиска раньше. Если в одной схеме оказываются несколько функций с одинаковыми типами в позициях обязательных параметров (что возможно, если в них определены разные наборы пропускаемых параметров), система не сможет выбрать оптимальную, и выдаст ошибку «неоднозначный вызов функции», если лучшее соответствие для вызова не будет найдено.

       Это создаёт угрозу при вызове с полным именем [2] любой функции, которая может оказаться в схеме, где могут создавать объекты недоверенные пользователи. Злонамеренный пользователь может создать функцию с именем уже существующей, продублировав параметры исходной и добавив дополнительные со значениями по умолчанию. В результате при по-

---

[2] Эта угроза неактуальна для имён без схемы, так как путь поиска, содержащий схемы, в которых недоверенные пользователи могут создавать объекты, не соответствует шаблону безопасного использования схем.

следующих вызовах будет выполняться не исходная функция. Для ликвидации этой угрозы помещайте функции в схемы, в которых создавать объекты могут только доверенные объекты.

2. Проверить, нет ли функции, принимающей в точности типы входных аргументов. Если такая функция есть (она может быть только одной в отобранном ранее наборе), использовать её. Отсутствие точного совпадения создаёт угрозу вызова с полным именем [2] функции в схеме, где могут создавать объекты недоверенные пользователи. В таких ситуациях приведите типы аргументов для получения точного соответствия. (В случаях с `unknown` совпадения на этом этапе не будет никогда.)

3. Если точное совпадение не найдено, проверить, не похож ли вызов функции на особую форму преобразования типов. Это имеет место, когда при вызове функции передаётся всего один аргумент и имя функции совпадает с именем (внутренним) некоторого типа данных. Более того, аргументом функции должна быть либо строка неопределённого типа, либо значение типа, двоично-совместимого с указанным или приводимого к нему с помощью функций ввода-вывода типа (то есть, преобразований в стандартный строковый тип и обратно). Если эти условия выполняются, вызов функции воспринимается как особая форма конструкции `CAST`. [3]

4. Найти самый подходящий.

   a. Отбросить кандидатов, для которых входные типы не совпадают и не могут быть преобразованы (неявным образом) так, чтобы они совпали. В данном случае считается, что константы типа `unknown` можно преобразовать во что угодно. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

   b. Если один из аргументов имеет тип домен, далее считать его типом базовый тип домена. Благодаря этому при поиске неоднозначно заданной функции домены будут подобны свои базовым типам.

   c. Просмотреть всех кандидатов и оставить только тех, для которых точно совпадают как можно больше типов аргументов. Оставить всех кандидатов, если точных совпадений нет. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

   d. Просмотреть всех кандидатов и оставить только тех, которые принимают предпочитаемые типы (из категории типов входных значений) в наибольшем числе позиций, где требуется преобразование типов. Оставить всех кандидатов, если ни один не принимает предпочитаемые типы. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

   e. Если какие-либо значения имеют тип `unknown`, проверить категории типов, принимаемых в данных позициях аргументов оставшимися кандидатами. Для каждой позиции выбрать категорию `string`, если какой-либо кандидат принимает эту категорию. (Эта склонность к строкам объясняется тем, что константа типа unknown выглядит как строка.) Если эта категория не подходит, но все оставшиеся кандидаты принимают одну категорию, выбрать её; в противном случае констатировать неудачу — сделать правильный выбор без дополнительных подсказок нельзя. Затем отбросить кандидатов, которые не принимают типы выбранной категории. Далее, если какой-либо кандидат принимает предпочитаемый тип из этой категории, отбросить кандидатов, принимающих другие, не предпочитаемые типы для данного аргумента. Оставить всех кандидатов, если эти проверки не прошёл ни один. Если остаётся только один кандидат, использовать его, в противном случае перейти к следующему шагу.

   f. Если в списке аргументов есть аргументы и типа `unknown`, и известного типа, и этот известный тип один для всех аргументов, предположить, что аргументы типа `unknown` также имеют этот тип, и проверить, какие кандидаты могут принимать этот тип в позиции аргумента `unknown`. Если остаётся только один кандидат, использовать его, в противном случае констатировать неудачу.

---

[3] Этот шаг нужен для поддержки приведений типов в стиле вызова функции, когда на самом деле соответствующей функции приведения нет. Если такая функция приведения есть, она обычно называется именем выходного типа и необходимости в особом подходе нет. За дополнительными комментариями обратитесь к CREATE CAST.

Заметьте, что для функций действуют те же правила «оптимального соответствия», что и для операторов. Они проиллюстрированы следующими примерами.

Пример 10.6. Разрешение функции округления по типам аргументов

В PostgreSQL есть только одна функция `round`, принимающая два аргумента: первый типа `numeric`, а второй — `integer`. Поэтому в следующем запросе первый аргумент `integer` автоматически приводится к типу `numeric`:

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

Таким образом, анализатор преобразует этот запрос в:

```
SELECT round(CAST (4 AS numeric), 4);
```

Так как числовые константы с десятичными точками изначально относятся к типу `numeric`, для следующего запроса преобразование типов не потребуется, так что он немного эффективнее:

```
SELECT round(4.0, 4);
```

Пример 10.7. Разрешение функций с переменными параметрами

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

Эта функция принимает в аргументах ключевое слово VARIADIC, но может вызываться и без него. Ей можно передавать и целочисленные, и любые числовые аргументы:

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
(1 row)
```

Однако для первого и второго вызова предпочтительнее окажутся специализированные функции, если таковые есть:

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

Если используется конфигурация по умолчанию и существует только первая функция, первый и второй вызовы будут небезопасными. Любой пользователь может перехватить их, создав вторую или третью функцию. Безопасным будет третий вызов, в котором тип аргумента соответствует в точности и используется ключевое слово `VARIADIC`.

Пример 10.8. Разрешение функции извлечения подстроки

В PostgreSQL есть несколько вариантов функции `substr`, и один из них принимает аргументы типов `text` и `integer`. Если эта функция вызывается со строковой константой неопределённого типа, система выбирает функцию, принимающую аргумент предпочитаемой категории `string` (а конкретнее, типа `text`).

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

Если текстовая строка имеет тип `varchar`, например когда данные поступают из таблицы, анализатор попытается привести её к типу `text`:

```
SELECT substr (varchar '1234', 3);

 substr
--------
     34
(1 row)
```

Этот запрос анализатор фактически преобразует в:

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

Примечание

Анализатор узнаёт из каталога `pg_cast`, что типы `text` и `varchar` двоично-совместимы, что означает, что один тип можно передать функции, принимающей другой, не выполняя физического преобразования. Таким образом, в данном случае операция преобразования на самом не добавляется.

И если функция вызывается с аргументом типа `integer`, анализатор попытается преобразовать его в тип `text`:

```
SELECT substr(1234, 3);
ОШИБКА: функция substr(integer, integer) не существует
ПОДСКАЗКА: Функция с данными именем и типами аргументов не найдена. Возможно, вам
следует добавить явные преобразования типов.
```

Этот вариант не работает, так как `integer` нельзя неявно преобразовать в `text`. Однако с явным преобразованием запрос выполняется:

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

10.4. Хранимое значение

Значения, вставляемые в таблицу, преобразуется в тип данных целевого столбца по следующему алгоритму.

Преобразование по типу хранения

1. Проверить точное совпадение с целевым типом.

2. Если типы не совпадают, попытаться привести тип к целевому. Это возможно, если в каталоге `pg_cast` (см. CREATE CAST) зарегистрировано *приведение присваивания* между двумя типами. Если же результат выражения — строка неизвестного типа, содержимое этой строки будет подано на вход процедуре ввода целевого типа.
3. Проверить, не требуется ли приведение размера для целевого типа. Приведение размера — это преобразование типа к такому же. Если это приведение описано в каталоге `pg_cast`, применить к его к результату выражения, прежде чем сохранить в целевом столбце. Функция, реализующая такое приведение, всегда принимает дополнительный параметр типа `integer`, в котором передаётся значение `atttypmod` для целевого столбца (обычно это её объявленный размер, хотя интерпретироваться значение `atttypmod` для разных типов данных может по-разному), и третий параметр типа `boolean`, передающий признак явное/неявное преобразование. Функция приведения отвечает за все операции с длиной, включая её проверку и усечение данных.

Пример 10.9. Преобразование для типа хранения `character`

Следующие запросы показывают, что сохраняемое значение подгоняется под размер целевого столбца, объявленного как `character(20)`:

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

Суть происходящего здесь в том, что две константы неизвестного типа по умолчанию воспринимаются как значения `text`, что позволяет применить к ним оператор `||` как оператор конкатенации значений `text`. Затем результат оператора, имеющий тип `text`, приводится к типу `bpchar` («blank-padded char» (символы, дополненные пробелами), внутреннее имя типа `character`) в соответствии с типом целевого столбца. (Так как типы `text` и `bpchar` двоично-совместимы, при этом преобразовании реальный вызов функции не добавляется.) Наконец, в системном каталоге находится функция изменения размера `bpchar(bpchar, integer, boolean)` и применяется для результата оператора и длины столбца. Эта связанная с типом функция проверяет длину данных и добавляет недостающие пробелы.

10.5. `UNION`, `CASE` и связанные конструкции

SQL-конструкция `UNION` взаимодействует с системой типов, так как ей приходится объединять значения возможно различных типов в единый результирующий набор. Алгоритм разрешения типов при этом применяется независимо к каждому выходному столбцу запроса. Конструкции `INTERSECT` и `EXCEPT` сопоставляют различные типы подобно `UNION`. По такому же алгоритму сопоставляют типы выражений и определяют тип своего результата некоторые другие конструкции, включая `CASE`, `ARRAY`, `VALUES` и функции `GREATEST` и `LEAST`.

Разрешение типов для `UNION`, `CASE` и связанных конструкций

1. Если все данные одного типа и это не тип `unknown`, выбрать его.
2. Если тип данных — домен, далее считать их типом базовый тип домена. [4]
3. Если все данные типа `unknown`, выбрать для результата тип `text` (предпочитаемый для категории string). В противном случае значения `unknown` для остальных правил игнорируются.
4. Если известные типы входных данных оказываются не из одной категории, констатировать неудачу.

---

[4] Так же, как домены воспринимаются при выборе операторов и функций, доменные типы могут сохраняться в конструкции `UNION` или подобной, если пользователь позаботится о том, чтобы все входные данные приводились к этому типу явно или неявно. В противном случае будет использоваться базовый тип домена.

5. Выбрать первый известный тип данных в качестве типа-кандидата, затем рассмотреть все остальные известные типы данных, слева направо. [5] Если ранее выбранный тип может быть неявно преобразован к другому типу, но преобразовать второй в первый нельзя, выбрать второй тип в качестве нового кандидата. Затем продолжать рассмотрение последующих данных. Если на любом этапе этого процесса выбирается предпочитаемый тип, следующие данные больше не рассматриваются.
6. Привести все данные к окончательно выбранному типу. Констатировать неудачу, если неявное преобразование из типа входных данных в выбранный тип невозможно.

Ниже это проиллюстрировано на примерах.

Пример 10.10. Разрешение типов с частичным определением в Union

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

В данном случае константа `'b'` неизвестного типа будет преобразована в тип `text`.

Пример 10.11. Разрешение типов в простом объединении

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

Константа `1.2` имеет тип `numeric` и целочисленное значение `1` может быть неявно приведено к типу `numeric`, так что используется этот тип.

Пример 10.12. Разрешение типов в противоположном объединении

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

Здесь значение типа `real` нельзя неявно привести к `integer`, но `integer` можно неявно привести к `real`, поэтому типом результата объединения будет `real`.

Пример 10.13. Разрешение типов во вложенном объединении

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR:  UNION types text and integer cannot be matched
```

Эта ошибка возникает из-за того, что Postgres Pro воспринимает множественные `UNION` как пары с вложенными операциями, то есть как запись

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

Внутренний `UNION` разрешается как выдающий тип `text`, согласно правилам, приведённым выше. Затем внешний `UNION` получает на вход типы `text` и `integer`, что и приводит к показанной ошибке.

---

[5] По историческим причинам в конструкции `CASE` выражение в предложении `ELSE` (если оно есть) обрабатывается как «первое», а предложения `THEN` рассматриваются после. Во всех остальных случаях, «слева направо» означает порядок, в котором выражения действительно идут в тексте запроса.

Эту проблему можно устранить, сделав так, чтобы у самого левого `UNION` минимум с одной стороны были данные желаемого типа результата.

Операции `INTERSECT` и `EXCEPT` также разрешаются по парам. Однако остальные конструкции, описанные в этом разделе, рассматривают все входные данные сразу.

10.6. Выходные столбцы `SELECT`

Правила, описанные в предыдущих разделах, распространяются на присваивание типов данных, кроме `unknown`, во всех выражениях в запросах SQL, за исключением бестиповых буквальных значений, принимающих вид простых выходных столбцов команды `SELECT`. Например, в запросе

```
SELECT 'Hello World';
```

ничто не говорит о том, какой тип должно принимать строковое буквальное значение. В этой ситуации Postgres Pro разрешит тип такого значения как `text`.

Когда `SELECT` является одной из ветвей конструкции `UNION` (или `INTERSECT`/`EXCEPT`) или когда он находится внутри `INSERT ... SELECT`, это правило не действует, так как более высокий приоритет имеют правила, описанные в предыдущих разделах. В первом случае тип бестипового буквального значения может быть получен из другой ветви `UNION`, а во втором — из целевого столбца.

Списки `RETURNING` в данном контексте воспринимаются так же, как выходные списки `SELECT`.

Примечание

До Postgres Pro 10 этого правила не было и бестиповые буквальные значения в выходном списке `SELECT` оставались с типом `unknown`. Это имело различные негативные последствия, так что было решено это изменить.

Глава 11. Индексы

Индексы — это традиционное средство увеличения производительности БД. Используя индекс, сервер баз данных может находить и извлекать нужные строки гораздо быстрее, чем без него. Однако с индексами связана дополнительная нагрузка на СУБД в целом, поэтому применять их следует обдуманно.

11.1. Введение

Предположим, что у нас есть такая таблица:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

и приложение выполняет много подобных запросов:

```
SELECT content FROM test1 WHERE id = константа;
```

Если система не будет заранее подготовлена, ей придётся сканировать всю таблицу `test1`, строку за строкой, чтобы найти все подходящие записи. Когда таблица `test1` содержит большое количество записей, а этот запрос должен вернуть всего несколько (возможно, одну или ноль), такое сканирование, очевидно, неэффективно. Но если создать в системе индекс по полю `id`, она сможет находить строки гораздо быстрее. Возможно, для этого ей понадобится опуститься всего на несколько уровней в дереве поиска.

Подобный подход часто используется в технической литературе: термины и понятия, которые могут представлять интерес, собираются в алфавитном указателе в конце книги. Читатель может просмотреть этот указатель довольно быстро и затем перейти сразу к соответствующей странице, вместо того, чтобы пролистывать всю книгу в поисках нужного материала. Так же, как задача автора предугадать, что именно будут искать в книге читатели, задача программиста баз данных — заранее определить, какие индексы будут полезны.

Создать индекс для столбца `id` рассмотренной ранее таблицы можно с помощью следующей команды:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Имя индекса `test1_id_index` может быть произвольным, главное, чтобы оно позволяло понять, для чего этот индекс.

Для удаления индекса используется команда `DROP INDEX`. Добавлять и удалять индексы можно в любое время.

Когда индекс создан, никакие дополнительные действия не требуются: система сама будет обновлять его при изменении данных в таблице и сама будет использовать его в запросах, где, по её мнению, это будет эффективнее, чем сканирование всей таблицы. Вам, возможно, придётся только периодически запускать команду `ANALYZE` для обновления статистических данных, на основе которых планировщик запросов принимает решения. В Главе 14 вы можете узнать, как определить, используется ли определённый индекс и при каких условиях планировщик может решить *не* использовать его.

Индексы могут быть полезны также при выполнении команд `UPDATE` и `DELETE` с условиями поиска. Кроме того, они могут применяться в поиске с соединением. То есть, индекс, определённый для столбца, участвующего в условии соединения, может значительно ускорить запросы с JOIN.

В целом индексы Postgres Pro можно использовать для оптимизации запросов, содержащих одно или несколько предложений `WHERE` или `JOIN` в таком виде:

```
индексированный-столбец индексируемый-оператор значение-сравнения
```

Здесь *`индексированный-столбец`* — это любой столбец или выражение, для которого был определён индекс, *`индексируемый-оператор`* — это оператор, который является членом *класса операторов* индекса для индексированного столбца (более подробную информацию об этом см. ниже), а *`значение-сравнения`* может быть любым выражением, которое не является изменчивым и не ссылается на таблицу с этим индексом.

В некоторых случаях планировщик запросов может извлечь индексируемое предложение этого вида из другой конструкции SQL. Например, если исходное предложение выглядело так:

```
значение сравнения оператор индексированный столбец
```

, то его можно преобразовать в индексируемую форму, если исходный *`оператор`* имеет коммутирующий оператор, который является членом класса операторов индекса.

Создание индекса для большой таблицы может занимать много времени. По умолчанию Postgres Pro позволяет параллельно с созданием индекса выполнять чтение (операторы `SELECT`) таблицы, но операции записи (`INSERT`, `UPDATE` и `DELETE`) блокируются до окончания построения индекса. Для производственной среды это ограничение часто бывает неприемлемым. Хотя есть возможность разрешить запись параллельно с созданием индексов, при этом нужно учитывать ряд оговорок — они описаны в подразделе Неблокирующее построение индексов.

После создания индекса система должна поддерживать его в состоянии, соответствующем данным таблицы. С этим связаны неизбежные накладные расходы при изменении данных. Наличие индексов также может препятствовать созданию кортежей только в куче. Таким образом, индексы, которые используются в запросах редко или вообще никогда, должны быть удалены.

11.2. Типы индексов

Postgres Pro поддерживает несколько типов индексов: B-дерево, хеш, GiST, SP-GiST, GIN, BRIN и расширение bloom. Для разных типов индексов применяются разные алгоритмы, ориентированные на определённые типы индексируемых предложений. По умолчанию команда `CREATE INDEX` создаёт индексы-B-деревья, эффективные в большинстве случаев. Выбрать другой тип можно, написав название типа индекса после ключевого слова `USING`. Например, создать хеш-индекс можно так:

```
CREATE INDEX имя ON таблица USING HASH (столбец);
```

11.2.1. B-дерево

B-деревья могут работать в условиях на равенство и в проверках диапазонов с данными, которые можно отсортировать в некотором порядке. Точнее, планировщик запросов Postgres Pro может задействовать индекс-B-дерево, когда индексируемый столбец участвует в сравнении с одним из следующих операторов:

```
<    <=    =    >=    >
```

При обработке конструкций, представимых как сочетание этих операторов, например `BETWEEN` и `IN`, так же может выполняться поиск по индексу-B-дереву. Кроме того, такие индексы могут использоваться и в условиях `IS NULL` и `IS NOT NULL` по индексированным столбцам.

Также оптимизатор может использовать эти индексы в запросах с операторами сравнения по шаблону `LIKE` и `~`, *если* этот шаблон определяется константой и он привязан к началу строки — например, `col LIKE 'foo%'` или `col ~ '^foo'`, но не `col LIKE '%bar'`. Но если ваша база данных использует не локаль C, для поддержки индексирования запросов с шаблонами вам потребуется создать индекс со специальным классом операторов; см. Раздел 11.10. Индексы-B-деревья можно использовать и для `ILIKE` и `~*`, но только если шаблон начинается не с алфавитных символов, то есть символов, не подверженных преобразованию регистра.

B-деревья могут также применяться для получения данных, отсортированных по порядку. Это не всегда быстрее простого сканирования и сортировки, но иногда бывает полезно.

11.2.2. Хеш

Хеш-индексы хранят 32-битный хеш-код, полученный из значения индексированного столбца, поэтому хеш-индексы работают только с простыми условиями равенства. Планировщик запросов может применить хеш-индекс, только если индексируемый столбец участвует в сравнении с оператором =. Создать такой индекс можно следующей командой:

```
CREATE INDEX имя ON таблица USING HASH (столбец);
```

11.2.3. GiST

GiST-индексы представляют собой не просто разновидность индексов, а инфраструктуру, позволяющую реализовать много разных стратегий индексирования. Как следствие, GiST-индексы могут применяться с разными операторами, в зависимости от стратегии индексирования (*класса операторов*). Например, стандартный дистрибутив Postgres Pro включает классы операторов GiST для нескольких двумерных типов геометрических данных, что позволяет применять индексы в запросах с операторами:

```
<<   &<   &>   >>   <<|   &<|   |&>   |>>   @>   <@   ~=   &&
```

(Эти операторы описаны в Разделе 9.11.) Классы операторов GiST, включённые в стандартный дистрибутив, описаны в Таблице 62.1. В коллекции `contrib` можно найти и другие классы операторов GiST, реализованные как отдельные проекты. За дополнительными сведениями обратитесь к Разделу 62.2.

GiST-индексы также могут оптимизировать поиск «ближайшего соседа», например такой:

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

который возвращает десять расположений, ближайших к заданной точке. Возможность такого применения индекса опять же зависит от класса используемого оператора. Операторы, которые можно использовать таким образом, перечислены в Таблице 62.1, в столбце «Операторы сортировки».

11.2.4. SP-GiST

Индексы SP-GiST, как и GiST, предоставляют инфраструктуру, поддерживающую различные типы поиска. SP-GiST позволяет организовывать на диске самые разные несбалансированные структуры данных, такие как деревья квадрантов, k-мерные и префиксные деревья. Например, стандартный дистрибутив Postgres Pro включает классы операторов SP-GiST для точек в двумерном пространстве, что позволяет применять индексы в запросах с операторами:

```
<<   >>   ~=   <@   <<|   |>>
```

(Эти операторы описаны в Разделе 9.11.) Классы операторов SP-GiST, включённые в стандартный дистрибутив, описаны в Таблице 62.2. За дополнительными сведениями обратитесь к Разделу 62.3.

Индексы SP-GiST, как и GiST, поддерживают поиск ближайших соседей. Для классов операторов SP-GiST, поддерживающих упорядочивание по расстоянию, соответствующий оператор указан в столбце «Операторы упорядочивания» в Таблице 62.2.

11.2.5. GIN

GIN-индексы представляют собой «инвертированные индексы», в которых могут содержаться значения с несколькими ключами, например массивы. Инвертированный индекс содержит отдельный элемент для значения каждого компонента, и может эффективно работать в запросах, проверяющих присутствие определённых значений компонентов.

Подобно GiST и SP-GiST, индексы GIN могут поддерживать различные определённые пользователем стратегии и в зависимости от них могут применяться с разными операторами. Например, стандартный дистрибутив Postgres Pro включает класс операторов GIN для массивов, что позволяет применять индексы в запросах с операторами:

```
<@   @>   =   &&
```

(Эти операторы описаны в Разделе 9.19.) Классы операторов GIN, включённые в стандартный дистрибутив, описаны в Таблице 62.3. В коллекции `contrib` и в отдельных проектах можно найти и много других классов операторов GIN. За дополнительными сведениями обратитесь к Разделу 62.4.

11.2.6. BRIN

BRIN-индексы (сокращение от Block Range INdexes, Индексы зон блоков) хранят обобщённые сведения о значениях, находящихся в физически последовательно расположенных блоках таблицы. Поэтому такие индексы наиболее эффективны для столбцов, значения в которых хорошо коррелируют с физическим порядком столбцов таблицы. Подобно GiST, SP-GiST и GIN, индексы BRIN могут поддерживать определённые пользователем стратегии и в зависимости от них применяться с разными операторами. Для типов данных, имеющих линейный порядок сортировки, записям в индексе соответствуют минимальные и максимальные значения данных в столбце для каждой зоны блоков. Это позволяет поддерживать запросы со следующими операторами:

```
<   <=   =   >=   >
```

Классы операторов BRIN, включённые в стандартный дистрибутив, описаны в Таблице 62.4. За дополнительными сведениями обратитесь к Разделу 62.5.

11.3. Составные индексы

Индексы можно создавать и по нескольким столбцам таблицы. Например, если у вас есть таблица:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(предположим, что вы поместили в неё содержимое каталога `/dev`) и вы часто выполняете запросы вида:

```
SELECT name FROM test2 WHERE major = константа AND minor = константа;
```

тогда имеет смысл определить индекс, покрывающий оба столбца `major` и `minor`. Например:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

В настоящее время составными могут быть только индексы типов B-дерево, GiST, GIN и BRIN. Возможность построения индекса по нескольким ключевым столбцам не зависит от возможности добавления в индекс неключевых столбцов (`INCLUDE`). Число столбцов в индексе ограничивается 32, включая столбцы `INCLUDE`. (Этот предел можно изменить при компиляции Postgres Pro.)

Составной индекс-B-дерево может применяться в условиях с любым подмножеством столбцов индекса, но наиболее эффективен он при ограничениях по ведущим (левым) столбцам. Точное правило состоит в том, что сканируемая область индекса определяется условиями равенства с ведущими столбцами и условиями неравенства с первым столбцом, не участвующим в условии равенства. Ограничения столбцов правее них также проверяются по индексу, так что обращение к таблице откладывается, но на размер сканируемой области индекса это уже не влияет. Например, если есть индекс по столбцам `(a, b, c)` и условие `WHERE a = 5 AND b >= 42 AND c < 77`, индекс будет сканироваться от первой записи `a` = 5 и `b` = 42 до последней с `a` = 5. Записи индекса, в которых `c` >= 77, не будут учитываться, но тем не менее будут просканированы. Этот индекс в принципе может использоваться в запросах с ограничениями по `b` и/или `c`, без ограничений столбца `a`, но при этом будет просканирован весь индекс, так что в большинстве случаев планировщик предпочтёт использованию индекса полное сканирование таблицы.

Составной индекс GiST может применяться в условиях с любым подмножеством столбцов индекса. Условия с дополнительными столбцами ограничивают записи, возвращаемые индексом, но в первую очередь сканируемая область индекса определяется ограничением первого столбца. GiST-индекс будет относительно малоэффективен, когда первый его столбец содержит только несколько различающихся значений, даже если дополнительные столбцы дают множество различных значений.

Составной индекс GIN может применяться в условиях с любым подмножеством столбцов индекса. В отличие от индексов GiST или B-деревьев, эффективность поиска по нему не меняется в зависимости от того, какие из его столбцов используются в условиях запроса.

Составной индекс BRIN может применяться в условиях запроса с любым подмножеством столбцов индекса. Подобно индексу GIN и в отличие от B-деревьев или GiST, эффективность поиска по нему не меняется в зависимости от того, какие из его столбцов используются в условиях запроса. Единственное, зачем в одной таблице могут потребоваться несколько индексов BRIN вместо одного составного индекса — это затем, чтобы применялись разные параметры хранения `pages_per_range`.

При этом, разумеется, каждый столбец должен использоваться с операторами, соответствующими типу индекса; ограничения с другими операторами рассматриваться не будут.

Составные индексы следует использовать обдуманно. В большинстве случаев индекс по одному столбцу будет работать достаточно хорошо и сэкономит время и место. Индексы по более чем трём столбцам вряд ли будут полезными, если только таблица не используется крайне однообразно. Описание достоинств различных конфигураций индексов можно найти в Разделе 11.5 и Разделе 11.9.

11.4. Индексы и предложения `ORDER BY`

Помимо простого поиска строк для выдачи в результате запроса, индексы также могут применяться для сортировки строк в определённом порядке. Это позволяет учесть предложение `ORDER BY` в запросе, не выполняя сортировку дополнительно. Из всех типов индексов, которые поддерживает Postgres Pro, сортировать данные могут только B-деревья — индексы других типов возвращают строки в неопределённом, зависящем от реализации порядке.

Планировщик может выполнить указание `ORDER BY`, либо просканировав существующий индекс, подходящий этому указанию, либо просканировав таблицу в физическом порядке и выполнив сортировку явно. Для запроса, требующего сканирования большой части таблицы, явная сортировка скорее всего будет быстрее, чем применение индекса, так как при последовательном чтении она потребует меньше операций ввода-вывода. Важный особый случай представляет `ORDER BY` в сочетании с `LIMIT` *n*: при явной сортировке системе потребуется обработать все данные, чтобы выбрать первые *n* строк, но при наличии индекса, соответствующего столбцам в `ORDER BY`, первые *n* строк можно получить сразу, не просматривая остальные вовсе.

По умолчанию элементы B-дерева хранятся в порядке возрастания, при этом значения NULL идут в конце (для упорядочивания равных записей используется табличный столбец TID). Это означает, что при прямом сканировании индекса по столбцу `x` порядок оказывается соответствующим указанию `ORDER BY x` (или точнее, `ORDER BY x ASC NULLS LAST`). Индекс также может сканироваться в обратную сторону, и тогда порядок соответствует указанию `ORDER BY x DESC` (или точнее, `ORDER BY x DESC NULLS FIRST`, так как для `ORDER BY DESC` подразумевается `NULLS FIRST`).

Вы можете изменить порядок сортировки элементов B-дерева, добавив уточнения `ASC`, `DESC`, `NULLS FIRST` и/или `NULLS LAST` при создании индекса; например:

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

Индекс, в котором элементы хранятся в порядке возрастания и значения NULL идут первыми, может удовлетворять указаниям `ORDER BY x ASC NULLS FIRST` или `ORDER BY x DESC NULLS LAST`, в зависимости от направления просмотра.

У вас может возникнуть вопрос, зачем нужны все четыре варианта при создании индексов, когда и два варианта с учётом обратного просмотра покрывают все виды `ORDER BY`. Для индексов по одному столбцу это и в самом деле излишне, но для индексов по многим столбцам это может быть полезно. Рассмотрим индекс по двум столбцам `(x, y)`: он может удовлетворять указанию `ORDER BY x, y` при прямом сканировании или `ORDER BY x DESC, y DESC` при обратном. Но вполне возможно, что приложение будет часто выполнять `ORDER BY x ASC, y DESC`. В этом случае получить такую сортировку от простого индекса нельзя, но можно получить подходящий индекс, определив его как `(x ASC, y DESC)` или `(x DESC, y ASC)`.

Очевидно, что индексы с нестандартными правилами сортировки весьма специфичны, но иногда они могут кардинально ускорить определённые запросы. Стоит ли вводить такие индексы, зависит от того, как часто выполняются запросы с необычным порядком сортировки.

11.5. Объединение нескольких индексов

При простом сканировании индекса могут обрабатываться только те предложения в запросе, в которых применяются операторы его класса и объединяет их `AND`. Например, для индекса `(a, b)` условие запроса `WHERE a = 5 AND b = 6` сможет использовать этот индекс, а запрос `WHERE a = 5 OR b = 6` — нет.

К счастью, Postgres Pro способен соединять несколько индексов (и в том числе многократно применять один индекс) и охватывать также случаи, когда сканирования одного индекса недостаточно. Система может сформировать условия `AND` и `OR` за несколько проходов индекса. Например, запрос `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` можно разбить на четыре сканирования индекса по `x`, по сканированию для каждой части условия. Затем результаты этих сканирований будут логически сложены (OR) вместе и дадут конечный результат. Другой пример — если у нас есть отдельные индексы по `x` и `y`, запрос `WHERE x = 5 AND y = 6` можно выполнить, применив индексы для соответствующих частей запроса, а затем вычислив логическое произведение (AND) для найденных строк, которое и станет конечным результатом.

Выполняя объединение нескольких индексов, система сканирует все необходимые индексы и создаёт в памяти *битовую карту* расположения строк таблицы, которые удовлетворяют условиям каждого индекса. Затем битовые карты объединяются операциями AND и OR, как того требуют условия в запросе. Наконец система обращается к соответствующим отмеченным строкам таблицы и возвращает их данные. Строки таблицы просматриваются в физическом порядке, как они представлены в битовой карте; это означает, что порядок сортировки индексов при этом теряется и в запросах с предложением `ORDER BY` сортировка будет выполняться отдельно. По этой причине, а также потому, что каждое сканирование индекса занимает дополнительное время, планировщик иногда выбирает простое сканирование индекса, несмотря на то, что можно было бы подключить и дополнительные индексы.

В большинстве приложений (кроме самых простых) полезными могут оказаться различные комбинации индексов, поэтому разработчик баз данных, определяя набор индексов, должен искать компромиссное решение. Иногда оказываются хороши составные индексы, а иногда лучше создать отдельные индексы и положиться на возможности объединения индексов. Например, если типичную нагрузку составляют запросы иногда с условием только по столбцу `x`, иногда только по `y`, а иногда по обоим столбцам, вы можете ограничиться двумя отдельными индексами по `x` и `y`, рассчитывая на то, что при обработке условий с обоими столбцами эти индексы будут объединяться. С другой стороны, вы можете создать один составной индекс по `(x, y)`. Этот индекс скорее всего будет работать эффективнее, чем объединение индексов, в запросах с двумя столбцами, но как говорилось в Разделе 11.3, он будет практически бесполезен для запросов с ограничениями только по `y`, так что одного этого индекса будет недостаточно. Выигрышным в этом случае может быть сочетание составного индекса с отдельным индексом по `y`. В запросах, где задействуется только `x`, может применяться составной индекс, хотя он будет больше и, следовательно, медленнее индекса по одному `x`. Наконец, можно создать все три индекса, но это будет оправдано, только если данные в таблице изменяются гораздо реже, чем выполняется поиск в таблице, при этом частота запросов этих трёх типов примерно одинакова. Если запросы какого-то одного типа выполняются гораздо реже других, возможно лучше будет оставить только два индекса, соответствующих наиболее частым запросам.

11.6. Уникальные индексы

Индексы также могут обеспечивать уникальность значения в столбце или уникальность сочетания значений в нескольких столбцах.

```
CREATE UNIQUE INDEX имя ON таблица (столбец [, ...]) [ NULLS [ NOT ] DISTINCT ];
```

В настоящее время обеспечение уникальности поддерживают только индексы-B-деревья.

Если индекс создаётся как уникальный, в таблицу нельзя будет добавить несколько строк с одинаковыми значениями ключа индекса. По умолчанию значения NULL в уникальном столбце считаются не равными друг другу, так что в таком столбце может быть несколько значений NULL. Параметр `NULLS NOT DISTINCT` изменяет это правило, так что значения NULL в индексе считаются равными друг другу. Составной уникальный индекс не принимает только те строки, в которых все индексируемые столбцы содержат одинаковые значения.

Когда для таблицы определяется ограничение уникальности или первичный ключ, Postgres Pro автоматически создаёт уникальный индекс по всем столбцам, составляющим это ограничение или первичный ключ (индекс может быть составным). Такой индекс и является механизмом, который обеспечивает выполнение ограничения.

Примечание

Для уникальных столбцов не нужно вручную создавать отдельные индексы — они просто продублируют индексы, созданные автоматически.

11.7. Индексы по выражениям

Индекс можно создать не только по столбцу нижележащей таблицы, но и по функции или скалярному выражению с одним или несколькими столбцами таблицы. Это позволяет быстро находить данные в таблице по результатам вычислений.

Например, для сравнений без учёта регистра символов часто используется функция `lower`:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

Этот запрос сможет использовать индекс, определённый для результата функции `lower(col1)` так:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Если мы объявим этот индекс уникальным (`UNIQUE`), он не даст добавить строки, в которых значения `col1` различаются только регистром, как и те, в которых значения `col1` действительно одинаковые. Таким образом, индексы по выражениям можно использовать ещё и для обеспечения ограничений, которые нельзя записать как простые ограничения уникальности.

Если же часто выполняются запросы вида:

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

тогда, возможно, стоит создать такой индекс:

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

Синтаксис команды `CREATE INDEX` обычно требует заключать индексные выражения в скобки, как показано во втором примере. Если же выражение представляет собой просто вызов функции, как в первом примере, дополнительные скобки можно опустить.

Поддержка индексируемых выражений обходится довольно дорого, так как эти выражения должны вычисляться при каждом добавлении строки и при каждом изменении без оптимизации HOT. Однако при поиске по индексу индексируемое выражение *не* вычисляется повторно, так как его результат уже сохранён в индексе. В рассмотренных выше случаях система видит запрос как `WHERE столбец_индекса = 'константа'`, и поэтому поиск выполняется так же быстро, как и с простым индексом. Таким образом, индексы по выражениям могут быть полезны, когда скорость извлечения данных гораздо важнее скорости добавления и изменения.

11.8. Частичные индексы

Частичный индекс — это индекс, который строится по подмножеству строк таблицы, определяемому условным выражением (оно называется *предикатом* частичного индекса). Такой индекс содержит записи только для строк, удовлетворяющих предикату. Частичные индексы довольно специфичны, но в ряде ситуаций они могут быть очень полезны.

Частичные индексы могут быть полезны, во-первых, тем, что позволяют избежать индексирования распространённых значений. Так как при поиске распространённого значения (такого, которое содержится в значительном проценте всех строк) индекс всё равно не будет использоваться, хранить эти строки в индексе нет смысла. Исключив их из индекса, можно уменьшить его размер, а значит и ускорить запросы, использующие этот индекс. Это также может ускорить операции изменения данных в таблице, так как индекс будет обновляться не всегда. Возможное применение этой идеи проиллюстрировано в Примере 11.1.

Пример 11.1. Настройка частичного индекса, исключающего распространённые значения

Предположим, что вы храните в базе данных журнал обращений к корпоративному сайту. Большая часть обращений будет происходить из диапазона IP-адресов вашей компании, а остальные могут быть откуда угодно (например, к нему могут подключаться внешние сотрудники с динамическими IP). Если при поиске по IP вас обычно интересуют внешние подключения, IP-диапазон внутренней сети компании можно не включать в индекс.

Пусть у вас есть такая таблица:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

Создать частичный индекс для нашего примера можно так:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

Так будет выглядеть типичный запрос, использующий этот индекс:

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

В нём фигурирует IP-адрес, попадающий в частичный индекс. Следующий запрос не может использовать частичный индекс, так как в нём IP-адрес не попадает в диапазон индекса:

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '192.168.100.23';
```

Заметьте, что при таком определении частичного индекса необходимо, чтобы распространённые значения были известны заранее, так что такие индексы лучше использовать, когда распределение данных не меняется. Хотя такие индексы можно пересоздавать время от времени, подстраиваясь под новое распределение, это значительно усложняет поддержку.

Во-вторых, частичные индексы могут быть полезны тем, что позволяют исключить из индекса значения, которые обычно не представляют интереса; это проиллюстрировано в Примере 11.2. При этом вы получаете те же преимущества, что и в предыдущем случае, но система не сможет извлечь «неинтересные» значения по этому индексу, даже если сканирование индекса может быть эффективным. Очевидно, настройка частичных индексов в таких случаях требует тщательного анализа и тестирования.

Пример 11.2. Настройка частичного индекса, исключающего неинтересные значения

Если у вас есть таблица, в которой хранятся и оплаченные, и неоплаченные счета, и при этом неоплаченные счета составляют только небольшую часть всей таблицы, но представляют наибольший интерес, производительность запросов можно увеличить, создав индекс только по неоплаченным счетам. Сделать это можно следующей командой:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

Этот индекс будет применяться, например в таком запросе:

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

Однако он также может применяться в запросах, где `order_nr` вообще не используется, например:

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

Конечно, он будет не так эффективен, как мог бы быть частичный индекс по столбцу `amount`, так как системе придётся сканировать его целиком. Тем не менее, если неоплаченных счетов сравнительно мало, выиграть при поиске неоплаченного счёта можно и с таким частичным индексом.

Заметьте, что в таком запросе этот индекс не будет использоваться:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

Счёт с номером 3501 может оказаться, как в числе неоплаченных, так и оплаченных.

Пример 11.2 также показывает, что индексируемый столбец не обязательно должен совпадать со столбцом, используемым в предикате. Postgres Pro поддерживает частичные индексы с произвольными предикатами — главное, чтобы в них фигурировали только столбцы индексируемой таблицы. Однако не забывайте, что предикат должен соответствовать условиям запросов, для оптимизации которых предназначается данный индекс. Точнее, частичный индекс будет применяться в запросе, только если система сможет понять, что условие `WHERE` данного запроса математически сводится к предикату индекса. Но учтите, что Postgres Pro не умеет доказывать математические утверждения об эквивалентности выражений, записанных в разных формах. (Составить программу для таких доказательств крайне сложно, и если даже это удастся, скорость её будет неприемлема для применения на практике.) Система может выявить только самые простые следствия с неравенствами; например, понять, что из «x < 1» следует «x < 2»; во всех остальных случаях условие предиката должно точно совпадать с условием в предложении `WHERE`, иначе индекс будет считаться неподходящим. Сопоставление условий происходит во время планирования запросов, а не во время выполнения. Как следствие, запросы с параметрами не будут работать с частичными индексами. Например, условие с параметром «x < ?» в подготовленном запросе никогда не будет сведено к «x < 2» при всех возможных значениях параметра.

Третье возможное применение частичных индексов вообще не связано с использованием индекса в запросах. Идея заключается в том, чтобы создать уникальный индекс по подмножеству строк таблицы, как в Примере 11.3. Это обеспечит уникальность среди строк, удовлетворяющих условию предиката, но никак не будет ограничивать остальные.

Пример 11.3. Настройка частичного уникального индекса

Предположим, что у нас есть таблица с результатами теста. Мы хотим, чтобы для каждого сочетания предмета и целевой темы была только одна запись об успешном результате, а неудачных попыток могло быть много. Вот как можно этого добиться:

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
```

```
    WHERE success;
```

Это подход будет особенно эффективным, когда неудачных попыток будет намного больше, чем удачных. Также можно потребовать, чтобы в столбце допускался только один NULL, создав уникальный частичный индекс с ограничением `IS NULL`.

Наконец, с помощью частичных индексов можно также переопределять выбираемый системой план запроса. Возможно, что для данных с неудачным распределением система решит использовать индекс, тогда как на самом деле это неэффективно. В этом случае индекс можно настроить так, чтобы в подобных запросах он не работал. Обычно Postgres Pro принимает разумные решения относительно применения индексов (т. е. старается не использовать их для получения распространённых значений, так что частичный индекс в вышеприведённом примере помог только уменьшить размер индекса, для отказа от использования индекса он не требовался), поэтому крайне неэффективный план может быть поводом для сообщения об ошибке.

Помните, что настраивая частичный индекс, вы тем самым заявляете, что знаете о данных гораздо больше, чем планировщик запросов. В частности, вы знаете, когда такой индекс может быть полезен. Это знание обязательно должно подкрепляться опытом и пониманием того, как работают индексы в Postgres Pro. В большинстве случаев преимущества частичных индексов по сравнению с обычными будут минимальными. Однако в ряде случаев эти индексы могут быть даже вредны, о чём говорится в Примере 11.4.

Пример 11.4. Не применяйте частичные индексы в качестве замены секционированию

У вас может возникнуть желание создать множество неперекрывающихся частичных индексов, например:

```
CREATE INDEX mytable_cat_1 ON mytable (data) WHERE category = 1;
CREATE INDEX mytable_cat_2 ON mytable (data) WHERE category = 2;
CREATE INDEX mytable_cat_3 ON mytable (data) WHERE category = 3;
...
CREATE INDEX mytable_cat_N ON mytable (data) WHERE category = N;
```

Но так делать не следует! Почти наверняка вам лучше использовать один не частичный индекс, объявленный так:

```
CREATE INDEX mytable_cat_data ON mytable (category, data);
```

(Поставьте первым столбец категорий, по причинам описанным в Разделе 11.3.) При поиске в большем индексе может потребоваться опуститься на несколько уровней ниже, чем при поиске в меньшем частичном, но это почти гарантированно будет дешевле, чем выбрать при планировании из всех частичных индексов подходящий. Сложность с выбором индекса объясняется тем, что система не знает, как взаимосвязаны частичные индексы, и ей придётся проверять каждый из них, чтобы понять, соответствует ли он текущему запросу.

Если ваша таблица настолько велика, что создавать один индекс кажется действительно плохой идей, рассмотрите возможность использования секционирования (см. Раздел 5.12). Когда применяется этот механизм, система понимает, что таблицы и индексы не перекрываются, и может выполнять запросы гораздо эффективнее.

Узнать о частичных индексах больше можно в следующих источниках: ston89b, olson93 и seshadri95.

11.9. Сканирование только индекса и покрывающие индексы

Все индексы в Postgres Pro являются *вторичными*, что значит, что каждый индекс хранится вне области основных данных таблицы (которая в терминологии Postgres Pro называется *кучей* таблицы). Это значит, что при обычном сканировании индекса для извлечения каждой строки необходимо прочитать данные и из индекса, и из кучи. Более того, тогда как элементы индекса, соответствующие заданному условию `WHERE`, обычно находятся в индексе рядом, строки таблицы могут располагаться в куче произвольным образом. Таким образом, обращение к куче при поиске по индексу влечёт множество операций произвольного чтения кучи, которые могут обойтись недёшево, особенно на традиционных вращающихся носителях. (Как описано в Разделе 11.5, сканирование по битовой карте пытается снизить стоимость этих операций, упорядочивая доступ к куче, но не более того.)

Чтобы решить эту проблему с производительностью, Postgres Pro поддерживает *сканирование только индекса*, при котором результат запроса может быть получен из самого индекса, без обращения к куче. Основная идея такого сканирования в том, чтобы выдавать значения непосредственно из элемента индекса, и не обращаться к соответствующей записи в куче. Для применения этого метода есть два фундаментальных ограничения:

1. Тип индекса должен поддерживать сканирование только индекса. Индексы-B-деревья поддерживают его всегда. Индексы GiST и SP-GiST могут поддерживать его с одними классами операторов и не поддерживать с другими. Другие индексы такое сканирование не поддерживают. Суть нижележащего требования в том, что индекс должен физически хранить или каким-то образом восстанавливать исходное значение данных для каждого элемента индекса. В качестве контрпримера, индексы GIN неспособны поддерживать сканирование только индекса, так как в элементах индекса обычно хранится только часть исходного значения данных.
2. Запрос должен обращаться только к столбцам, сохранённым в индексе. Например, если в таблице построен индекс по столбцам `x` и `y`, и в ней есть также столбец `z`, такие запросы будут использовать сканирование только индекса:

   ```
   SELECT x, y FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND y < 42;
   ```

   А эти запросы не будут:

   ```
   SELECT x, z FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND z < 42;
   ```

   (Индексы по выражениям и частичные индексы усложняют это правило, как описано ниже.)

Если два этих фундаментальных ограничения выполняются, то все данные, требуемые для выполнения запроса, содержатся в индексе, так что сканирование только по индексу физически возможно. Но в Postgres Pro существует и ещё одно требование для сканирования таблицы: необходимо убедиться, что все возвращаемые строки «видны» в снимке MVCC запроса, как описано в Главе 13. Информация о видимости хранится не в элементах индекса, а только в куче; поэтому на первый взгляд может показаться, что для получения данных каждой строки всё равно необходимо обращаться к куче. И это в самом деле так, если в таблице недавно произошли изменения. Однако для редко меняющихся данных есть возможность обойти эту проблему. Postgres Pro отслеживает для каждой страницы в куче таблицы, являются ли все строки в этой странице достаточно старыми, чтобы их видели все текущие и будущие транзакции. Это отражается в битах в *карте видимости* таблицы. Процедура сканирования только индекса, найдя потенциально подходящую запись в индексе, проверяет бит в карте видимости для соответствующей страницы в куче. Если он установлен, значит эта строка видна, и данные могут быть возвращены сразу. В противном случае придётся посетить запись строки в куче и проверить, видима ли она, так что никакого выигрыша по сравнению с обычным сканированием индекса не будет. Однако в благоприятном случае обращение к куче заменяется обращением к карте видимости; но так как карта видимости на четыре порядка меньше соответствующей ей области кучи, для работы с ней требуется много меньше операций физического ввода-вывода. В большинстве ситуаций карта видимости просто всё время находится в памяти.

Таким образом, тогда как сканирование только по индексу возможно лишь при выполнении двух фундаментальных требований, оно даст выигрыш, только если для значительной части страниц в куче таблицы установлены биты полной видимости. Но таблицы, в которых меняется лишь небольшая часть строк, встречаются достаточно часто, чтобы этот тип сканирования был весьма полезен на практике.

Чтобы эффективно использовать возможность сканирования только индекса, вы можете создавать *покрывающие индексы*. Такие индексы специально предназначены для включения столбцов, которые требуются в определённых часто выполняемых запросах. Так как в запросах обычно нужно получить не только столбцы, по которым выполняется поиск, Postgres Pro позволяет создать индекс, в котором некоторые столбцы будут просто «дополнительной нагрузкой», но не войдут в поисковый ключ. Это реализуется предложением `INCLUDE`, в котором перечисляются дополнительные столбцы. Например, если часто выполняется запрос вида

```
SELECT y FROM tab WHERE x = 'key';
```

при традиционном подходе его можно ускорить, создав индекс только по `x`. Однако такой индекс:

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

может удовлетворить такие запросы при сканировании только индекса, так как значение `y` можно получить из индекса, не обращаясь к данным в куче.

Так как столбец `y` не является частью поискового ключа, он не обязательно должен иметь тип данных, воспринимаемый данным индексом; он просто сохраняется внутри индекса и никак не обрабатывается механизмом индекса. Кроме того, в случае с уникальным индексом, например:

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

условие уникальности распространяется только на столбец `x`, а не на `x` и `y` в совокупности. (Предложение `INCLUDE` можно также добавить в ограничения `UNIQUE` и `PRIMARY KEY`, что позволяет определить такой индекс альтернативным образом.)

Добавлять в индекс неключевые дополнительные столбцы следует обдуманно, особенно когда это большие столбцы. Если размер кортежа в индексе превысит максимально допустимый размер для типа индексов, при добавлении данных возникнет ошибка. В любом случае в неключевых столбцах дублируются данные из самой таблицы, что приводит к раздуванию индекса, а следствием этого может быть замедление запросов. И помните, что практический смысл включать дополнительные столбцы в индекс есть только тогда, когда таблица меняется достаточно медленно, и при сканировании только индекса не приходится обращаться к куче. Если кортеж в любом случае придётся прочитывать из кучи, получить значение столбца из него ничего не стоит. Покрывающие индексы имеют и другие ограничения: в настоящее время в качестве неключевых столбцов нельзя задать выражения, и поддерживаются такие индексы только трёх типов: B-деревья, GiST и SP-GiST.

До появления в Postgres Pro покрывающих индексов (`INCLUDE`) пользователям иногда приходилось задействовать дополнительные столбцы как обычные столбцы индекса, то есть писать

```
CREATE INDEX tab_x_y ON tab(x, y);
```

даже не намереваясь когда-либо использовать `y` в предложении `WHERE`. Это работает, когда дополнительные столбцы добавляются в конец; делать их начальными неразумно по причинам, описанным в Разделе 11.3. Однако этот подход не годится для случая, когда вам нужно обеспечить уникальность ключевого столбца (столбцов).

В процессе *усечения суффикса* с верхних уровней B-дерева всегда удаляются неключевые столбцы. Так как они содержат дополнительную нагрузку, они никогда не управляют поиском по индексу. В этом процессе также удаляется один или несколько замыкающих столбцов, если остающегося префикса ключевых столбцов оказывается достаточно, чтобы описать кортежи на нижележащем уровне B-дерева. На практике и в покрывающих индексах без предложения `INCLUDE` часто удаётся избежать хранения столбцов, которые вверху по сути являются допнагрузкой. Однако явное обозначение дополнительных столбцов неключевыми *гарантирует* минимальный размер кортежей на верхних уровнях.

В принципе сканирование только индекса может применяться и с индексами по выражениям. Например, при наличии индекса по `f(x)`, где `x` — столбец таблицы, должно быть возможно выполнить

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

как сканирование только индекса; и это очень заманчиво, если `f()` — сложная для вычисления функция. Однако планировщик Postgres Pro в настоящее время может вести себя не очень разумно. Он считает, что запрос может выполняться со сканированием только индекса, лишь когда из индекса могут быть получены все *столбцы*, требующиеся для запроса. В этом примере `x` фигурирует только в контексте `f(x)`, но планировщик не замечает этого и решает, что сканирование только по индексу невозможно. Если сканирование только индекса заслуживает того, эту проблему можно обойти, добавив `x` как неключевой столбец, например:

```
CREATE INDEX tab_f_x ON tab (f(x)) INCLUDE (x);
```

Если это делается ради предотвращения многократных вычислений `f(x)`, следует также учесть, что планировщик не обязательно свяжет упоминания `f(x)`, фигурирующие вне индексируемых предложений `WHERE`, со столбцом индекса. Обычно он делает это правильно в простых запросах, вроде показанного выше, но не в запросах с соединениями. Эти недостатки могут быть устранены в будущих версиях Postgres Pro.

С использованием частичных индексов при сканировании только по индексу тоже связаны интересные особенности. Предположим, что у нас есть частичный индекс, показанный в Примере 11.3:

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

В принципе с ним мы можем произвести сканирование только по индексу при выполнении запроса

```
SELECT target FROM tests WHERE subject = 'some-subject' AND success;
```

Но есть одна проблема: предложение `WHERE` обращается к столбцу `success`, который отсутствует в результирующих столбцах индекса. Тем не менее сканирование только индекса возможно, так как плану не нужно перепроверять эту часть предложения `WHERE` во время выполнения: у всех записей, найденных в индексе, значение `success = true`, так что в плане его не нужно проверять явно. Postgres Pro версий 9.6 и новее распознает такую ситуацию и сможет произвести сканирование только по индексу, но старые версии неспособны на это.

11.10. Семейства и классы операторов

В определении индекса можно указать *класс операторов* для каждого столбца индекса.

```
CREATE INDEX имя ON таблица (столбец класс_оп [ ( параметры_класса_оп ) ] [параметры
 сортировки] [, ...]);
```

Класс операторов определяет, какие операторы будет использовать индекс для этого столбца. Например, индекс-B-дерево по столбцу `int4` будет использовать класс `int4_ops`; этот класс операторов включает операции со значениями типа `int4`. На практике часто достаточно принять класс операторов, назначенный для типа столбца классом по умолчанию. Однако для некоторых типов данных могут иметь смысл несколько разных вариантов индексирования и реализовать их как раз позволяют разные классы операторов. Например, комплексные числа можно сортировать как по вещественной части, так и по модулю. Получить два варианта индексов для них можно, определив два класса операторов для данного типа и выбрав соответствующий класс при создании индекса. Выбранный класс операторов задаст основной порядок сортировки данных (его можно уточнить, добавив параметры `COLLATE`, `ASC`/`DESC` и/или `NULLS FIRST`/`NULLS LAST`).

Помимо классов операторов по умолчанию есть ещё несколько встроенных:

- Классы операторов `text_pattern_ops`, `varchar_pattern_ops` и `bpchar_pattern_ops` поддерживают индексы-B-деревья для типов `text`, `varchar` и `char`, соответственно. От стандартных классов операторов они отличаются тем, что сравнивают значения по символам, не применяя правила сортировки, определённые локалью. Благодаря этому они подходят для запросов с поиском по шаблону (с `LIKE` и регулярными выражениями POSIX), когда локаль базы данных не стандартная «C». Например, вы можете проиндексировать столбец `varchar` так:

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Заметьте, что при этом также следует создать индекс с классом операторов по умолчанию, если вы хотите ускорить запросы с обычными сравнениями <, <=, > и >= за счёт применения индексов. Классы операторов *xxx*`_pattern_ops` не подходят для таких сравнений. (Однако для проверки равенств эти классы операторов вполне пригодны.) В подобных случаях для одного столбца можно создать несколько индексов с разными классами операторов. Если же вы используете локаль C, классы операторов *xxx*`_pattern_ops` вам не нужны, так как для поиска по шаблону в локали C будет достаточно индексов с классом операторов по умолчанию.

Следующий запрос выводит список всех существующих классов операторов:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

Класс операторов на самом деле является всего лишь подмножеством большой структуры, называемой *семейством операторов*. В случаях, когда несколько типов данных ведут себя одинаково, часто имеет смысл определить операторы так, чтобы они могли использоваться с индексами сразу нескольких типов. Сделать это можно, сгруппировав классы операторов для этих типов в одном семействе операторов. Такие многоцелевые операторы, являясь членами семейства, не будут связаны с каким-либо одним его классом.

Расширенная версия предыдущего запроса показывает семейство операторов, к которому принадлежит каждый класс операторов:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc, pg_opfamily opf
    WHERE opc.opcmethod = am.oid AND
          opc.opcfamily = opf.oid
    ORDER BY index_method, opclass_name;
```

Этот запрос выводит все существующие семейства операторов и все операторы, включённые в эти семейства:

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

Подсказка

В psql есть команды `\dAc`, `\dAf` и `\dAo`, которые выводят более усовершенствованные версии этих запросов.

11.11. Индексы и правила сортировки

Один индекс может поддерживать только одно правило сортировки для индексируемого столбца. Поэтому при необходимости применять разные правила сортировки могут потребоваться несколько индексов.

Рассмотрим следующие операторы:

```
CREATE TABLE test1c (
    id integer,
```

```
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

Этот индекс автоматически использует правило сортировки нижележащего столбца. И запрос вида

```
SELECT * FROM test1c WHERE content > константа;
```

сможет использовать этот индекс, так как при сравнении по умолчанию будет действовать правило сортировки столбца. Однако этот индекс не поможет ускорить запросы с каким-либо другим правилом сортировки. Поэтому, если интерес представляют также и запросы вроде

```
SELECT * FROM test1c WHERE content > константа COLLATE "y";
```

для них можно создать дополнительный индекс, поддерживающий правило сортировки `"y"`, примерно так:

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```

11.12. Контроль использования индексов

Хотя индексы в Postgres Pro не требуют какого-либо обслуживания или настройки, это не избавляет от необходимости проверять, как и какие индексы используются на самом деле в реальных условиях. Узнать, как отдельный запрос использует индексы, можно с помощью команды EXPLAIN; её применение для этих целей описывается в Разделе 14.1. Также возможно собрать общую статистику об использовании индексов на работающем сервере, как описано в Разделе 26.2.

Вывести универсальную формулу, определяющую, какие индексы нужно создавать, довольно сложно, если вообще возможно. В предыдущих разделах рассматривались некоторые типовые ситуации, иллюстрирующие подходы к этому вопросу. Часто найти ответ на него помогают эксперименты. Ниже приведены ещё несколько советов:

- Всегда начинайте исследование с ANALYZE. Эта команда собирает статистические данные о распределении значений в таблице, которые необходимы для оценивания числа строк, возвращаемых запросов. А это число, в свою очередь, нужно планировщику, чтобы оценить реальные затраты для всевозможных планов выполнения запроса. Не имея реальной статистики, планировщик будет вынужден принять некоторые значения по умолчанию, которые почти наверняка не будут соответствовать действительности. Поэтому понять, как индекс используется приложением без предварительного запуска `ANALYZE`, практически невозможно. Подробнее это рассматривается в Подразделе 23.1.3 и Подразделе 23.1.6.
- Используйте в экспериментах реальные данные. Анализируя работу системы с тестовыми данными, вы поймёте, какие индексы нужны для тестовых данных, но не более того.

  Особенно сильно искажают картину очень маленькие наборы тестовых данных. Тогда как для извлечения 1000 строк из 100000 может быть применён индекс, для выбора 1 из 100 он вряд ли потребуется, так как 100 строк скорее всего уместятся в одну страницу данных на диске и никакой другой план не будет лучше обычного сканирования 1 страницы.

  Тем не менее, пока приложение не эксплуатируется, создавать какие-то тестовые данные всё равно нужно, и это нужно делать обдуманно. Если вы наполняете базу данных очень близкими, или наоборот, случайными значениями, либо добавляете строки в отсортированном порядке, вы получите совсем не ту статистику распределения, что дадут реальные данные.
- Когда индексы не используются, ради тестирования может быть полезно подключить их принудительно. Для этого можно воспользоваться параметрами выполнения, позволяющими выключать различные типы планов (см. Подраздел 18.7.1). Например, выключив наиболее простые планы: последовательное сканирование (`enable_seqscan`) и соединения с вложенными циклами (`enable_nestloop`), вы сможете заставить систему выбрать другой план. Если же система продолжает выполнять сканирование или соединение с вложенными циклами, вероятно, у неё есть более серьёзная причина не использовать индекс; например, индекс может не

соответствовать условию запроса. (Какие индексы работают в запросах разных типов, обсуждалось в предыдущих разделах.)

- Если система начинает использовать индекс только под принуждением, тому может быть две причины: либо система права и применять индекс в самом деле неэффективно, либо оценка стоимости применения индекса не соответствует действительности. В этом случае вам следует замерить время выполнения запроса с индексами и без них. В анализе этой ситуации может быть полезна команда `EXPLAIN ANALYZE`.
- Если выясняется, что оценка стоимости неверна, это может иметь тоже два объяснения. Общая стоимость вычисляется как произведение цены каждого узла плана для одной строки и оценки избирательности узла плана. Цены узлов при необходимости можно изменить параметрами выполнения (описанными в Подразделе 18.7.2). С другой стороны, оценка избирательности может быть неточной из-за некачественной статистики. Улучшить её можно, настроив параметры сбора статистики (см. ALTER TABLE).

Если ваши попытки скорректировать стоимость планов не увенчаются успехом, возможно вам останется только явно заставить систему использовать нужный индекс. Вероятно, имеет смысл также связаться с разработчиками Postgres Pro, чтобы прояснить ситуацию.

Глава 12. Полнотекстовый поиск

12.1. Введение

Полнотекстовый поиск (или просто *поиск текста*) — это возможность находить *документы* на естественном языке, соответствующие *запросу*, и, возможно, дополнительно сортировать их по релевантности для этого запроса. Наиболее распространённая задача — найти все документы, содержащие *слова запроса*, и выдать их отсортированными по степени *соответствия* запросу. Понятия `запроса` и `соответствия` довольно расплывчаты и зависят от конкретного приложения. В самом простом случае `запросом` считается набор слов, а `соответствие` определяется частотой слов в документе.

Операторы текстового поиска существуют в СУБД уже многие годы. В Postgres Pro для текстовых типов данных есть операторы `~`, `~*`, `LIKE` и `ILIKE`, но им не хватает очень важных вещей, которые требуются сегодня от информационных систем:

- Нет поддержки лингвистической функциональности, даже для английского языка. Возможности регулярных выражений ограничены — они не рассчитаны на работу со словоформами, например, `подходят` и `подходить`. С ними вы можете пропустить документы, которые содержат `подходят`, но, вероятно, и они представляют интерес при поиске по ключевому слову `подходить`. Конечно, можно попытаться перечислить в регулярном выражении все варианты слова, но это будет очень трудоёмко и чревато ошибками (некоторые слова могут иметь десятки словоформ).
- Они не позволяют упорядочивать результаты поиска (по релевантности), а без этого поиск неэффективен, когда находятся сотни подходящих документов.
- Они обычно выполняются медленно из-за отсутствия индексов, так как при каждом поиске приходится просматривать все документы.

Полнотекстовая индексация заключается в *предварительной обработке* документов и сохранении индекса для последующего быстрого поиска. Предварительная обработка включает следующие операции:

Разбор документов на фрагменты. При этом полезно выделить различные классы фрагментов, например, числа, слова, словосочетания, почтовые адреса и т. д., которые будут обрабатываться по-разному. В принципе классы фрагментов могут зависеть от приложения, но для большинства применений вполне подойдёт предопределённый набор классов. Эту операцию в Postgres Pro выполняет *анализатор* (parser). Вы можете использовать как стандартный анализатор, так и создавать свои, узкоспециализированные.

Преобразование фрагментов в лексемы. Лексема — это *нормализованный* фрагмент, в котором разные словоформы приведены к одной. Например, при нормализации буквы верхнего регистра приводятся к нижнему, а из слов обычно убираются окончания (в частности, `s` или `es` в английском). Благодаря этому можно находить разные формы одного слова, не вводя вручную все возможные варианты. Кроме того, на данном шаге обычно исключаются *стоп-слова*, то есть слова, настолько распространённые, что искать их нет смысла. (Другими словами, фрагменты представляют собой просто подстроки текста документа, а лексемы — это слова, имеющие ценность для индексации и поиска.) Для выполнения этого шага в Postgres Pro используются *словари*. Набор существующих стандартных словарей при необходимости можно расширять, создавая свои собственные.

Хранение документов в форме, подготовленной для поиска. Например, каждый документ может быть представлен в виде сортированного массива нормализованных лексем. Помимо лексем часто желательно хранить информацию об их положении для *ранжирования по близости*, чтобы документ, в котором слова запроса расположены «плотнее», получал более высокий ранг, чем документ с разбросанными словами.

Словари позволяют управлять нормализацией фрагментов с большой гибкостью. Создавая словари, можно:

- Определять стоп-слова, которые не будут индексироваться.

- Сопоставлять синонимы с одним словом, используя Ispell.
- Сопоставлять словосочетания с одним словом, используя тезаурус.
- Сопоставлять различные склонения слова с канонической формой, используя словарь Ispell.
- Сопоставлять различные склонения слова с канонической формой, используя стеммер Snowball.

Для хранения подготовленных документов в PostgreSQL предназначен тип данных `tsvector`, а для представления обработанных запросов — тип `tsquery` (Раздел 8.11). С этими типами данных работают целый ряд функций и операторов (Раздел 9.13), и наиболее важный из них — оператор соответствия `@@`, с которым мы познакомимся в Подразделе 12.1.2. Для ускорения полнотекстового поиска могут применяться индексы (Раздел 12.9).

12.1.1. Что такое документ?

Документ — это единица обработки в системе полнотекстового поиска; например, журнальная статья или почтовое сообщение. Система поиска текста должна уметь разбирать документы и сохранять связи лексем (ключевых слов) с содержащим их документом. Впоследствии эти связи могут использоваться для поиска документов с заданными ключевыми словами.

В контексте поиска в Postgres Pro документ — это обычно содержимое текстового поля в строке таблицы или, возможно, сочетание (объединение) таких полей, которые могут храниться в разных таблицах или формироваться динамически. Другими словами, документ для индексации может создаваться из нескольких частей и не храниться где-либо как единое целое. Например:

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE m.mid = d.did AND m.mid = 12;
```

Примечание

На самом деле в этих примерах запросов следует использовать функцию `coalesce`, чтобы значение `NULL` в каком-либо одном атрибуте не привело к тому, что результирующим документом окажется `NULL`.

Документы также можно хранить в обычных текстовых файлах в файловой системе. В этом случае база данных может быть просто хранилищем полнотекстового индекса и исполнителем запросов, а найденные документы будут загружаться из файловой системы по некоторым уникальным идентификаторам. Однако для загрузки внешних файлов требуются права суперпользователя или поддержка специальных функций, так что это обычно менее удобно, чем хранить все данные внутри БД Postgres Pro. Кроме того, когда всё хранится в базе данных, это упрощает доступ к метаданным документов при индексации и выводе результатов.

Для нужд текстового поиска каждый документ должен быть сведён к специальному формату `tsvector`. Поиск и ранжирование выполняется исключительно с этим представлением документа — исходный текст потребуется извлечь, только когда документ будет отобран для вывода пользователю. Поэтому мы часто подразумеваем под `tsvector` документ, тогда как этот тип, конечно, содержит только компактное представление всего документа.

12.1.2. Простое соответствие текста

Полнотекстовый поиск в Postgres Pro реализован на базе оператора соответствия `@@`, который возвращает `true`, если `tsvector` (документ) соответствует `tsquery` (запросу). Для этого оператора не важно, какой тип записан первым:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
```

```
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
----------
 f
```

Как можно догадаться из этого примера, `tsquery` — это не просто текст, как и `tsvector`. Значение типа `tsquery` содержит искомые слова, это должны быть уже нормализованные лексемы, возможно объединённые в выражение операторами И, ИЛИ, НЕ и ПРЕДШЕСТВУЕТ. (Подробнее синтаксис описан в Подразделе 8.11.2.) Вы можете воспользоваться функциями `to_tsquery`, `plainto_tsquery` и `phraseto_tsquery`, которые могут преобразовать заданный пользователем текст в значение `tsquery`, прежде всего нормализуя слова в этом тексте. Функция `to_tsvector` подобным образом может разобрать и нормализовать текстовое содержимое документа. Так что запрос с поиском соответствия на практике выглядит скорее так:

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

Заметьте, что соответствие не будет обнаружено, если запрос записан как

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

так как слово `rats` не будет нормализовано. Элементами `tsvector` являются лексемы, предположительно уже нормализованные, так что `rats` считается не соответствующим `rat`.

Оператор `@@` также может принимать типы `text`, позволяя опустить явные преобразования текстовых строк в типы `tsvector` и `tsquery` в простых случаях. Всего есть четыре варианта этого оператора:

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

Первые два мы уже видели раньше. Форма `text@@tsquery` равнозначна выражению `to_tsvector(x) @@ y`, а форма `text@@text` — выражению `to_tsvector(x) @@ plainto_tsquery(y)`.

В значении `tsquery` оператор `&` (И) указывает, что оба его операнда должны присутствовать в документе, чтобы он удовлетворял запросу. Подобным образом, оператор `|` (ИЛИ) указывает, что в документе должен присутствовать минимум один из его операндов, тогда как оператор `!` (НЕ) указывает, что его операнд *не* должен присутствовать, чтобы условие удовлетворялось. Например, запросу `fat & ! rat` соответствуют документы, содержащие `fat` и не содержащие `rat`.

Фразовый поиск возможен с использованием оператора `<->` (ПРЕДШЕСТВУЕТ) типа `tsquery`, который находит соответствие, только если его операнды расположены рядом и в заданном порядке. Например:

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <-> error');
```

```
 ?column?
----------
 f
```

Более общая версия оператора ПРЕДШЕСТВУЕТ имеет вид `<N>`, где *N* — целое число, выражающее разность между позициями найденных лексем. Запись `<1>` равнозначна `<->`, тогда как `<2>` допускает существование ровно одной лексемы между этими лексемами и т. д. Функция `phraseto_tsquery` задействует этот оператор для конструирования `tsquery`, который может содержать многословную фразу, включающую в себя стоп-слова. Например:

```
SELECT phraseto_tsquery('cats ate rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <-> 'rat'

SELECT phraseto_tsquery('the cats ate the rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <2> 'rat'
```

Особый случай, который иногда бывает полезен, представляет собой запись `<0>`, требующая, чтобы обоим лексемам соответствовало одно слово.

Сочетанием операторов `tsquery` можно управлять, применяя скобки. Без скобок операторы имеют следующие приоритеты, в порядке возрастания: `|`, `&`, `<->` и самый приоритетный — `!`.

Стоит отметить, что операторы И/ИЛИ/НЕ имеют несколько другое значение, когда они применяются в аргументах оператора ПРЕДШЕСТВУЕТ, так как в этом случае имеет значение точная позиция совпадения. Например, обычному `!x` соответствуют только документы, не содержащие `x` нигде. Но условию `!x <-> y` соответствует `y`, если оно не следует непосредственно за `x`; при вхождении `x` в любом другом месте документа он не будет исключаться из рассмотрения. Другой пример: для условия `x & y` обычно требуется, чтобы и `x`, и `y` встречались в каком-то месте документа, но для выполнения условия `(x & y) <-> z` требуется, чтобы `x` и `y` располагались в одном месте, непосредственно перед `z`. Таким образом, этот запрос отличается от `x <-> z & y <-> z`, которому удовлетворяют документы, содержащие две отдельные последовательности `x z` и `y z`. (Этот конкретный запрос в таком виде, как он записан, не имеет смысла, так как `x` и `y` не могут находиться в одном месте; но в более сложных ситуациях, например, с шаблонами поиска по маске, запросы этого вида могут быть полезны.)

12.1.3. Конфигурации

До этого мы рассматривали очень простые примеры поиска текста. Как было упомянуто выше, весь функционал текстового поиска позволяет делать гораздо больше: пропускать определённые слова (стоп-слова), обрабатывать синонимы и выполнять сложный анализ слов, например, выделять фрагменты не только по пробелам. Все эти функции управляются *конфигурациями текстового поиска*. В Postgres Pro есть набор предопределённых конфигураций для многих языков, но вы также можете создавать собственные конфигурации. (Все доступные конфигурации можно просмотреть с помощью команды `\dF` в psql.)

Подходящая конфигурация для данной среды выбирается во время установки и записывается в параметре default_text_search_config в `postgresql.conf`. Если вы используете для всего кластера одну конфигурацию текстового поиска, вам будет достаточно этого параметра в `postgresql.conf`. Если же требуется использовать в кластере разные конфигурации, но для каждой базы данных одну определённую, её можно задать командой `ALTER DATABASE ... SET`. В противном случае конфигурацию можно выбрать в рамках сеанса, определив параметр `default_text_search_config`.

У каждой функции текстового поиска, зависящей от конфигурации, есть необязательный аргумент `regconfig`, в котором можно явно указать конфигурацию для данной функции. Значение `default_text_search_config` используется, только когда этот аргумент опущен.

Для упрощения создания конфигураций текстового поиска они строятся из более простых объектов. В Postgres Pro есть четыре типа таких объектов:

- *Анализаторы текстового поиска* разделяют документ на фрагменты и классифицируют их (например, как слова или числа).
- *Словари текстового поиска* приводят фрагменты к нормализованной форме и отбрасывают стоп-слова.
- *Шаблоны текстового поиска* предоставляют функции, образующие реализацию словарей. (При создании словаря просто задаётся шаблон и набор параметров для него.)
- *Конфигурации текстового поиска* выбирают анализатор и набор словарей, который будет использоваться для нормализации фрагментов, выданных анализатором.

Анализаторы и шаблоны текстового поиска строятся из низкоуровневых функций на языке C; чтобы создать их, нужно программировать на C, а подключить их к базе данных может только суперпользователь. (В подкаталоге `contrib/` инсталляции Postgres Pro можно найти примеры дополнительных анализаторов и шаблонов.) Так как словари и конфигурации представляют собой просто наборы параметров, связывающие анализаторы и шаблоны, их можно создавать, не имея административных прав. Далее в этой главе будут приведены примеры их создания.

12.2. Таблицы и индексы

В предыдущем разделе приводились примеры, которые показывали, как можно выполнить сопоставление с простыми текстовыми константами. В этом разделе показывается, как находить текст в таблице, возможно с применением индексов.

12.2.1. Поиск в таблице

Полнотекстовый поиск можно выполнить, не применяя индекс. Следующий простой запрос выводит заголовок (`title`) каждой строки, содержащей слово `friend` в поле `body`:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

При этом также будут найдены связанные слова, такие как `friends` и `friendly`, так как все они сводятся к одной нормализованной лексеме.

В показанном выше примере для разбора и нормализации строки явно выбирается конфигурация `english`. Хотя параметры, задающие конфигурацию, можно опустить:

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

Такой запрос будет использовать конфигурацию, заданную в параметре default_text_search_config.

В следующем более сложном примере выбираются десять документов, изменённых последними, со словами `create` и `table` в полях `title` или `body`:

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

Чтобы найти строки, содержащие `NULL` в одном из полей, нужно воспользоваться функцией `coalesce`, но здесь мы опустили её вызовы для краткости.

Хотя такие запросы будут работать и без индекса, для большинства приложений скорость будет неприемлемой; этот подход рекомендуется только для нерегулярного поиска и динамического содержимого. Для практического применения полнотекстового поиска обычно создаются индексы.

12.2.2. Создание индексов

Для ускорения текстового поиска мы можем создать индекс GIN (см. Раздел 12.9):

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', body));
```

Заметьте, что здесь используется функция `to_tsvector` с двумя аргументами. В выражениях, определяющих индексы, можно использовать только функции, в которых явно задаётся имя конфигурации текстового поиска (см. Раздел 11.7). Это объясняется тем, что содержимое индекса не должно зависеть от значения параметра default_text_search_config. В противном случае содержимое индекса может быть неактуальным, если разные его элементы `tsvector` будут создаваться с разными конфигурациями текстового поиска и нельзя будет понять, какую именно использовать. Выгрузить и восстановить такой индекс будет невозможно.

Так как при создании индекса использовалась версия `to_tsvector` с двумя аргументами, этот индекс будет использоваться только в запросах, где `to_tsvector` вызывается с двумя аргументами и во втором передаётся имя той же конфигурации. То есть, `WHERE to_tsvector('english', body) @@ 'a & b'` сможет использовать этот индекс, а `WHERE to_tsvector(body) @@ 'a & b'` — нет. Это гарантирует, что индекс будет использоваться только с той конфигурацией, с которой создавались его элементы.

Индекс можно создать более сложным образом, определив для него имя конфигурации в другом столбце таблицы, например:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector(config_name, body));
```

где `config_name` — имя столбца в таблице `pgweb`. Так можно сохранить имя конфигурации, связанной с элементом индекса, и, таким образом, иметь в одном индексе элементы с разными конфигурациями. Это может быть полезно, например, когда в коллекции документов хранятся документы на разных языках. И в этом случае в запросах должен использоваться тот же индекс (с таким же образом задаваемой конфигурацией), например, так: `WHERE to_tsvector(config_name, body) @@ 'a & b'`.

Индексы могут создаваться даже по объединению столбцов:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', title || ' ' ||
 body));
```

Ещё один вариант — создать отдельный столбец `tsvector`, в котором сохранить результат `to_tsvector`. Чтобы этот столбец автоматически синхронизировался с исходными данными, он создаётся как сохранённый генерируемый столбец. Следующий пример показывает, как можно подготовить для индексации объединённое содержимое столбцов `title` и `body`, применив функцию `coalesce` для получения желаемого результата, даже когда один из столбцов `NULL`:

```
ALTER TABLE pgweb
    ADD COLUMN textsearchable_index_col tsvector
               GENERATED ALWAYS AS (to_tsvector('english', coalesce(title, '') || ' '
 || coalesce(body, ''))) STORED;
```

Затем мы создаём индекс GIN для ускорения поиска:

```
CREATE INDEX textsearch_idx ON pgweb USING GIN (textsearchable_index_col);
```

Теперь мы можем быстро выполнять полнотекстовый поиск:

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

Хранение вычисленного выражения индекса в отдельном столбце даёт ряд преимуществ. Во-первых, для использования индекса в запросах не нужно явно указывать имя конфигурации текстового поиска. Как показано в вышеприведённом примере, в этом случае запрос может зависеть от `default_text_search_config`. Во-вторых, поиск выполняется быстрее, так как для проверки соответствия данных индексу не нужно повторно выполнять `to_tsvector`. (Это актуально больше для индексов GiST, чем для GIN; см. Раздел 12.9.) С другой стороны, схему с индексом по выражению проще реализовать и она позволяет сэкономить место на диске, так как представление `tsvector` не хранится явно.

12.3. Управление текстовым поиском

Для реализации полнотекстового поиска необходимы функции, позволяющие создать `tsvector` из документа и `tsquery` из запроса пользователя. Кроме того, результаты нужно выдавать в удобном порядке, так что нам потребуется функция, оценивающая релевантность документа для данного запроса. Важно также иметь возможность выводить найденный текст подходящим образом. В Postgres Pro есть все необходимые для этого функции.

12.3.1. Разбор документов

Для преобразования документа в тип `tsvector` Postgres Pro предоставляет функцию `to_tsvector`.

`to_tsvector([`*`конфигурация`*` regconfig,] `*`документ`*` text) returns tsvector`

`to_tsvector` разбирает текстовый документ на фрагменты, сводит фрагменты к лексемам и возвращает значение `tsvector`, в котором перечисляются лексемы и их позиции в документе. При обработке документа используется указанная конфигурация текстового поиска или конфигурация по умолчанию. Простой пример:

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

В этом примере мы видим, что результирующий `tsvector` не содержит слова `a`, `on` и `it`, слово `rats` превратилось `rat`, а знак препинания «`-`» был проигнорирован.

Функция `to_tsvector` внутри вызывает анализатор, который разбивает текст документа на фрагменты и классифицирует их. Для каждого фрагмента она проверяет список словарей (Раздел 12.6), определяемый типом фрагмента. Первый же словарь, *распознавший* фрагмент, выдаёт одну или несколько представляющих его *лексем*. Например, `rats` превращается в `rat`, так как один из словарей понимает, что слово `rats` — это слово `rat` во множественном числе. Некоторое слова распознаются как *стоп-слова* (Подраздел 12.6.1) и игнорируются как слова, фигурирующие в тексте настолько часто, что искать их бессмысленно. В нашем примере это `a`, `on` и `it`. Если фрагмент не воспринимается ни одним словарём из списка, он так же игнорируется. В данном примере это происходит со знаком препинания `-`, так как с таким типом фрагмента (`символы-разделители`) не связан никакой словарь и значит такие фрагменты никогда не будут индексироваться. Выбор анализатора, словарей и индексируемых типов фрагментов определяется конфигурацией текстового поиска (Раздел 12.7). В одной базе данных можно использовать разные конфигурации, в том числе, предопределённые конфигурации для разных языков. В нашем примере мы использовали конфигурацию по умолчанию для английского языка — `english`.

Для назначения элементам `tsvector` разных *весов* используется функция `setweight`. Вес элемента задаётся буквой `A`, `B`, `C` или `D`. Обычно это применяется для обозначения важности слов в разных частях документа, например в заголовке или в теле документа. Затем эта информация может использоваться при ранжировании результатов поиска.

Так как `to_tsvector(NULL)` вернёт `NULL`, мы советуем использовать `coalesce` везде, где соответствующее поле может быть NULL. Создавать `tsvector` из структурированного документа рекомендуется так:

```
UPDATE tt SET ti =
```

```
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

Здесь мы использовали `setweight` для пометки происхождения каждой лексемы в сформированных значениях `tsvector` и объединили помеченные значения с помощью оператора конкатенации типов `tsvector ||`. (Подробнее эти операции рассматриваются в Подразделе 12.4.1.)

12.3.2. Разбор запросов

Postgres Pro предоставляет функции `to_tsquery`, `plainto_tsquery`, `phraseto_tsquery` и `websearch_to_tsquery` для приведения запроса к типу `tsquery`. Функция `to_tsquery` даёт больше возможностей, чем `plainto_tsquery` или `phraseto_tsquery`, но более строга к входным данным. Функция `websearch_to_tsquery` представляет собой упрощённую версию `to_tsquery` с альтернативным синтаксисом, подобным тому, что принят в поисковых системах в Интернете.

`to_tsquery([`*`конфигурация`*` regconfig,] `*`текст_запроса`*` text) returns tsquery`

`to_tsquery` создаёт значение `tsquery` из *`текста_запроса`*, который может состоять из простых фрагментов, разделённых логическими операторами `tsquery`: `&` (И), `|` (ИЛИ), `!` (НЕ) и `<->` (ПРЕДШЕСТВУЕТ), возможно, сгруппированных скобками. Другими словами, входное значение для `to_tsquery` должно уже соответствовать общим правилам для значений `tsquery`, описанным в Подразделе 8.11.2. Различие их состоит в том, что во вводимом в `tsquery` значении фрагменты воспринимаются буквально, тогда как `to_tsquery` нормализует фрагменты, приводя их к лексемам, используя явно указанную или подразумеваемую конфигурацию, и отбрасывая стоп-слова. Например:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

Как и при вводе значения `tsquery`, для каждой лексемы можно задать вес(а), чтобы при поиске можно было выбрать из `tsvector` только лексемы с заданными весами. Например:

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

К лексеме также можно добавить `*`, определив таким образом условие поиска по префиксу:

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

Такая лексема будет соответствовать любому слову в `tsvector`, начинающемуся с данной подстроки.

`to_tsquery` может также принимать фразы в апострофах. Это полезно в основном когда конфигурация включает тезаурус, который может обрабатывать такие фразы. В показанном ниже примере предполагается, что тезаурус содержит правило `supernovae stars : sn`:

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

Если убрать эти апострофы, `to_tsquery` не примет фрагменты, не разделённые операторами И, ИЛИ и ПРЕДШЕСТВУЕТ, и выдаст синтаксическую ошибку.

`plainto_tsquery([`*`конфигурация`*` regconfig,] `*`текст_запроса`*` text) returns tsquery`

`plainto_tsquery` преобразует неформатированный *`текст_запроса`* в значение `tsquery`. Текст разбирается и нормализуется подобно тому, как это делает `to_tsvector`, а затем между оставшимися словами вставляются операторы `&` (И) типа `tsquery`.

Пример:

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

Заметьте, что `plainto_tsquery` не распознает во входной строке операторы `tsquery`, метки весов или обозначения префиксов:

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

В данном случае все знаки пунктуации были отброшены.

`phraseto_tsquery([`*`конфигурация`*` regconfig,] `*`текст_запроса`*` text) returns tsquery`

`phraseto_tsquery` ведёт себя подобно `plainto_tsquery`, за исключением того, что она вставляет между оставшимися словами оператор `<->` (ПРЕДШЕСТВУЕТ) вместо оператора `&` (И). Кроме того, стоп-слова не просто отбрасываются, а подсчитываются, и вместо операторов `<->` используются операторы `<`*`N`*`>` с подсчитанным числом. Эта функция полезна при поиске точных последовательностей лексем, так как операторы ПРЕДШЕСТВУЕТ проверяют не только наличие всех лексем, но и их порядок.

Пример:

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
```

Как и `plainto_tsquery`, функция `phraseto_tsquery` не распознает во входной строке операторы типа `tsquery`, метки весов или обозначения префиксов:

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
-----------------------------
 'fat' <-> 'rat' <-> 'c'
```

`websearch_to_tsquery([`*`конфигурация`*` regconfig,] `*`текст_запроса`*` text) returns tsquery`

Функция `websearch_to_tsquery` создаёт значение `tsquery` из *`текста_запроса`*, используя альтернативный синтаксис, в котором запрос задаётся просто неформатированным текстом. В отличие от `plainto_tsquery` и `phraseto_tsquery`, она также принимает определённые операторы. Более того, эта функция никогда не выдаёт синтаксические ошибки, что позволяет осуществлять поиск по произвольному заданному пользователем запросу. Она поддерживает следующий синтаксис:

- `текст не в кавычках`: текст, не заключённый в кавычки, который будет преобразован в слова, разделяемые операторами `&`, как его восприняла бы функция `plainto_tsquery`.
- `"текст в кавычках"`: текст, заключённый в кавычки, будет преобразован в слова, разделяемые операторами `<->`, как его восприняла бы функция `phraseto_tsquery`.
- `OR`: слово «or» будет преобразовано в оператор `|`.
- `-`: знак минуса будет преобразован в оператор `!`.

Другая пунктуация игнорируется. Поэтому функция `websearch_to_tsquery`, как и `plainto_tsquery` с `phraseto_tsquery`, не распознаёт во входной строке операторы `tsquery`, метки весов или обозначения префиксов.

Примеры:

```
SELECT websearch_to_tsquery('english', 'The fat rats');
 websearch_to_tsquery
----------------------
 'fat' & 'rat'
(1 row)

SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
       websearch_to_tsquery
----------------------------------
 'supernova' <-> 'star' & !'crab'
(1 row)

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
       websearch_to_tsquery
-----------------------------------
 'sad' <-> 'cat' | 'fat' <-> 'rat'
(1 row)

SELECT websearch_to_tsquery('english', 'signal -"segmentation fault"');
         websearch_to_tsquery
---------------------------------------
 'signal' & !( 'segment' <-> 'fault' )
(1 row)

SELECT websearch_to_tsquery('english', '""" )( dummy \\ query <->');
 websearch_to_tsquery
----------------------
 'dummi' & 'queri'
(1 row)
```

12.3.3. Ранжирование результатов поиска

Ранжирование документов можно представить как попытку оценить, насколько они релевантны заданному запросу и отсортировать их так, чтобы наиболее релевантные выводились первыми. В Postgres Pro встроены две функции ранжирования, принимающие во внимание лексическую, позиционную и структурную информацию; то есть, они учитывают, насколько часто и насколько близко встречаются в документе ключевые слова и какова важность содержащей их части документа. Однако само понятие релевантности довольно размытое и во многом определяется приложением. Приложения могут использовать для ранжирования и другую информацию, например, время изменения документа. Встроенные функции ранжирования можно рассматривать лишь как примеры реализации. Для своих конкретных задач вы можете разработать собственные функции ранжирования и/или учесть при обработке их результатов дополнительные факторы.

Ниже описаны две встроенные функции ранжирования:

`ts_rank([`*`веса`*` float4[],] `*`вектор`*` tsvector, `*`query`*` tsquery [, `*`нормализация`*` integer]) returns float4`

Ранжирует векторы по частоте найденных лексем.

`ts_rank_cd([`*`веса`*` float4[],] `*`вектор`*` tsvector, `*`query`*` tsquery [, `*`нормализация`*` integer]) returns float4`

Эта функция вычисляет *плотность покрытия* для данного вектора документа и запроса, используя метод, разработанный Кларком, Кормаком и Тадхоуп и описанный в статье «Relevance

Ranking for One to Three Term Queries» в журнале «Information Processing and Management» в 1999 г. Плотность покрытия вычисляется подобно рангу `ts_rank`, но в расчёт берётся ещё и близость соответствующих лексем друг к другу.

Для вычисления результата этой функции требуется информация о позиции лексем. Поэтому она игнорируют «очищенные» от этой информации лексемы в `tsvector`. Если во входных данных нет неочищенных лексем, результат будет равен нулю. (За дополнительными сведениями о функции `strip` и позиционной информации в данных `tsvector` обратитесь к Подразделу 12.4.1.)

Для обеих этих функций аргумент *веса* позволяет придать больший или меньший вес словам, в зависимости от их меток. В передаваемом массиве весов определяется, насколько весома каждая категория слов, в следующем порядке:

```
{вес D, вес C, вес B, вес A}
```

Если этот аргумент опускается, подразумеваются следующие значения:

```
{0.1, 0.2, 0.4, 1.0}
```

Обычно весами выделяются слова из особых областей документа, например из заголовка или краткого введения, с тем, чтобы эти слова считались более и менее значимыми, чем слова в основном тексте документа.

Так как вероятность найти ключевые слова увеличивается с размером документа, при ранжировании имеет смысл учитывать его, чтобы, например, документ с сотней слов, содержащий пять вхождений искомых слов, считался более релевантным, чем документ с тысячей слов и теми же пятью вхождениями. Обе функции ранжирования принимают целочисленный параметр *нормализации*, определяющий, как ранг документа будет зависеть от его размера. Этот параметр представляет собой битовую маску и управляет несколькими режимами: вы можете включить сразу несколько режимов, объединив значения оператором `|` (например так: `2|4`).

- 0 (по умолчанию): длина документа не учитывается
- 1: ранг документа делится на 1 + логарифм длины документа
- 2: ранг документа делится на его длину
- 4: ранг документа делится на среднее гармоническое расстояние между блоками (это реализовано только в `ts_rank_cd`)
- 8: ранг документа делится на число уникальных слов в документе
- 16: ранг документа делится на 1 + логарифм числа уникальных слов в документе
- 32: ранг делится своё же значение + 1

Если включены несколько флагов, соответствующие операции выполняются в показанном порядке.

Важно заметить, что функции ранжирования не используют никакую внешнюю информацию, так что добиться нормализации до 1% или 100% невозможно, хотя иногда это желательно. Применив параметр 32 (`rank/(rank+1)`), можно свести все ранги к диапазону 0..1, но это изменение будет лишь косметическим, на порядке сортировки результатов это не отразится.

В данном примере выбираются десять найденных документов с максимальным рангом:

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
```

```
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

Тот же пример с нормализованным рангом:

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

Ранжирование может быть довольно дорогостоящей операцией, так как для вычисления ранга необходимо прочитать `tsvector` каждого подходящего документа и это займёт значительное время, если придётся обращаться к диску. К сожалению, избежать этого вряд ли возможно, так как на практике по многим запросам выдаётся большое количество результатов.

12.3.4. Выделение результатов

Представляя результаты поиска, в идеале нужно выделять часть документа и показывать, как он связан с запросом. Обычно поисковые системы показывают фрагменты документа с отмеченными искомыми словами. В Postgres Pro для реализации этой возможности представлена функция `ts_headline`.

```
ts_headline([конфигурация regconfig,] документ text, query tsquery [, параметры text])
 returns text
```

`ts_headline` принимает документ вместе с запросом и возвращает выдержку из документа, в которой выделяются слова из запроса. В частности, функция будет использовать запрос для выбора соответствующих фрагментов текста, а затем выделять все слова, которые встречаются в запросе, даже если позиции этих слов не соответствуют ограничениям запроса. Применяемую для разбора документа конфигурацию можно указать в параметре *`config`*. Если этот параметр опущен, применяется конфигурация `default_text_search_config`.

Если в параметрах передаётся строка *`options`*, она должна состоять из списка разделённых запятыми пар *`параметр`*=*`значение`*. Параметры могут быть следующими:

- `MaxWords`, `MinWords` (целочисленные): эти числа определяют нижний и верхний предел размера выдержки. Значения по умолчанию: 35 и 15, соответственно.
- `ShortWord` (целочисленный): слова такой длины или короче в начале и конце выдержки будут отбрасываться, за исключением искомых слов. Значение по умолчанию, равное 3, исключает распространённые английские артикли.
- `HighlightAll` (логический): при значении `true` выдержкой будет весь документ, и три предыдущие параметра игнорируются. Значение по умолчанию: `false`.

- `MaxFragments` (целочисленный): максимальное число выводимых текстовых фрагментов. Значение по умолчанию, равное нулю, выбирает режим создания выдержек без фрагментов. При положительном значении выбирается режим с фрагментами (см. ниже).
- `StartSel`, `StopSel`: строки, которые будут разграничивать слова запроса в документе, выделяя их среди остальных. Значения по умолчанию «`<b>`» и «`</b>`» подходят для отображения в формате HTML (однако обратите внимание на предупреждение ниже).
- `FragmentDelimiter` (строка): в случае, когда фрагментов несколько, они будут разделяться указанной строкой. Значение по умолчанию: «` ... `».

Предупреждение: защита от межсайтового скриптинга (XSS)

Безопасность вывода функции `ts_headline` для прямого включения в веб-страницы не гарантируется. Если для `HighlightAll` установлено значение `false` (по умолчанию), некоторые простые XML-теги удаляются из документа, но это не гарантирует удаление всей HTML-разметки. Таким образом это не даёт эффективную защиту от таких атак, как межсайтовый скриптинг (XSS), при работе с недоверенными вводными данными. Для защиты от таких атак из передаваемого на вход документа необходимо удалять всю HTML-разметку или применять средства очистки HTML для вывода.

Имена этих параметров распознаются без учёта регистра. Значения, содержащие пробелы или запятые, должны заключаться в двойные кавычки.

В режиме формирования выдержек без фрагментов `ts_headline` находит вхождения слов заданного *запроса* и возвращает одно из вхождений, предпочитая те, что содержат как можно больше слов из запроса в пределах допустимого размера выдержки. В режиме с фрагментами `ts_headline` находит вхождения слов запроса и разделяет эти вхождения на «фрагменты», состоящие не более чем из `MaxWords` слов, предпочитая те, что содержат больше искомых слов, а затем может «растянуть» фрагменты, добавив в них соседние слова. Второй режим полезнее, когда слова запроса находятся не рядом, а разбросаны по документу, или когда желательно увидеть сразу несколько вхождений. В случаях, когда соответствие запросу найти не удаётся, в обоих режимах возвращаются первые `MinWords` слов из документа.

Пример использования:

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('english', 'query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms                       +
 and return them in order of their <b>similarity</b> to the+
 <b>query</b>.

SELECT ts_headline('english',
  'Search terms may occur
many times in a document,
requiring ranking of the search matches to decide which
occurrences to display in the result.',
  to_tsquery('english', 'search & term'),
  'MaxFragments=10, MaxWords=7, MinWords=3, StartSel=<<, StopSel=>>');
                        ts_headline
------------------------------------------------------------
 <<Search>> <<terms>> may occur                            +
```

```
 many times ... ranking of the <<search>> matches to decide
```

Функция `ts_headline` работает с оригинальным документом, а не его сжатым представлением `tsvector`, так что она может быть медленной и использовать её следует осмотрительно.

12.4. Дополнительные возможности

В этом разделе описываются дополнительные функции и операторы, которые могут быть полезны при поиске текста.

12.4.1. Обработка документов

В Подразделе 12.3.1 показывалось, как обычные текстовые документы можно преобразовать в значения `tsvector`. Postgres Pro предлагает также набор функций и операторов для обработки документов, уже представленных в формате `tsvector`.

`tsvector || tsvector`

Оператор конкатенации значений `tsvector` возвращает вектор, объединяющий лексемы и позиционную информацию двух векторов, переданных ему в аргументах. В полученном результате сохраняются позиции и метки весов. При этом позиции в векторе справа сдвигаются на максимальное значение позиции в векторе слева, что почти равносильно применению `to_tsvector` к результату конкатенации двух исходных строк документов. (Почти, потому что стоп-слова, исключаемые в конце левого аргумента, при конкатенации исходных строк влияют на позиции лексем в правой части, а при конкатенации `tsvector` — нет.)

Преимущество же конкатенации документов в векторной форме по сравнению с конкатенацией текста до вызова `to_tsvector` заключается в том, что так можно разбирать разные части документа, применяя разные конфигурации. И так как функция `setweight` помечает все лексемы данного вектора одинаково, разбирать текст и выполнять `setweight` нужно до объединения разных частей документа с подразумеваемым разным весом.

`setweight(`*`вектор`*` tsvector, `*`вес`*` "char") returns tsvector`

`setweight` возвращает копию входного вектора, помечая в ней каждую позицию заданным *весом*, меткой `A`, `B`, `C` или `D`. (Метка `D` по умолчанию назначается всем векторам, так что при выводе она опускается.) Эти метки сохраняются при конкатенации векторов, что позволяет придавать разные веса словам из разных частей документа и, как следствие, ранжировать их по-разному.

Заметьте, что веса назначаются *позициям*, а не *лексемам*. Если входной вектор очищен от позиционной информации, `setweight` не делает ничего.

`length(`*`вектор`*` tsvector) returns integer`

Возвращает число лексем, сохранённых в векторе.

`strip(`*`вектор`*` tsvector) returns tsvector`

Возвращает вектор с теми же лексемами, что и в данном, но без информации о позиции и весе. Очищенный вектор обычно оказывается намного меньше исходного, но при этом и менее полезным. С очищенными векторами хуже работает ранжирование, а также оператор `<->` (ПРЕДШЕСТВУЕТ) типа `tsquery` никогда не найдёт соответствие в них, так как не сможет определить расстояние между вхождениями лексем.

Полный список связанных с `tsvector` функций приведён в Таблице 9.43.

12.4.2. Обработка запросов

В Подразделе 12.3.2 показывалось, как обычные текстовые запросы можно преобразовывать в значения `tsquery`. Postgres Pro предлагает также набор функций и операторов для обработки запросов, уже представленных в формате `tsquery`.

`tsquery && tsquery`

Возвращает логическое произведение (AND) двух данных запросов.

`tsquery || tsquery`

Возвращает логическое объединение (OR) двух данных запросов.

`!! tsquery`

Возвращает логическое отрицание (NOT) данного запроса.

`tsquery <-> tsquery`

Возвращает запрос, который ищет соответствие первому данному запросу, за которым следует соответствие второму данному запросу, с применением оператора `<->` (ПРЕДШЕСТВУЕТ) типа `tsquery`. Например:

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
          ?column?
----------------------------
 'fat' <-> ( 'cat' | 'rat' )
```

`tsquery_phrase(`*`запрос1`*` tsquery, `*`запрос2`*` tsquery [, `*`расстояние`*` integer ]) returns tsquery`

Возвращает запрос, который ищет соответствие первому данному запросу, за которым следует соответствие второму данному запросу (точное число лексем между ними задаётся параметром *`расстояние`*), с применением оператора `<N>` типа `tsquery`. Например:

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
  tsquery_phrase
------------------
 'fat' <10> 'cat'
```

`numnode(`*`query`*` tsquery) returns integer`

Возвращает число узлов (лексем и операторов) в значении `tsquery`. Эта функция помогает определить, имеет ли смысл *`запрос`* (тогда её результат > 0) или он содержит только стоп-слова (тогда она возвращает 0). Примеры:

```
SELECT numnode(plainto_tsquery('the any'));
ЗАМЕЧАНИЕ:  запрос поиска текста игнорируется, так как содержит только стоп-слова
 или не содержит лексем
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

`querytree(`*`query`*` tsquery) returns text`

Возвращает часть `tsquery`, которую можно использовать для поиска по индексу. Эта функция помогает выявить неиндексируемые запросы, к примеру такие, которые содержат только стоп-слова или условия отрицания. Например:

```
SELECT querytree(to_tsquery('defined'));
 querytree
-----------
 'defin'
```

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
 T
```

12.4.2.1. Перезапись запросов

Семейство запросов `ts_rewrite` ищет в данном `tsquery` вхождения целевого подзапроса и заменяет каждое вхождение указанной подстановкой. По сути эта операция похожа на замену подстроки в строке, только рассчитана на работу с `tsquery`. Сочетание целевого подзапроса с подстановкой можно считать *правилом перезаписи запроса*. Набор таких правил перезаписи может быть очень полезен при поиске. Например, вы можете улучшить результаты, добавив синонимы (например, `big apple`, `nyc` и `gotham` для `new york`) или сузить область поиска, чтобы нацелить пользователя на некоторую область. Это в некотором смысле пересекается с функциональностью тезаурусов (Подраздел 12.6.4). Однако при таком подходе вы можете изменять правила перезаписи «на лету», тогда как при обновлении тезауруса необходима переиндексация.

`ts_rewrite (`*`query`*` tsquery, `*`цель`*` tsquery, `*`замена`*` tsquery) returns tsquery`

Эта форма `ts_rewrite` просто применяет одно правило перезаписи: *`цель`* заменяется *`подстановкой`* везде, где она находится в *`запросе`*. Например:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

`ts_rewrite (`*`query`*` tsquery, `*`выборка`*` text) returns tsquery`

Эта форма `ts_rewrite` принимает начальный *`запрос`* и SQL-команду *`select`*, которая задаётся текстовой строкой. Команда *`select`* должна выдавать два столбца типа `tsquery`. Для каждой строки результата *`select`* вхождения первого столбца (цели) заменяются значениями второго столбца (подстановкой) в тексте *`запроса`*. Например:

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

Заметьте, что когда таким способом применяются несколько правил перезаписи, порядок их применения может иметь значение, поэтому в исходном запросе следует добавить `ORDER BY` по какому-либо ключу.

Давайте рассмотрим практический пример на тему астрономии. Мы развернём запрос `supernovae`, используя правила перезаписи в таблице:

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
           ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

Мы можем скорректировать правила перезаписи, просто изменив таблицу:

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');
```

```
SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                 ts_rewrite
---------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

Перезапись может быть медленной, когда задано много правил перезаписи, так как соответствия будут проверяться для каждого правила. Чтобы отфильтровать явно неподходящие правила, можно использовать проверки включения для типа `tsquery`. В следующем примере выбираются только те правила, которые могут соответствовать исходному запросу:

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. Триггеры для автоматического обновления

Примечание

Описанный в этом разделе подход считается устаревшим в связи с появлением возможности использовать генерируемые столбцы, как описано в Подразделе 12.2.2.

Когда представление документа в формате `tsvector` хранится в отдельном столбце, необходимо создать триггер, который будет обновлять его содержимое при изменении столбцов, из которых составляется исходный документ. Для этого можно использовать две встроенные триггерные функции или написать свои собственные.

```
tsvector_update_trigger(столбец_tsvector, имя_конфигурации, столбец_текста [, ...])
tsvector_update_trigger_column(столбец_tsvector, столбец_конфигурации, столбец_текста
 [, ...])
```

Эти триггерные функции автоматически вычисляют значение для столбца `tsvector` из одного или нескольких текстовых столбцов с параметрами, указанными в команде `CREATE TRIGGER`. Пример их использования:

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title    |         body
------------+-----------------------
 title here | the body text is here
```

С таким триггером любое изменение в полях `title` или `body` будет автоматически отражаться в содержимом `tsv`, так что приложению не придётся заниматься этим.

Первым аргументом этих функций должно быть имя столбца `tsvector`, содержимое которого будет обновляться. Ещё один аргумент — конфигурация текстового поиска, которая будет использоваться для преобразования. Для `tsvector_update_trigger` имя конфигурации передаётся просто как второй аргумент триггера. Это имя должно быть определено полностью, чтобы поведение триггера не менялось при изменениях в пути поиска (`search_path`). Для `tsvector_update_trigger_column` во втором аргументе триггера передаётся имя другого столбца таблицы, который должен иметь тип `regconfig`. Это позволяет использовать разные конфигурации для разных строк. В оставшихся аргументах передаются имена текстовых столбцов (типа `text`, `varchar` или `char`). Их содержимое будет включено в документ в заданном порядке. При этом значения NULL будут пропущены (а другие столбцы будут индексироваться).

Ограничение этих встроенных триггеров заключается в том, что они обрабатывают все столбцы одинаково. Чтобы столбцы обрабатывались по-разному, например для текста заголовка задавался не тот же вес, что для тела документа, потребуется разработать свой триггер. К примеру, так это можно сделать на языке PL/pgSQL:

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')), 'A') ||
     setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();
```

Помните, что, создавая значения `tsvector` в триггерах, важно явно указывать имя конфигурации, чтобы содержимое столбца не зависело от изменений `default_text_search_config`. В противном случае могут возникнуть проблемы, например результаты поиска изменятся после выгрузки и восстановления данных.

12.4.4. Сбор статистики по документу

Функция `ts_stat` может быть полезна для проверки конфигурации и нахождения возможных стоп-слов.

```
ts_stat(sql_запрос text, [веса text,]
        OUT слово text, OUT число_док integer,
        OUT число_вхожд integer) returns setof record
```

Здесь *sql_запрос* — текстовая строка, содержащая SQL-запрос, который должен возвращать один столбец `tsvector`. Функция `ts_stat` выполняет запрос и возвращает статистику по каждой отдельной лексеме (слову), содержащейся в данных `tsvector`. Её результат представляется в столбцах

- *слово* `text` — значение лексемы
- *число_док* `integer` — число документов (значений `tsvector`), в которых встретилось слово
- *число_вхожд* `integer` — общее число вхождений слова

Если передаётся параметр *weights*, то подсчитываются только вхождения с указанными в нём весами.

Например, найти десять наиболее часто используемых слов в коллекции документов можно так:

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

Следующий запрос возвращает тоже десять слов, но при выборе их учитываются только вхождения с весами `A` или `B`:

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. Анализаторы

Задача анализаторов текста — разделить текст документа на *фрагменты* и присвоить каждому из них тип из набора, определённого в самом анализаторе. Заметьте, что анализаторы не меняют текст — они просто выдают позиции предполагаемых слов. Вследствие такой ограниченности их функций, собственные специфические анализаторы бывают нужны гораздо реже, чем собственные словари. В настоящее время в Postgres Pro есть только один встроенный анализатор, который может быть полезен для широкого круга приложений.

Этот встроенный анализатор называется `pg_catalog.default`. Он распознаёт 23 типа фрагментов, перечисленные в Таблице 12.1.

Таблица 12.1. Типы фрагментов, выделяемых стандартным анализатором

Псевдоним	Описание	Пример
`asciiword`	Слово только из букв ASCII	`elephant`
`word`	Слово из любых букв	`mañana`
`numword`	Слово из букв и цифр	`beta1`
`asciihword`	Слово только из букв ASCII с дефисами	`up-to-date`
`hword`	Слово из любых букв с дефисами	`lógico-matemática`
`numhword`	Слово из букв и цифр с дефисами	`postgresql-beta1`
`hword_asciipart`	Часть слова с дефисами, только из букв ASCII	`postgresql` в словосочетании `postgresql-beta1`
`hword_part`	Часть слова с дефисами, из любых букв	`lógico` или `matemática` в словосочетании `lógico-matemática`
`hword_numpart`	Часть слова с дефисами, из букв и цифр	`beta1` в словосочетании `postgresql-beta1`
`email`	Адрес электронной почты	`foo@example.com`
`protocol`	Префикс протокола	`http://`
`url`	URL	`example.com/stuff/index.html`
`host`	Имя узла	`example.com`
`url_path`	Путь в адресе URL	`/stuff/index.html`, как часть URL
`file`	Путь или имя файла	`/usr/local/foo.txt`, если не является частью URL
`sfloat`	Научная запись числа	`-1.234e56`
`float`	Десятичная запись числа	`-1.234`
`int`	Целое со знаком	`-1234`
`uint`	Целое без знака	`1234`
`version`	Номер версии	`8.3.0`
`tag`	Тег XML	`<a href="dictionaries.html">`

Псевдоним	**Описание**	**Пример**
`entity`	Сущность XML	`&`
`blank`	Символы-разделители	(любые пробельные символы или знаки препинания, не попавшие в другие категории)

Примечание

Понятие «буквы» анализатор определяет исходя из локали, заданной для базы данных, в частности параметра `lc_ctype`. Слова, содержащие только буквы из ASCII (латинские буквы), распознаются как фрагменты отдельного типа, так как иногда бывает полезно выделить их. Для многих европейских языков типы фрагментов `word` и `asciiword` можно воспринимать как синонимы.

`email` принимает не все символы, которые считаются допустимыми по стандарту *RFC 5322*. В частности, имя почтового ящика помимо алфавитно-цифровых символов может содержать только точку, минус и подчёркивание.

`tag` поддерживает не все действительные имена тегов, определённые в *Рекомендации W3C по XML*. А именно поддерживаются только имена тегов, которые начинаются с буквы из ASCII, знака подчёркивания или двоеточия и содержат только буквы, цифры, дефисы, знаки подчёркивания и двоеточия. `tag` также включает комментарии XML, начинающиеся с `<!--` и заканчивающиеся `-->`, и объявления XML (однако обратите внимание, что сюда входит всё, что начинается с `<?x` и заканчивается `>`).

Анализатор может выделить в одном тексте несколько перекрывающихся фрагментов. Например, слово с дефисом будет выдано как целое составное слово и по частям:

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+--------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

Это поведение считается желательным, так как это позволяет находить при последующем поиске и всё слово целиком, и его части. Ещё один показательный пример:

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. Словари

Словари полнотекстового поиска предназначены для исключения *стоп-слов* (слов, которые не должны учитываться при поиске) и *нормализации* слов, чтобы разные словоформы считались совпадающими. Успешно нормализованное слово называется *лексемой*. Нормализация и исключение стоп-слов не только улучшает качество поиска, но и уменьшает размер представления документа в формате `tsvector`, и, как следствие, увеличивает быстродействие. Нормализация не всегда имеет лингвистический смысл, обычно она зависит от требований приложения.

Несколько примеров нормализации:

- Лингвистическая нормализация — словари Ispell пытаются свести слова на входе к нормализованной форме, а стеммеры убирают окончания слов
- Адреса URL могут быть канонизированы, чтобы например следующие адреса считались одинаковыми:
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- Названия цветов могут быть заменены их шестнадцатеричными значениями, например `red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- При индексировании чисел можно отбросить цифры в дробной части для сокращения множества всевозможных чисел, чтобы например *3.14*159265359, *3.14*15926 и *3.14* стали одинаковыми после нормализации, при которой после точки останутся только две цифры.

Словарь — это программа, которая принимает на вход фрагмент и возвращает:

- массив лексем, если входной фрагмент известен в словаре (заметьте, один фрагмент может породить несколько лексем)
- одну лексему с установленным флагом `TSL_FILTER` для замены исходного фрагмента новым, чтобы следующие словари работали с новым вариантом (словарь, который делает это, называется *фильтрующим словарём*)
- пустой массив, если словарь воспринимает этот фрагмент, но считает его стоп-словом
- `NULL`, если словарь не воспринимает полученный фрагмент

В Postgres Pro встроены стандартные словари для многих языков. Есть также несколько предопределённых шаблонов, на основании которых можно создавать новые словари с изменёнными параметрами. Все эти шаблоны описаны ниже. Если же ни один из них не подходит, можно создать и свои собственные шаблоны. Соответствующие примеры можно найти в каталоге `contrib/` инсталляции Postgres Pro.

Конфигурация текстового поиска связывает анализатор с набором словарей, которые будут обрабатывать выделенные им фрагменты. Для каждого типа фрагментов, выданных анализатором, в конфигурации задаётся отдельный список словарей. Найденный анализатором фрагмент проходит через все словари по порядку, пока какой-либо словарь не увидит в нём знакомое для него слово. Если он окажется стоп-словом или его не распознает ни один словарь, этот фрагмент не будет учитываться при индексации и поиске. Обычно результат определяет первый же словарь, который возвращает не `NULL`, и остальные словари уже не проверяются; однако фильтрующий словарь может заменить полученное слово другим, которое и будет передано следующим словарям.

Общее правило настройки списка словарей заключается в том, чтобы поставить наиболее частные и специфические словари в начале, затем перечислить более общие и закончить самым общим словарём, например стеммером Snowball или словарём `simple`, который распознаёт всё. Например, для поиска по теме астрономии (конфигурация `astro_en`) тип фрагментов `asciiword` (слово из букв ASCII) можно связать со словарём синонимов астрономических терминов, затем с обычным английским словарём и наконец со стеммером английских окончаний Snowball:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

Фильтрующий словарь можно включить в любом месте списка, кроме конца, где он будет бесполезен. Фильтрующие словари бывают полезны для частичной нормализации слов и упрощения задачи следующих словарей. Например, фильтрующий словарь может удалить из текста диакритические знаки, как это делает модуль unaccent.

12.6.1. Стоп-слова

Стоп-словами называются слова, которые встречаются очень часто, практически в каждом документе, и поэтому не имеют различительной ценности. Таким образом, при полнотекстовом поиске их можно игнорировать. Например, в каждом английском тексте содержатся артикли `a` и `the`, так что хранить их в индексе бессмысленно. Однако стоп-слова влияют на позиции лексем в значении `tsvector`, от чего, в свою очередь, зависит ранжирование:

```
SELECT to_tsvector('english', 'in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

В результате отсутствуют позиции 1,2,4, потому что фрагменты в этих позициях оказались стоп-словами. Ранги, вычисленные для документов со стоп-словами и без них, могут значительно различаться:

```
SELECT ts_rank_cd (to_tsvector('english', 'in the list of stop words'),
 to_tsquery('list & stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english', 'list stop words'), to_tsquery('list &
 stop'));
 ts_rank_cd
------------
        0.1
```

Как именно обрабатывать стоп-слова, определяет сам словарь. Например, словари `ispell` сначала нормализуют слова, а затем просматривают список стоп-слов, тогда как стеммеры `Snowball` просматривают свой список стоп-слов в первую очередь. Это различие в поведении объясняется стремлением уменьшить шум.

12.6.2. Простой словарь

Работа шаблона словарей `simple` сводится к преобразованию входного фрагмента в нижний регистр и проверки результата по файлу со списком стоп-слов. Если это слово находится в файле, словарь возвращает пустой массив и фрагмент исключается из дальнейшего рассмотрения. В противном случае словарь возвращает в качестве нормализованной лексемы слово в нижнем регистре. Этот словарь можно настроить и так, чтобы все слова, кроме стоп-слов, считались неопознанными и передавались следующему словарю в списке.

Определить словарь на основе шаблона `simple` можно так:

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

Здесь `english` — базовое имя файла со стоп-словами. Полным именем файла будет `$SHAREDIR/tsearch_data/english.stop`, где `$SHAREDIR` указывает на каталог с общими данными Postgres Pro, часто это `/usr/local/share/postgresql` (точно узнать его можно с помощью команды `pg_config --sharedir`). Этот текстовый файл должен содержать просто список слов, по одному слову в строке. Пустые строки и окружающие пробелы игнорируются, все символы переводятся в нижний регистр и на этом обработка файла заканчивается.

Теперь мы можем проверить наш словарь:

```
SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
```

```
 {}
```

Мы также можем настроить словарь так, чтобы он возвращал `NULL` вместо слова в нижнем регистре, если оно не находится в файле стоп-слов. Для этого нужно присвоить параметру `Accept` значение `false`. Продолжая наш пример:

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

Со значением `Accept` = `true` (по умолчанию) словарь `simple` имеет смысл включать только в конце списка словарей, так как он никогда не передаст фрагмент следующему словарю. И напротив, `Accept` = `false` имеет смысл, только если за ним следует ещё минимум один словарь.

Внимание

Большинство словарей работают с дополнительными файлами, например, файлами стоп-слов. Содержимое этих файлов *должно* иметь кодировку UTF-8. Если база данных работает в другой кодировке, они будут переведены в неё, когда сервер будет загружать их.

Внимание

Обычно в рамках одного сеанса дополнительный файл словаря загружается только один раз, при первом использовании. Если же вы измените его и захотите, чтобы существующие сеансы работали с новым содержимым, выполните для этого словаря команду `ALTER TEXT SEARCH DICTIONARY`. Это обновление словаря может быть «фиктивным», фактически не меняющим значения никаких параметров.

12.6.3. Словарь синонимов

Этот шаблон словарей используется для создания словарей, заменяющих слова синонимами. Словосочетания такие словари не поддерживают (используйте для этого тезаурус (Подраздел 12.6.4)). Словарь синонимов может помочь в преодолении лингвистических проблем, например, не дать стеммеру английского уменьшить слово «Paris» до «pari». Для этого достаточно поместить в словарь синонимов строку `Paris paris` и поставить этот словарь перед словарём `english_stem`. Например:

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
```

```
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias    |   description    | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------
+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
```

Шаблон `synonym` принимает единственный параметр, `SYNONYMS`, в котором задаётся базовое имя его файла конфигурации — в данном примере это `my_synonyms`. Полным именем файла будет `$SHAREDIR/tsearch_data/my_synonyms.syn` (где `$SHAREDIR` указывает на каталог общих данных Postgres Pro). Содержимое этого файла должны составлять строки с двумя словами в каждой (первое — заменяемое слово, а второе — его синоним), разделёнными пробелами. Пустые строки и окружающие пробелы при разборе этого файла игнорируются.

Шаблон `synonym` также принимает необязательный параметр `CaseSensitive`, который по умолчанию имеет значение `false`. Когда `CaseSensitive` равен `false`, слова в файле синонимов переводятся в нижний регистр, вместе с проверяемыми фрагментами. Если же он не равен `true`, регистр слов в файле и проверяемых фрагментов не меняются, они сравниваются «как есть».

В конце синонима в этом файле можно добавить звёздочку (`*`), тогда этот синоним будет рассматриваться как префикс. Эта звёздочка будет игнорироваться в `to_tsvector()`, но `to_tsquery()` изменит результат, добавив в него маркер сопоставления префикса (см. Подраздел 12.3.2). Например, предположим, что файл `$SHAREDIR/tsearch_data/synonym_sample.syn` имеет следующее содержание:

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

С ним мы получим такие результаты:

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn', 'indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst', 'indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst', 'indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
---------------------------------
 'are' 'indexes' 'useful' 'very'
```

```
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst', 'indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. Тезаурус

Тезаурус (или сокращённо TZ) содержит набор слов и информацию о связях слов и словосочетаний, то есть более широкие понятия (Broader Terms, BT), более узкие понятия (Narrow Terms, NT), предпочитаемые названия, исключаемые названия, связанные понятия и т. д.

В основном тезаурус заменяет исключаемые слова и словосочетания предпочитаемыми и может также сохранить исходные слова для индексации. Текущая реализация тезауруса в Postgres Pro представляет собой расширение словаря синонимов с поддержкой *фраз*. Конфигурация тезауруса определяется файлом следующего формата:

```
# это комментарий
образец слов(а) : индексируемые слова
другой образец слов(а) : другие индексируемые слова
...
```

Здесь двоеточие (`:`) служит разделителем между исходной фразой и её заменой.

Прежде чем проверять соответствие фраз, тезаурус нормализует файл конфигурации, используя *внутренний словарь* (который указывается в конфигурации словаря-тезауруса). Этот внутренний словарь для тезауруса может быть только одним. Если он не сможет распознать какое-либо слово, произойдёт ошибка. В этом случае необходимо либо исключить это слово, либо добавить его во внутренний словарь. Также можно добавить звёздочку (`*`) перед индексируемыми словами, чтобы они не проверялись по внутреннему словарю, но все слова-образцы *должны* быть известны внутреннему словарю.

Если входному фрагменту соответствуют несколько фраз в этом списке, тезаурус выберет самое длинное определение, а если таких окажется несколько, самое последнее из них.

Выделить во фразе какие-то стоп-слова нельзя; вместо этого можно вставить `?` в том месте, где может оказаться стоп-слово. Например, в предположении, что `a` и `the` — стоп-слова по внутреннему словарю:

```
? one ? two : swsw
```

соответствует входным строкам `a one the two` и `the one a two`, так что обе эти строки будут заменены на `swsw`.

Как и обычный словарь, тезаурус должен привязываться к лексемам определённых типов. Так как тезаурус может распознавать фразы, он должен запоминать своё состояние и взаимодействовать с анализатором. Учитывая свои привязки, он может либо обрабатывать следующий фрагмент, либо прекратить накопление фразы. Поэтому настройка тезаурусов в системе требует особого внимания. Например, если привязать тезаурус только к типу фрагментов `asciiword`, тогда определение в тезаурусе `one 7` не будет работать, так как этот тезаурус не связан с типом `uint`.

Внимание

Тезаурусы используются при индексации, поэтому при любом изменении параметров или содержимого тезауруса *необходима* переиндексация. Для большинства других типов словарей при небольших изменениях, таких как удаление и добавление стоп-слов, переиндексация не требуется.

12.6.4.1. Конфигурация тезауруса

Для создания нового словаря-тезауруса используется шаблон `thesaurus`. Например:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

Здесь:

- `thesaurus_simple` — имя нового словаря
- `mythesaurus` — базовое имя файла конфигурации тезауруса. (Полным путём к файлу будет `$SHAREDIR/tsearch_data/mythesaurus.ths`, где `$SHAREDIR` указывает на каталог общих данных PostgreSQL.)
- `pg_catalog.english_stem` — внутренний словарь (в данном случае это стеммер Snowball для английского) для нормализации тезауруса. Заметьте, что внутренний словарь имеет собственную конфигурацию (например, список стоп-слов), но здесь она не рассматривается.

Теперь тезаурус `thesaurus_simple` можно связать с желаемыми типами фрагментов в конфигурации, например так:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

12.6.4.2. Пример тезауруса

Давайте рассмотрим простой астрономический тезаурус `thesaurus_astro`, содержащий несколько астрономических терминов:

```
supernovae stars : sn
crab nebulae : crab
```

Ниже мы создадим словарь и привяжем некоторые типы фрагментов к астрономическому тезаурусу и английскому стеммеру:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

Теперь можно проверить, как он работает. Функция `ts_lexize` не очень полезна для проверки тезауруса, так как она обрабатывает входную строку как один фрагмент. Вместо неё мы можем использовать функции `plainto_tsquery` и `to_tsvector`, которые разбивают входную строку на несколько фрагментов:

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

В принципе так же можно использовать `to_tsquery`, если заключить аргумент в кавычки:

```
SELECT to_tsquery(' ''supernova star''');
 to_tsquery
------------
 'sn'
```

Заметьте, что `supernova star` совпадает с `supernovae stars` в `thesaurus_astro`, так как мы подключили стеммер `english_stem` в определении тезауруса. Этот стеммер удалил конечные буквы `e` и `s`.

Чтобы проиндексировать исходную фразу вместе с заменой, её нужно просто добавить в правую часть соответствующего определения:

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Словарь Ispell

Шаблон словарей Ispell поддерживает *морфологические словари*, которые могут сводить множество разных лингвистических форм слова к одной лексеме. Например, английский словарь Ispell может связать вместе все склонения и спряжения ключевого слова `bank`: `banking`, `banked`, `banks`, `banks'`,`bank's` и т. п.

Стандартный дистрибутив Postgres Pro не включает файлы конфигурации Ispell. Загрузить словари для множества языков можно со страницы *Ispell*. Кроме того, поддерживаются и другие современные форматы словарей: *MySpell* (OO < 2.0.1) и *Hunspell* (OO >= 2.0.2). Большой набор соответствующих словарей можно найти на странице *OpenOffice Wiki*.

Чтобы создать словарь Ispell, выполните следующие действия:

- загрузите файлы конфигурации словаря. Пакет с дополнительным словарём OpenOffice имеет расширение `.oxt`. Из него необходимо извлечь файлы `.aff` и `.dic`, и сменить их расширения на `.affix` и `.dict`, соответственно. Для некоторых файлов словарей также необходимо преобразовать символы в кодировку UTF-8 с помощью, например, таких команд (для норвежского языка):

  ```
  iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
  iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
  ```
- скопируйте файлы в каталог `$SHAREDIR/tsearch_data`
- загрузите эти файлы в Postgres Pro следующей командой:

  ```
  CREATE TEXT SEARCH DICTIONARY english_hunspell (
      TEMPLATE = ispell,
      DictFile = en_us,
      AffFile = en_us,
      Stopwords = english);
  ```

Здесь параметры `DictFile`, `AffFile` и `StopWords` определяют базовые имена файлов словаря, аффиксов и стоп-слов. Файл стоп-слов должен иметь тот же формат, что рассматривался выше в описании словаря `simple`. Формат других файлов здесь не рассматривается, но его можно узнать по вышеуказанным веб-адресам.

Словари Ispell обычно воспринимают ограниченный набор слов, так что за ними следует подключить другой, более общий словарь, например, Snowball, который принимает всё.

Файл `.affix` для Ispell имеет такую структуру:

```
prefixes
flag *A:
```

```
    .           >   RE      # As in enter > reenter
suffixes
flag T:
    E           >   ST      # As in late > latest
    [^AEIOU]Y   >   -Y,IEST # As in dirty > dirtiest
    [AEIOU]Y    >   EST     # As in gray > grayest
    [^EY]       >   EST     # As in small > smallest
```

А файл `.dict` — такую:

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

Формат файла `.dict` следующий:

```
basic_form/affix_class_name
```

В файле `.affix` каждый флаг аффиксов описывается в следующем формате:

```
условие > [-отсекаемые_буквы,] добавляемый_аффикс
```

Здесь условие записывается в формате, подобном формату регулярных выражений. В нём возможно описать группы `[...]` и `[^...]`. Например, запись `[AEIOU]Y` означает, что последняя буква слова — "`y`", а предпоследней может быть "`a`", "`e`", "`i`", "`o`" или "`u`". Запись `[^EY]` означает, что последняя буква не "`e`" и не "`y`".

Словари Ispell поддерживают разделение составных слов, что бывает полезно. Заметьте, что для этого в файле аффиксов нужно пометить специальным оператором `compoundwords controlled` слова, которые могут участвовать в составных образованиях:

```
compoundwords  controlled z
```

Вот как это работает для норвежского языка:

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

Формат MySpell представляет собой подмножество формата Hunspell. Файл `.affix` словаря Hunspell имеет следующую структуру:

```
PFX A Y 1
PFX A   0     re         .
SFX T N 4
SFX T   0     st         e
SFX T   y     iest       [^aeiou]y
SFX T   0     est        [aeiou]y
SFX T   0     est        [^ey]
```

Первая строка класса аффиксов — заголовок. Поля правил аффиксов указываются после заголовка:

- имя параметра (PFX или SFX)
- флаг (имя класса аффиксов)
- отсекаемые символы в начале (в префиксе) или в конце (в суффиксе) слова
- добавляемый аффикс
- условие в формате, подобном регулярным выражениям.

Файл `.dict` подобен файлу `.dict` словаря Ispell:

```
larder/M
lardy/RT
large/RSPMYT
largehearted
```

> **Примечание**
>
> Словарь MySpell не поддерживает составные слова. С другой стороны, Hunspell поддерживает множество операции с ними, но в настоящее время Postgres Pro использует только самые простые из этого множества.

12.6.6. Словарь Snowball

Шаблон словарей Snowball основан на проекте Мартина Потера, изобретателя популярного алгоритма стемминга для английского языка. Сейчас Snowball предлагает алгоритмы и для многих других языков (за подробностями обратитесь на *сайт Snowball*). Каждый алгоритм знает, как для данного языка свести распространённые словоформы к начальной форме. Для словаря Snowball задаётся обязательный параметр `language`, определяющий, какой именно стеммер использовать, и может задаваться параметр `stopword`, указывающий файл со списком исключаемых слов. (Стандартные списки стоп-слов Postgres Pro используется также в и проекте Snowball.) Например, встроенное определение выглядит так

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

Формат файла стоп-слов не отличается от рассмотренного ранее.

Словарь Snowball распознаёт любые фрагменты, даже если он не может упростить слова, так что он должен быть самым последним в списке словарей. Помещать его перед другими словарями нет смысла, так как после него никакой фрагмент не будет передан следующему словарю.

12.7. Пример конфигурации

Конфигурация текстового поиска определяет всё, что необходимо для преобразования документа в формат `tsvector`: анализатор, который будет разбивать текст на фрагменты, и словари, которые будут преобразовывать фрагменты в лексемы. При каждом вызове `to_tsvector` или `to_tsquery` обязательно используется конфигурация текстового поиска. В конфигурации сервера есть параметр default_text_search_config, задающий имя конфигурации текстового поиска по умолчанию, которая будет использоваться, когда при вызове функций поиска соответствующий аргумент не определён. Этот параметр можно задать в `postgresql.conf` или установить в рамках отдельного сеанса с помощью команды `SET`.

В системе есть несколько встроенных конфигураций текстового поиска и вы можете легко дополнить их своими. Для удобства управления объектами текстового поиска в PostgreSQL реализованы соответствующие SQL-команды и специальные команды в psql, выводящие информацию об этих объектах (Раздел 12.10).

В качестве примера использования этих команд мы создадим конфигурацию `pg`, взяв за основу встроенную конфигурацию `english`:

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

Мы будем использовать список синонимов, связанных с Postgres Pro, в файле `$SHAREDIR/tsearch_data/pg_dict.syn`. Этот файл содержит строки:

```
postgres    pg
```

```
pgsql       pg
postgresql  pg
```

Мы определим словарь синонимов следующим образом:

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

Затем мы зарегистрируем словарь Ispell `english_ispell`, у которого есть собственные файлы конфигурации:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Теперь мы можем настроить сопоставления для слов в конфигурации `pg`:

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

Мы решили не индексировать и не учитывать при поиске некоторые типы фрагментов, которые не обрабатываются встроенной конфигурацией:

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Теперь мы можем протестировать нашу конфигурацию:

```
SELECT * FROM ts_debug('public.pg', '
Postgres Pro, the highly scalable, SQL compliant, open source
object-relational database management system, is now undergoing
beta testing of the next version of our software.
');
```

И наконец мы выбираем в текущем сеансе эту конфигурацию, созданную в схеме `public`:

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. Тестирование и отладка текстового поиска

Поведение нестандартной конфигурации текстового поиска по мере её усложнения может стать непонятным. В этом разделе описаны функции, полезные для тестирования объектов текстового поиска. Вы можете тестировать конфигурацию как целиком, так и по частям, отлаживая анализаторы и словари по отдельности.

12.8.1. Тестирование конфигурации

Созданную конфигурацию текстового поиска можно легко протестировать с помощью функции `ts_debug`.

```
ts_debug([конфигурация regconfig,] документ text,
         OUT псевдоним text,
         OUT описание text,
         OUT фрагмент text,
         OUT словари regdictionary[],
         OUT словарь regdictionary,
         OUT лексемы text[])
         returns setof record
```

`ts_debug` выводит информацию обо всех фрагментах данного документа, которые были выданы анализатором и обработаны настроенными словарями. Она использует конфигурацию, указанную в аргументе *`config`*, или `default_text_search_config`, если этот аргумент опущен.

`ts_debug` возвращает по одной строке для каждого фрагмента, найденного в тексте анализатором. Эта строка содержит следующие столбцы:

- *`синоним`* `text` — краткое имя типа фрагмента
- *`описание`* `text` — описание типа фрагмента
- *`фрагмент`* `text` — текст фрагмента
- *`словари`* `regdictionary[]` — словари, назначенные в конфигурации для фрагментов такого типа
- *`словарь`* `regdictionary` — словарь, распознавший этот фрагмент, или `NULL`, если подходящего словаря не нашлось
- *`лексемы`* `text[]` — лексемы, выданные словарём, распознавшим фрагмент, или `NULL`, если подходящий словарь не нашёлся; может быть также пустым массивом (`{}`), если фрагмент распознан как стоп-слово

Простой пример:

```
SELECT * FROM ts_debug('english', 'a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
```

```
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

Для более полной демонстрации мы сначала создадим конфигурацию `public.english` и словарь Ispell для английского языка:

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;
```

```
SELECT * FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias    |   description   |    token    |         dictionaries          |
 dictionary   |   lexemes
-----------+-----------------+-------------+-------------------------------
+----------------+-------------
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} |
 english_ispell | {}
 blank     | Space symbols   |             | {}                            |
     |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} |
 english_ispell | {bright}
 blank     | Space symbols   |             | {}                            |
     |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} |
 english_stem   | {supernova}
```

В этом примере слово `Brightest` было воспринято анализатором как фрагмент `ASCII word` (синоним `asciiword`). Для этого типа фрагментов список словарей включает `english_ispell` и `english_stem`. Данное слово было распознано словарём `english_ispell`, который свёл его к `bright`. Слово `supernovaes` оказалось незнакомо словарю `english_ispell`, так что оно было передано следующему словарю, который его благополучно распознал (на самом деле `english_stem` — это стеммер Snowball, который распознаёт всё, поэтому он включён в список словарей последним).

Слово `The` было распознано словарём `english_ispell` как стоп-слово (см. Подраздел 12.6.1) и поэтому не будет индексироваться. Пробелы тоже отбрасываются, так как в данной конфигурации для них нет словарей.

Вы можете уменьшить ширину вывода, явно перечислив только те столбцы, которые вы хотите видеть:

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. Тестирование анализатора

Следующие функции позволяют непосредственно протестировать анализатор текстового поиска.

```
ts_parse(имя_анализатора text, документ text,
         OUT код_фрагмента integer, OUT фрагмент text) returns setof record
ts_parse(oid_анализатора oid, документ text,
         OUT код_фрагмента integer, OUT фрагмент text) returns setof record
```

`ts_parse` разбирает данный документ и возвращает набор записей, по одной для каждого извлечённого фрагмента. Каждая запись содержит `код_фрагмента`, код назначенного типа фрагмента, и `фрагмент`, собственно текст фрагмента. Например:

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(имя_анализатора text, OUT код_фрагмента integer,
              OUT псевдоним text, OUT описание text) returns setof record
ts_token_type(oid_анализатора oid, OUT код_фрагмента integer,
              OUT псевдоним text, OUT описание text) returns setof record
```

`ts_token_type` возвращает таблицу, описывающую все типы фрагментов, которые может распознать анализатор. Для каждого типа в этой таблице указывается целочисленный `tokid` (идентификатор), который анализатор использует для пометки фрагмента этого типа, `alias` (псевдоним), с которым этот тип фигурирует в командах конфигурации, и `description` (краткое описание). Например:

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
```

12.8.3. Тестирование словаря

Для тестирования словаря предназначена функция `ts_lexize`.

`ts_lexize(`*`словарь`*` regdictionary, `*`фрагмент`*` text) returns text[]`

`ts_lexize` возвращает массив лексем, если входной *`фрагмент`* известен словарю, либо пустой массив, если этот фрагмент считается в словаре стоп-словом, либо `NULL`, если он не был распознан.

Примеры:

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

> **Примечание**
>
> Функция `ts_lexize` принимает одиночный *фрагмент*, а не просто текст. Вот пример возможного заблуждения:
>
> ```
> SELECT ts_lexize('thesaurus_astro', 'supernovae stars') is null;
>  ?column?
> ----------
>  t
> ```
>
> Хотя фраза `supernovae stars` есть в тезаурусе `thesaurus_astro`, `ts_lexize` не работает, так как она не разбирает входной текст, а воспринимает его как один фрагмент. Поэтому для проверки тезаурусов следует использовать функции `plainto_tsquery` и `to_tsvector`, например:
>
> ```
> SELECT plainto_tsquery('supernovae stars');
>  plainto_tsquery
> -----------------
>  'sn'
> ```

12.9. Типы индексов, предпочитаемые для текстового поиска

Для ускорения полнотекстового поиска можно использовать индексы GIN, RUM и GiST. Заметьте, что эти индексы не требуются для поиска, но если по какому-то столбцу поиск выполняется регулярно, обычно желательно создать индекс.

Чтобы создать такой индекс, выполните одну из следующих команд:

`CREATE INDEX `*`имя`*` ON `*`таблица`*` USING GIN (`*`столбец`*`);`

Создаёт индекс на базе GIN (Generalized Inverted Index, Обобщённый Инвертированный Индекс). *`Столбец`* должен иметь тип `tsvector`.

`CREATE INDEX `*`имя`*` ON `*`таблица`*` USING RUM (`*`столбец`*`);`

Создаёт индекс на базе RUM. *`Столбец`* должен иметь тип `tsvector`.

`CREATE INDEX `*`имя`*` ON `*`таблица`*` USING GIST (`*`столбец`*` [ { DEFAULT | tsvector_ops } (siglen = `*`число`*`) ] );`

Создаёт индекс на базе GiST (Generalized Search Tree, Обобщённое дерево поиска). Здесь *`столбец`* может иметь тип `tsvector` или `tsquery`. Необязательный числовой параметр `siglen` определяет длину сигнатуры в байтах (подробнее об этом ниже).

Более предпочтительными для текстового поиска являются индексы GIN и основанные на методе доступа `GIN` индексы RUM. Будучи инвертированными индексами, они содержат записи для всех отдельных слов (лексем) с компактным списком мест их вхождений. При поиске нескольких слов можно найти первое, а затем воспользоваться индексом и исключить строки, в которых дополнительные слова отсутствуют. Индексы GIN хранят только слова (лексемы) из значений `tsvector` и теряют информацию об их весах. Таким образом для выполнения запроса с весами потребуется перепроверить строки в таблице.

Индекс GiST допускает *неточности*, то есть он допускает ложные попадания и поэтому их нужно исключать дополнительно, сверяя результат с фактическими данными таблицы. (Postgres Pro делает это автоматически.) Индексы GiST являются неточными, так как все документы в них представляются сигнатурой фиксированной длины. Длина сигнатуры в байтах определяется значением необязательного целочисленного параметра `siglen`. По умолчанию (когда параметр `siglen` отсутствует) равно 124 байтам, а максимальная длина сигнатуры равна 2024 байтам. Сигнатура формируется в результате представления присутствия каждого слова как одного бита в строке из n-бит, а затем логического объединения этих битовых строк. Если двум словам будет соответствовать одна битовая позиция, попадание оказывается ложным. Если для всех слов оказались установлены соответствующие биты (в случае фактического или ложного попадания), для проверки правильности предположения о совпадении слов необходимо прочитать строку таблицы. При увеличении размера сигнатуры поиск работает точнее (сканируется меньшая область в индексе и меньше страниц кучи), но сам индекс становится больше.

Индекс GiST может быть покрывающим, то есть использовать функциональность `INCLUDE`. В качестве дополнительных в него могут включаться столбцы, для типа данных которых не определён класс операторов GiST. Атрибуты, включаемые в индекс, сохраняются в нём без сжатия.

Неточность индекса приводит к снижению производительности из-за дополнительных обращений к записям таблицы, для которых предположение о совпадении оказывается ложным. Так как произвольный доступ к таблице обычно не бывает быстрым, это ограничивает применимость индексов GiST. Вероятность ложных попаданий зависит от ряда факторов, например от количества уникальных слов, так что его рекомендуется сокращать, применяя словари.

Заметьте, что построение индекса GIN часто можно ускорить, увеличив maintenance_work_mem, тогда как время построения индекса GiST не зависит от этого параметра.

Правильно используя индексы GIN, RUM и GiST и разделяя большие коллекции документов на секции, можно реализовать очень быстрый поиск с возможностью обновления «на лету». Секционировать данные можно как на уровне базы, с использованием наследования таблиц, так и распределив документы по разным серверам и затем собирая результаты внешнего поиска, например, средствами доступа к стороннним данным. Последний вариант возможен благодаря тому, что функции ранжирования используют только локальную информацию.

12.10. Поддержка psql

Информацию об объектах конфигурации текстового поиска можно получить в psql с помощью следующего набора команд:

```
\dF{d,p,t}[+] [ШАБЛОН]
```

Необязательный `+` в этих командах включает более подробный вывод.

В необязательном параметре *`ШАБЛОН`* может указываться имя объекта текстового поиска (возможно, дополненное схемой). Если *`ШАБЛОН`* не указан, выводится информация обо всех видимых объектах. *`ШАБЛОН`* может содержать регулярное выражение с *разными* масками для схемы и объекта. Это иллюстрируют следующие примеры:

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
```

```
 public | fulltext_cfg |
```

```
=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+----------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

Возможны следующие команды:

`\dF[+] [ШАБЛОН]`

Список конфигураций текстового поиска (добавьте + для дополнительных сведений).

```
=> \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+------------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

`\dFd[+] [ШАБЛОН]`

Список словарей текстового поиска (добавьте + для дополнительных сведений).

```
=> \dFd
                             List of text search dictionaries
   Schema   |      Name       |                        Description
------------+-----------------
+-----------------------------------------------------------
 pg_catalog | arabic_stem     | snowball stemmer for arabic language
 pg_catalog | armenian_stem   | snowball stemmer for armenian language
 pg_catalog | basque_stem     | snowball stemmer for basque language
 pg_catalog | catalan_stem    | snowball stemmer for catalan language
 pg_catalog | danish_stem     | snowball stemmer for danish language
 pg_catalog | dutch_stem      | snowball stemmer for dutch language
```

```
 pg_catalog | english_stem    | snowball stemmer for english language
 pg_catalog | finnish_stem    | snowball stemmer for finnish language
 pg_catalog | french_stem     | snowball stemmer for french language
 pg_catalog | german_stem     | snowball stemmer for german language
 pg_catalog | greek_stem      | snowball stemmer for greek language
 pg_catalog | hindi_stem      | snowball stemmer for hindi language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian language
 pg_catalog | indonesian_stem | snowball stemmer for indonesian language
 pg_catalog | irish_stem      | snowball stemmer for irish language
 pg_catalog | italian_stem    | snowball stemmer for italian language
 pg_catalog | lithuanian_stem | snowball stemmer for lithuanian language
 pg_catalog | nepali_stem     | snowball stemmer for nepali language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese language
 pg_catalog | romanian_stem   | snowball stemmer for romanian language
 pg_catalog | russian_stem    | snowball stemmer for russian language
 pg_catalog | serbian_stem    | snowball stemmer for serbian language
 pg_catalog | simple          | simple dictionary: just lower case and check for
 stopword
 pg_catalog | spanish_stem    | snowball stemmer for spanish language
 pg_catalog | swedish_stem    | snowball stemmer for swedish language
 pg_catalog | tamil_stem      | snowball stemmer for tamil language
 pg_catalog | turkish_stem    | snowball stemmer for turkish language
 pg_catalog | yiddish_stem    | snowball stemmer for yiddish language
```

`\dFp[+] [ШАБЛОН]`

Список анализаторов текстового поиска (добавьте + для дополнительных сведений).

```
=> \dFp
       List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

       Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+------------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
```

```
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

`\dFt[+] [ШАБЛОН]`

Список шаблонов текстового поиска (добавьте `+` для дополнительных сведений).

```
=> \dFt
                           List of text search templates
   Schema   |   Name    |                        Description
------------+-----------+-----------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and check for stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

12.11. Ограничения

Текущая реализация текстового поиска в Postgres Pro имеет следующие ограничения:

- Длина лексемы не может превышать 2 килобайта
- Длина значения `tsvector` (лексемы и их позиции) не может превышать 1 мегабайт
- Число лексем должно быть меньше 2^{64}
- Значения позиций в `tsvector` должны быть от 0 до 16383
- Расстояние в операторе `<`*N*`>` (ПРЕДШЕСТВУЕТ) типа `tsquery` не может быть больше 16384
- Не больше 256 позиций для одной лексемы
- Число узлов (лексемы + операторы) в значении `tsquery` должно быть меньше 32768

Для сравнения, документация PostgreSQL 8.1 содержала 335 420 слов, из них 10 441 уникальных, а наиболее часто употребляющееся в ней слово «postgresql» встречается 6 127 раз в 655 документах.

Другой пример — архивы списков рассылки PostgreSQL содержали 910 989 уникальных слов в 57 491 343 лексемах в 461 020 сообщениях.

Глава 13. Управление конкурентным доступом

В этой главе описывается поведение СУБД Postgres Pro в ситуациях, когда два или более сеансов пытаются одновременно обратиться к одним и тем же данным. В таких ситуациях важно, чтобы все сеансы могли эффективно работать с данными, и при этом сохранялась целостность данных. Обсуждаемые в этой главе темы заслуживают внимания всех разработчиков баз данных.

13.1. Введение

Postgres Pro предоставляет разработчикам богатый набор средств для управления конкурентным доступом к данным. Внутри он поддерживает целостность данных, реализуя модель MVCC (Multiversion Concurrency Control, Многоверсионное управление конкурентным доступом). Это означает, что каждый SQL-оператор видит снимок данных (*версию базы данных*) на определённый момент времени, вне зависимости от текущего состояния данных. Это защищает операторы от несогласованности данных, возможной, если другие конкурирующие транзакции внесут изменения в те же строки данных, и обеспечивает тем самым *изоляцию транзакций* для каждого сеанса баз данных. MVCC, отходя от методик блокирования, принятых в традиционных СУБД, снижает уровень конфликтов блокировок и таким образом обеспечивает более высокую производительность в многопользовательской среде.

Основное преимущество использования модели MVCC по сравнению с блокированием заключается в том, что блокировки MVCC, полученные для чтения данных, не конфликтуют с блокировками, полученными для записи, и поэтому чтение никогда не мешает записи, а запись чтению. Postgres Pro гарантирует это даже для самого строгого уровня изоляции транзакций, используя инновационный уровень изоляции SSI (*Serializable Snapshot Isolation*, Сериализуемая изоляция снимков).

Для приложений, которым в принципе не нужна полная изоляция транзакций и которые предпочитают явно определять точки конфликтов, в Postgres Pro также есть средства блокировки на уровне таблиц и строк. Однако при правильном использовании MVCC обычно обеспечивает лучшую производительность, чем блокировки. Кроме этого, приложения могут использовать рекомендательные блокировки, не привязанные к какой-либо одной транзакции.

13.2. Изоляция транзакций

Стандарт SQL определяет четыре уровня изоляции транзакций. Наиболее строгий из них — сериализуемый, определяется одним абзацем, говорящем, что при параллельном выполнении нескольких сериализуемых транзакций должны гарантированно выдавать такой же результат, как если бы они запускались по очереди в некотором порядке. Остальные три уровня определяются через описания особых явлений, которые возможны при взаимодействии параллельных транзакций, но не допускаются на определённом уровне. Как отмечается в стандарте, из определения сериализуемого уровня вытекает, что на этом уровне ни одно из этих явлений не возможно. (В самом деле — если эффект транзакций должен быть тем же, что и при их выполнении по очереди, как можно было бы увидеть особые явления, связанные с другими транзакциями?)

Стандарт описывает следующие особые условия, недопустимые для различных уровней изоляции:

«грязное» чтение

Транзакция читает данные, записанные параллельной незавершённой транзакцией.

неповторяемое чтение

Транзакция повторно читает те же данные, что и раньше, и обнаруживает, что они были изменены другой транзакцией (которая завершилась после первого чтения).

фантомное чтение

Транзакция повторно выполняет запрос, возвращающий набор строк для некоторого условия, и обнаруживает, что набор строк, удовлетворяющих условию, изменился из-за транзакции, завершившейся за это время.

аномалия сериализации

Результат успешной фиксации группы транзакций оказывается несогласованным при всевозможных вариантах исполнения этих транзакций по очереди.

Уровни изоляции транзакций, описанные в стандарте SQL и реализованные в Postgres Pro, описываются в Таблице 13.1.

Таблица 13.1. Уровни изоляции транзакций

Уровень изоляции	«Грязное» чтение	Неповторяемое чтение	Фантомное чтение	Аномалия сериализации
Read uncommitted (Чтение незафиксированных данных)	Допускается, но не в PG	Возможно	Возможно	Возможно
Read committed (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возможно
Repeatable read (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PG	Возможно
Serializable (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозможно

В Postgres Pro вы можете запросить любой из четырёх уровней изоляции транзакций, однако внутри реализованы только три различных уровня, то есть режим Read Uncommitted в Postgres Pro действует как Read Committed. Причина этого в том, что только так можно сопоставить стандартные уровни изоляции с реализованной в Postgres Pro архитектурой многоверсионного управления конкурентным доступом.

В этой таблице также показано, что реализация Repeatable Read в Postgres Pro не допускает фантомного чтения. Это соответствует стандарту SQL, поскольку он устанавливает, каких аномалий *не* должно быть на определённых уровнях изоляции, но допускает и более строгие ограничения. Поведение имеющихся уровней изоляции подробно описывается в следующих подразделах.

Для выбора нужного уровня изоляции транзакций используется команда SET TRANSACTION.

Важно

Поведение некоторых функций и типов данных Postgres Pro в транзакциях подчиняется особым правилам. В частности, изменения последовательностей (и следовательно, счётчика в столбце, объявленному как `serial`) немедленно видны во всех остальных транзакциях и не откатываются назад, если выполнившая их транзакция прерывается. См. Раздел 9.17 и Подраздел 8.1.4.

13.2.1. Уровень изоляции Read Committed

Read Committed — уровень изоляции транзакции, выбираемый в Postgres Pro по умолчанию. В транзакции, работающей на этом уровне, запрос `SELECT` (без предложения `FOR UPDATE/SHARE`) видит только те данные, которые были зафиксированы до начала запроса; он никогда не увидит незафиксированных данных или изменений, внесённых параллельными транзакциями в процессе выполнения запроса. По сути запрос `SELECT` видит снимок базы данных в момент начала выполнения запроса. Однако `SELECT` видит результаты изменений, внесённых ранее в этой же транзакции, даже если они ещё не зафиксированы. Также заметьте, что два последовательных оператора `SELECT` могут видеть разные данные даже в рамках одной транзакции, если какие-то другие транзакции зафиксируют изменения после запуска первого `SELECT`, но до запуска второго.

Команды `UPDATE`, `DELETE`, `SELECT FOR UPDATE` и `SELECT FOR SHARE` ведут себя подобно `SELECT` при поиске целевых строк: они найдут только те целевые строки, которые были зафиксированы на момент начала команды. Однако к моменту, когда они будут найдены, эти целевые строки могут быть уже изменены (а также удалены или заблокированы) другой параллельной транзакцией. В этом случае запланированное изменение будет отложено до фиксирования или отката первой изменяющей данные транзакции (если она ещё выполняется). Если первая изменяющая транзакция откатывается, её результат отбрасывается и вторая изменяющая транзакция может продолжить изменение изначально полученной строки. Если первая транзакция зафиксировалась, но в результате удалила эту строку, вторая будет игнорировать её, а в противном случае попытается выполнить свою операцию с изменённой версией строки. Условие поиска в команде (предложение `WHERE`) вычисляется повторно для выяснения, соответствует ли по-прежнему этому условию изменённая версия строки. Если да, вторая изменяющая транзакция продолжают свою работу с изменённой версией строки. Применительно к командам `SELECT FOR UPDATE` и `SELECT FOR SHARE` это означает, что изменённая версия строки блокируется и возвращается клиенту.

Похожим образом ведёт себя `INSERT` с предложением `ON CONFLICT DO UPDATE`. В режиме Read Committed каждая строка, предлагаемая для добавления, будет либо вставлена, либо изменена. Если не возникнет несвязанных ошибок, гарантируется один из этих двух исходов. Если конфликт будет вызван другой транзакцией, результат которой ещё не видим для `INSERT`, предложение `UPDATE` подействует на эту строку, даже несмотря на то, что эта команда обычным образом может не видеть *никакую* версию этой строки.

При выполнении `INSERT` с предложением `ON CONFLICT DO NOTHING` строка может не добавиться в результате действия другой транзакции, эффект которой не виден в снимке команды `INSERT`. Это опять же имеет место только в режиме Read Committed.

`MERGE` позволяет пользователю указать различные комбинации подкоманд `INSERT`, `UPDATE` и `DELETE`. Команда `MERGE` с подкомандами `INSERT` и `UPDATE` работает аналогично `INSERT` с `ON CONFLICT DO UPDATE`, но не гарантирует, что какая-либо из команд `INSERT` или `UPDATE` будет выполнена. Если `MERGE` пытается выполнить `UPDATE` или `DELETE`, и строка одновременно изменяется, но условие соединения по-прежнему выполняется для текущего целевого и текущего исходного кортежа, то `MERGE` будет работать так же, как команды `UPDATE` или `DELETE`, и выполнит свои действия с изменённой версией строки. Однако поскольку в `MERGE` могут указываться несколько действий, и они могут быть условными, условия для всех действий заново вычисляются для изменённой версии строки, начиная с первого действия, даже если действие, которое первоначально оказалось подходящим, идёт в списке действий позже. С другой стороны, если строка одновременно изменяется так, что условие соединения не выполняется, `MERGE` будет вычислять следующие условия действий `NOT MATCHED BY SOURCE` и `NOT MATCHED [BY TARGET]` и выполнит действие каждого типа с первым подходящим условием. Если строка одновременно удаляется, `MERGE` будет вычислять следующие условия действий `NOT MATCHED [BY TARGET]` и выполнит действие с первым подходящим условием. Если `MERGE` пытается выполнить `INSERT` при наличии уникального индекса, и одновременно добавляется повторяющаяся строка, возникает ошибка нарушения уникальности; `MERGE` не пытается избежать таких ошибок, заново вычисляя условия `MATCHED`.

Вследствие описанных выше правил, изменяющая команда может увидеть несогласованное состояние: она может видеть результаты параллельных команд, изменяющих те же строки, что пытается изменить она, но при этом она не видит результаты этих команд в других строках таблиц. Из-за этого поведения уровень Read Committed не подходит для команд со сложными условиями поиска; однако он вполне пригоден для простых случаев. Например, рассмотрим изменение баланса счёта в таких транзакциях:

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

Если две такие транзакции пытаются параллельно изменить баланс счёта 12345, мы, естественно, хотим, чтобы вторая транзакция работала с изменённой версией строки счёта. Так как каждая команда влияет только на определённую строку, если она будет видеть изменённую версию строки, это не приведёт к проблемам несогласованности.

В более сложных ситуациях уровень Read Committed может приводить к нежелательным результатам. Например, рассмотрим команду `DELETE`, работающую со строками, которые параллельно добавляет и удаляет из множества, определённого её условием, другая команда. Например, предположим, что `website` — таблица из двух строк, в которых `website.hits` равны `9` и `10`:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- выполняется параллельно:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

Команда `DELETE` не сделает ничего, даже несмотря на то, что строка с `website.hits = 10` была в таблице и до, и после выполнения `UPDATE`. Это происходит потому, что строка со значением `9` до изменения пропускается, а когда команда `UPDATE` завершается и `DELETE` получает освободившуюся блокировку, строка с `10` теперь содержит `11`, а это значение уже не соответствует условию.

Так как в режиме Read Committed каждая команда начинается с нового снимка состояния, который включает результаты всех транзакций, зафиксированных к этому моменту, последующие команды в одной транзакции будут в любом случае видеть эффекты всех параллельных зафиксированных транзакций. Вопрос здесь состоит в том, видит ли *одна* команда абсолютно согласованное состояние базы данных.

Частичная изоляция транзакций, обеспечиваемая в режиме Read Committed, приемлема для множества приложений. Этот режим быстр и прост в использовании, однако он подходит не для всех случаев. Приложениям, выполняющим сложные запросы и изменения, могут потребоваться более строго согласованное представление данных, чем то, что даёт Read Committed.

13.2.2. Уровень изоляции Repeatable Read

В режиме *Repeatable Read* видны только те данные, которые были зафиксированы до начала транзакции, но не видны незафиксированные данные и изменения, произведённые другими транзакциями в процессе выполнения данной транзакции. (Однако каждый запрос будет видеть эффекты предыдущих изменений в своей транзакции, несмотря на то, что они не зафиксированы.) Это самое строгое требование, которое стандарт SQL вводит для этого уровня изоляции, и при его выполнении предотвращаются все явления, описанные в Таблице 13.1, за исключением аномалий сериализации. Как было сказано выше, это не противоречит стандарту, так как он определяет только *минимальную* защиту, которая должна обеспечиваться на каждом уровне изоляции.

Этот уровень отличается от Read Committed тем, что запрос в транзакции данного уровня видит снимок данных на момент начала первого оператора в *транзакции* (не считая команд управления транзакциями), а не начала текущего оператора. Таким образом, последовательные команды `SELECT` в *одной* транзакции видят одни и те же данные; они не видят изменений, внесённых и зафиксированных другими транзакциями после начала их текущей транзакции.

Приложения, использующие этот уровень, должны быть готовы повторить транзакции в случае сбоев сериализации.

Команды `UPDATE`, `DELETE`, `MERGE`, `SELECT FOR UPDATE` и `SELECT FOR SHARE` ведут себя подобно `SELECT` при поиске целевых строк: они найдут только те целевые строки, которые были зафиксированы на момент начала транзакции. Однако к моменту, когда они будут найдены, эти целевые строки могут быть уже изменены (а также удалены или заблокированы) другой параллельной транзакцией. В этом случае транзакция в режиме Repeatable Read будет ожидать фиксирования или отката первой изменяющей данные транзакции (если она ещё выполняется). Если первая изменяющая транзакция откатывается, её результат отбрасывается и текущая транзакция может продолжить изменение изначально полученной строки. Если же первая транзакция зафиксировалась и в результате изменила или удалила эту строку, а не просто заблокировала её, произойдёт откат текущей транзакции с сообщением

```
ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения
```

так как транзакция уровня Repeatable Read не может изменять или блокировать строки, изменённые другими транзакциями с момента её начала.

Когда приложение получает это сообщение об ошибке, оно должна прервать текущую транзакцию и попытаться повторить её с самого начала. Во второй раз транзакция увидит внесённое до этого изменение как часть начального снимка базы данных, так что новая версия строки вполне может использоваться в качестве отправной точки для изменения в повторной транзакции.

Заметьте, что потребность в повторении транзакции может возникнуть, только если эта транзакция изменяет данные; в транзакциях, которые только читают данные, конфликтов сериализации не бывает.

Режим Repeatable Read строго гарантирует, что каждая транзакция видит полностью стабильное представление базы данных. Однако это представление не обязательно будет согласовано с некоторым последовательным выполнением транзакций одного уровня. Например, даже транзакция, которая только читает данные, в этом режиме может видеть строку, показывающую, что некоторое задание завершено, но *не* видеть одну из строк логических частей задания, так как эта транзакция может прочитать более раннюю версию строки задания, чем ту, для которой параллельно добавлялась очередная логическая часть. Строго исполнить бизнес-правила в транзакциях, работающих на этом уровне изоляции, скорее всего не удастся без явных блокировок конфликтующих транзакций.

Для реализации уровня изоляции Repeatable Read применяется подход, который называется в академической литературе по базам данных и в других СУБД *Изоляция снимков* (Snapshot Isolation). По сравнению с системами, использующими традиционный метод блокировок, затрудняющий параллельное выполнение, при этом подходе наблюдается другое поведение и другая производительность. В некоторых СУБД могут существовать даже два отдельных уровня Repeatable Read и Snapshot Isolation с различным поведением. Допускаемые особые условия, представляющие отличия двух этих подходов, не были формализованы разработчиками теории БД до развития стандарта SQL и их рассмотрение выходит за рамки данного руководства. В полном объёме эта тема освещается в berenson95.

Примечание

До версии 9.1 в PostgreSQL при запросе режима Serializable поведение системы в точности соответствовало вышеописанному. Таким образом, чтобы сейчас получить старое поведение Serializable, нужно запрашивать режим Repeatable Read.

13.2.3. Уровень изоляции Serializable

Уровень *Serializable* обеспечивает самую строгую изоляцию транзакций. На этом уровне моделируется последовательное выполнение всех зафиксированных транзакций, как если бы транзакции выполнялись одна за другой, последовательно, а не параллельно. Однако, как и на уровне Repeatable Read, на этом уровне приложения должны быть готовы повторять транзакции из-за сбоев сериализации. Фактически этот режим изоляции работает так же, как и Repeatable Read, только он дополнительно отслеживает условия, при которых результат параллельно выполняемых сериализуемых транзакций может не согласовываться с результатом этих же транзакций, выполняемых по очереди. Это отслеживание не привносит дополнительных препятствий для выполнения, кроме тех, что присущи режиму Repeatable Read, но тем не менее создаёт некоторую добавочную нагрузку, а при выявлении исключительных условий регистрируется *аномалия сериализации* и происходит *сбой сериализации*.

Например, рассмотрим таблицу `mytab`, изначально содержащую:

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

Предположим, что сериализуемая транзакция A вычисляет:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

а затем вставляет результат (30) в поле `value` в новую строку со значением `class = 2`. В это же время сериализуемая транзакция B вычисляет:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

получает результат 300 и вставляет его в новую строку со значением `class = 1`. Затем обе транзакции пытаются зафиксироваться. Если бы одна из этих транзакций работала в режиме Repeatable Read, зафиксироваться могли бы обе; но так как полученный результат не соответствовал бы последовательному порядку, в режиме Serializable будет зафиксирована только одна транзакция, а вторая закончится откатом с сообщением:

```
ОШИБКА: не удалось сериализовать доступ из-за зависимостей чтения/записи между
  транзакциями
```

Это объясняется тем, что при выполнении A перед B транзакция B вычислила бы сумму 330, а не 300, а при выполнении в обратном порядке A вычислила бы другую сумму.

Рассчитывая, что сериализуемые транзакции предотвратят аномалии, важно понимать, что любые данные, полученные из постоянной таблицы пользователя, не должны считаться действительными, пока транзакция, прочитавшая их, не будет успешно зафиксирована. Это верно даже для транзакций, не модифицирующих данные, за исключением случая, когда данные считываются в *откладываемой* транзакции такого типа. В этом случае данные могут считаться действительными, так как такая транзакция ждёт, пока не сможет получить снимок, гарантированно предотвращающий подобные проблемы. Во всех остальных случаях приложения не должны полагаться на результаты чтения данных в транзакции, которая не была зафиксирована; в случае ошибки и отката приложения должны повторять транзакцию, пока она не будет завершена успешно.

Для полной гарантии сериализуемости в Postgres Pro применяются *предикатные блокировки*, то есть блокировки, позволяющие определить, когда запись могла бы повлиять на результат предыдущего чтения параллельной транзакции, если бы эта запись выполнялась сначала. В Postgres Pro эти блокировки не приводят к фактическим блокировкам данных и, следовательно, никоим образом *не* могут повлечь взаимоблокировки транзакций. Они помогают выявить и отметить зависимости между параллельными транзакциями уровня Serializable, которые в определённых сочетаниях могут приводить к аномалиям сериализации. Транзакции Read Committed или Repeatable Read для обеспечения целостности данных, напротив, должны либо блокировать таблицы целиком, что помешает пользователям обращаться к этим таблицам, либо применять `SELECT FOR UPDATE` или `SELECT FOR SHARE`, что не только заблокирует другие транзакции, но и создаст дополнительную нагрузку на диск.

Предикатные блокировки в Postgres Pro, как и в большинстве других СУБД, устанавливаются для данных, фактически используемых в транзакции. Они отображаются в системном представлении `pg_locks` со значением `mode` равным `SIReadLock`. Какие именно блокировки будут затребованы при выполнении запроса, зависит от плана запроса, при этом детализированные блокировки (например, блокировки строк) могут объединяться в более общие (например, в блокировки страниц) в процессе транзакции для экономии памяти, расходуемой для отслеживания блокировок. Транзакция `READ ONLY` может даже освободить свои блокировки SIRead до завершения, если обнаруживается, что конфликты, которые могли бы привести к аномалии сериализации, исключены. На самом деле для транзакций `READ ONLY` этот факт чаще всего устанавливается в самом начале, так что они обходятся без предикатных блокировок. Если же вы явно запросите транзакцию `SERIALIZABLE READ ONLY DEFERRABLE`, она будет заблокирована до тех пор, пока не сможет установить этот факт. (Это *единственный* случай, когда транзакции уровня Serializable блокируются, а транзакции Repeatable Read — нет.) С другой стороны, блокировки SIRead часто должны сохраняться и после фиксирования транзакции, пока не будут завершены другие, наложившиеся на неё транзакции.

При правильном использовании сериализуемые транзакции могут значительно упростить разработку приложений. Гарантия того, что любое сочетание успешно зафиксированных параллельных сериализуемых транзакций даст тот же результат, что и последовательность этих транзакций, выполненных по очереди, означает, что если вы уверены, что единственная транзакция определённого содержания работает правильно, когда она запускается отдельно, вы можете быть уверены, что она будет работать так же правильно в любом сочетании сериализуемых транзакций, вне зависимости от того, что они делают, либо же она не будет зафиксирована успешно. При этом важно, чтобы в среде, где применяется этот подход, была реализована общая обработка сбоев сериализации (которые можно определить по значению SQLSTATE '40001'), так как заведомо определить, какие именно транзакции могут стать жертвами зависимостей чтения/записи и не будут зафиксированы для предотвращения аномалий сериализации, обычно очень сложно. Отслеживание зависимостей чтения-записи неизбежно создаёт дополнительную нагрузку, как и перезапуск транзакций, не зафиксированных из-за сбоев сериализации, но если на другую чашу весов положить нагрузку и блокирование, связанные с применением явных блокировок и `SELECT FOR UPDATE` или `SELECT FOR SHARE`, использовать сериализуемые транзакции в ряде случаев окажется выгоднее.

Тогда как уровень изоляции транзакций Serializable в Postgres Pro позволяет фиксировать параллельные транзакции, только если есть уверенность, что тот же результат будет получен при последовательном их выполнении, он не всегда предотвращает ошибки, которые не возникли бы при действительно последовательном выполнении. В частности, можно столкнуться с нарушениями ограничений уникальности, вызванными наложением сериализуемых транзакций, даже после явной проверки отсутствия ключа перед добавлением его. Этого можно избежать, если *все* сериализуемые транзакции, добавляющие потенциально конфликтующие ключи, будут предварительно явно проверять, можно ли вставить ключ. Например, приложение, добавляющее новый ключ, может запрашивать его у пользователя и затем проверять, существует ли он, сначала пытаясь найти его, либо генерировать новый ключ, выбирая максимальное существующее значение и увеличивая его на один. Если некоторые сериализуемые транзакции добавляют новые ключи сразу, не следуя этому протоколу, возможны нарушения ограничений уникальности, даже когда они не наблюдались бы при последовательном выполнении этих транзакций.

Применяя сериализуемые транзакции для управления конкурентным доступом, примите к сведению следующие рекомендации:

- Объявляйте транзакции как `READ ONLY`, если это отражает их суть.
- Управляйте числом активных подключений, при необходимости используя пул соединений. Это всегда полезно для увеличения производительности, но особенно важно это в загруженной системе с сериализуемыми транзакциями.
- Заключайте в одну транзакцию не больше команд, чем необходимо для обеспечения целостности.
- Не оставляйте соединения «простаивающими в транзакции» дольше, чем необходимо. Для автоматического отключения затянувшихся транзакций можно применить параметр конфигурации idle_in_transaction_session_timeout.
- Исключите явные блокировки, `SELECT FOR UPDATE` и `SELECT FOR SHARE` там, где они не нужны благодаря защите, автоматически предоставляемой сериализуемыми транзакциями.
- Когда система вынуждена объединять предикатные блокировки уровня страницы в одну предикатную блокировку уровня таблицы из-за нехватки памяти, может возрасти частота сбоев сериализации. Избежать этого можно, увеличив параметр max_pred_locks_per_transaction, max_pred_locks_per_relation и/или max_pred_locks_per_page.
- Последовательное сканирование всегда влечёт за собой предикатную блокировку на уровне таблицы. Это приводит к увеличению сбоев сериализации. В таких ситуациях бывает полезно склонить систему к использованию индексов, уменьшая random_page_cost и/или увеличивая cpu_tuple_cost. Однако тут важно сопоставить выигрыш от уменьшения числа откатов и перезапусков транзакций с проигрышем от возможного менее эффективного выполнения запросов.

Для реализации уровня изоляции Serializable применяется подход, который называется в академической литературе по базам данных *Изоляция снимков* (Snapshot Isolation), с дополнительными проверками на предмет аномалий сериализации. По сравнению с другими системами, использующими традиционный метод блокировок, при этом подходе наблюдается другое поведение и другая производительность. Подробнее это освещается в ports12.

13.3. Явные блокировки

Для управления параллельным доступом к данным в таблицах Postgres Pro предоставляет несколько режимов явных блокировок. Эти режимы могут применяться для блокировки данных со стороны приложения в ситуациях, когда MVCC не даёт желаемый результат. Кроме того, большинство команд Postgres Pro автоматически получают блокировки соответствующих режимов, защищающие от удаления или изменения задействованных таблиц, несовместимого с характером выполняемой команды. (Например, `TRUNCATE` не может безопасно выполняться одновременно с другими операциями с этой таблицей, так что во избежание конфликта эта команда получает блокировку `ACCESS EXCLUSIVE` для данной таблицы.)

Список текущих активных блокировок на сервере можно получить, прочитав системное представление `pg_locks`. За дополнительными сведениями о наблюдении за состоянием менеджера блокировок обратитесь к Главе 26.

13.3.1. Блокировки на уровне таблицы

В приведённом ниже списке перечислены имеющиеся режимы блокировок и контексты, где их автоматически применяет Postgres Pro. Вы можете также явно запросить любую из этих блокировок с помощью команды LOCK. Помните, что все эти режимы работают на уровне таблицы, даже если имя режима содержит слово «row»; такие имена сложились исторически. В некоторой степени эти имена отражают типичное применение каждого режима блокировки, но смысл у всех один. Единственное, что действительно отличает один режим блокировки от другого, это набор режимов, с которыми конфликтует каждый из них (см. Таблицу 13.2). Две транзакции не могут одновременно владеть блокировками конфликтующих режимов для одной и той же таблицы. (Однако учтите, что транзакция никогда не конфликтует с собой. Например, она может запросить блокировку `ACCESS EXCLUSIVE`, а затем `ACCESS SHARE` для той же таблицы.) При этом разные транзакции свободно могут одновременно владеть блокировками неконфликтующих режимов. Заметьте, что некоторые режимы блокировки конфликтуют сами с собой (например, блокировкой `ACCESS EXCLUSIVE` в один момент времени может владеть только одна транзакция), а некоторые — нет (например, блокировку `ACCESS SHARE` могут получить сразу несколько транзакций).

Режимы блокировок на уровне таблицы

`ACCESS SHARE` (`AccessShareLock`)

Конфликтует только с режимом блокировки `ACCESS EXCLUSIVE`.

Команда `SELECT` получает такую блокировку для таблиц, на которые она ссылается. Вообще говоря, блокировку в этом режиме получает любой запрос, который только *читает* таблицу, но не меняет её данные.

`ROW SHARE` (`RowShareLock`)

Конфликтует с режимами блокировки `EXCLUSIVE` и `ACCESS EXCLUSIVE`.

Команда `SELECT` устанавливает блокировку в этом режиме для всех таблиц, для которых задано указание `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` или `FOR KEY SHARE` (в дополнение к блокировкам `ACCESS SHARE` для любых других таблиц, задействованных без каких-либо явных указаний блокировки `FOR ...`).

`ROW EXCLUSIVE` (`RowExclusiveLock`)

Конфликтует с режимами блокировки `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` и `ACCESS EXCLUSIVE`.

Команды `UPDATE`, `DELETE`, `INSERT` и `MERGE` получают такую блокировку для целевой таблицы (в дополнение к блокировкам `ACCESS SHARE` для всех других задействованных таблиц). Вообще говоря, блокировку в этом режиме получает любая команда, которая *изменяет данные* в таблице.

`SHARE UPDATE EXCLUSIVE` (`ShareUpdateExclusiveLock`)

Конфликтует с режимами блокировки `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` и `ACCESS EXCLUSIVE`. Этот режим защищает таблицу от параллельного изменения схемы и запуска процесса `VACUUM`.

Запрашивается командами `VACUUM` (без `FULL`), `ANALYZE`, `CREATE INDEX CONCURRENTLY`, `CREATE STATISTICS`, `COMMENT ON`, `REINDEX CONCURRENTLY` и некоторыми формами `ALTER INDEX` и `ALTER TABLE` (за подробностями обратитесь к описанию этих команд).

`SHARE` (`ShareLock`)

Конфликтует с режимами блокировки `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` и `ACCESS EXCLUSIVE`. Этот режим защищает таблицу от параллельного изменения данных.

Запрашивается командой `CREATE INDEX` (без параметра `CONCURRENTLY`).

`SHARE ROW EXCLUSIVE` (`ShareRowExclusiveLock`)

Конфликтует с режимами блокировки `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` и `ACCESS EXCLUSIVE`. Этот режим защищает таблицу от параллельных изменений данных и при этом он является самоисключающим, так что такую блокировку может получить только один сеанс.

Запрашивается командой `CREATE TRIGGER` и некоторыми формами `ALTER TABLE`).

`EXCLUSIVE` (`ExclusiveLock`)

Конфликтует с режимами блокировки `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` и `ACCESS EXCLUSIVE`. Этот режим совместим только с блокировкой `ACCESS SHARE`, то есть параллельно с транзакцией, получившей блокировку в этом режиме, допускается только чтение таблицы.

Запрашивается командой `REFRESH MATERIALIZED VIEW CONCURRENTLY`.

`ACCESS EXCLUSIVE` (`AccessExclusiveLock`)

Конфликтует с режимами блокировок `ACCESS SHARE`, `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` и `ACCESS EXCLUSIVE`. Этот режим гарантирует, что кроме транзакции, получившей эту блокировку, никакая другая транзакция не может обращаться к таблице каким-либо способом.

Запрашивается командами `DROP TABLE`, `TRUNCATE`, `REINDEX`, `CLUSTER`, `VACUUM FULL` и `REFRESH MATERIALIZED VIEW` (без `CONCURRENTLY`). Блокировку на этом уровне запрашивают также многие виды `ALTER INDEX` и `ALTER TABLE`. В этом режиме по умолчанию запрашивают блокировку и операторы `LOCK TABLE`, если явно не выбран другой режим.

Подсказка

Только блокировка `ACCESS EXCLUSIVE` блокирует оператор `SELECT` (без `FOR UPDATE/SHARE`).

Кроме того, для поддержки системы 1С:Предприятие реализованы ещё два режима блокировок. Эти режимы не конфликтуют ни с какими режимами, описанными выше. Их можно использовать, но лучше вместо них применять рекомендательные блокировки, которые предоставляют ту же функциональность.

`APPLICATION SHARE`

Конфликтует только с режимом блокировки `APPLICATION EXCLUSIVE`.

`APPLICATION EXCLUSIVE`

Конфликтует с режимами блокировок `APPLICATION SHARE` и `APPLICATION EXCLUSIVE`.

Полученная транзакцией блокировка обычно сохраняется до конца транзакции. Но если блокировка получена после установки точки сохранения, она освобождается немедленно в случае отката к этой точке. Это согласуется с принципом действия `ROLLBACK` — эта команда отменяет эффекты всех команд после точки сохранения. То же справедливо и для блокировок, полученных в блоке исключений PL/pgSQL: при выходе из блока с ошибкой такие блокировки освобождаются.

Таблица 13.2. Конфликтующие режимы блокировки

Запрашиваемый режим блокировки	**Существующий режим блокировки**									
	`ACCESS SHARE`	`ROW SHARE`	`ROW EXCL.`	`SHARE UPDATE EXCL.`	`SHARE`	`SHARE ROW EXCL.`	`EXCL.`	`ACCESS EXCL.`	`APP. SHARE`	`APP. EXCL.`
`ACCESS SHARE`								X		
`ROW SHARE`							X	X		
`ROW EXCL.`					X	X	X	X		
`SHARE UPDATE EXCL.`				X	X	X	X	X		
`SHARE`			X	X		X	X	X		
`SHARE ROW EXCL.`			X	X	X	X	X	X		
`EXCL.`		X	X	X	X	X	X	X		
`ACCESS EXCL.`	X	X	X	X	X	X	X	X		
`APP. SHARE`										X
`APP. EXCL.`									X	X

13.3.2. Блокировки на уровне строк

Помимо блокировок на уровне таблицы, существуют блокировки на уровне строк, перечисленные ниже с контекстами, где Postgres Pro применяет их по умолчанию. Полный перечень конфликтов блокировок на уровне строк приведён в Таблице 13.3. Заметьте, что одна транзакция может владеть несколькими конфликтующими блокировками одной строки, даже в разных подтранзакциях; но две разных транзакции никогда не получат конфликтующие блокировки одной и той же строки. Блокировки на уровне строк блокируют только *запись в определённые строки*, но никак не влияют на выборку. Снимаются такие блокировки, как и блокировки на уровне таблицы, в конце транзакции или при откате к точке сохранения.

Режимы блокировки на уровне строк

`FOR UPDATE`

В режиме `FOR UPDATE` строки, выданные оператором `SELECT`, блокируются как для изменения. При этом они защищаются от блокировки, изменения и удаления другими транзакциями до завершения текущей. То есть другие транзакции, пытающиеся выполнить `UPDATE`, `DELETE`, `SELECT`

`FOR UPDATE`, `SELECT FOR NO KEY UPDATE`, `SELECT FOR SHARE` или `SELECT FOR KEY SHARE` с этими строками, будут заблокированы до завершения текущей транзакции; и наоборот, команда `SELECT FOR UPDATE` будет ожидать окончания параллельной транзакции, в которой выполнилась одна из этих команд с той же строкой, а затем установит блокировку и вернёт изменённую строку (или не вернёт, если она была удалена). Однако в транзакции `REPEATABLE READ` или `SERIALIZABLE` возникнет ошибка, если блокируемая строка изменилась с момента начала транзакции. Подробнее это обсуждается в Разделе 13.4.

Режим блокировки `FOR UPDATE` также запрашивается на уровне строки любой командой `DELETE` и командой `UPDATE`, изменяющей значения определённых столбцов. В настоящее время блокировка с `UPDATE` касается столбцов, по которым создан уникальный индекс, применимый в качестве внешнего ключа (так что на частичные индексы и индексы выражений это не распространяется), но в будущем это может поменяться.

`FOR NO KEY UPDATE`

Действует подобно `FOR UPDATE`, но запрашиваемая в этом режиме блокировка слабее: она не будет блокировать команды `SELECT FOR KEY SHARE`, пытающиеся получить блокировку тех же строк. Этот режим блокировки также запрашивается любой командой `UPDATE`, которая не требует блокировки `FOR UPDATE`.

`FOR SHARE`

Действует подобно `FOR NO KEY UPDATE`, за исключением того, что для каждой из полученных строк запрашивается разделяемая, а не исключительная блокировка. Разделяемая блокировка не позволяет другим транзакциям выполнять с этими строками `UPDATE`, `DELETE`, `SELECT FOR UPDATE` или `SELECT FOR NO KEY UPDATE`, но допускает `SELECT FOR SHARE` и `SELECT FOR KEY SHARE`.

`FOR KEY SHARE`

Действует подобно `FOR SHARE`, но устанавливает более слабую блокировку: блокируется `SELECT FOR UPDATE`, но не `SELECT FOR NO KEY UPDATE`. Блокировка разделяемого ключа не позволяет другим транзакциям выполнять команды `DELETE` и `UPDATE`, только если они меняют значение ключа (но не другие `UPDATE`), и при этом допускает выполнение команд `SELECT FOR NO KEY UPDATE`, `SELECT FOR SHARE` и `SELECT FOR KEY SHARE`.

Postgres Pro не держит информацию об изменённых строках в памяти, так что никаких ограничений на число блокируемых строк нет. Однако блокировка строки может повлечь запись на диск, например, если `SELECT FOR UPDATE` изменяет выбранные строки, чтобы заблокировать их, при этом происходит запись на диск.

Таблица 13.3. Конфликтующие блокировки на уровне строк

Запрашиваемый режим блокировки	**Текущий режим блокировки**			
	FOR KEY SHARE	**FOR SHARE**	**FOR NO KEY UPDATE**	**FOR UPDATE**
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

13.3.3. Блокировки на уровне страниц

В дополнение к блокировкам на уровне таблицы и строк, для управления доступом к страницам таблиц в общих буферах используются блокировки на уровне страниц, исключительные и разделяемые. Эти блокировки освобождаются немедленно после выборки или изменения строк. Разработчикам приложений обычно можно не задумываться о блокировках страниц, здесь они упоминаются только для полноты картины.

13.3.4. Взаимоблокировки

Частое применение явных блокировок может увеличить вероятность *взаимоблокировок*, то есть ситуаций, когда две (или более) транзакций держат блокировки так, что взаимно блокируют друг друга. Например, если транзакция 1 получает исключительную блокировку таблицы A, а затем пытается получить исключительную блокировку таблицы B, которую до этого получила транзакция 2, в данный момент требующая исключительную блокировку таблицы A, ни одна из транзакций не сможет продолжить работу. Postgres Pro автоматически выявляет такие ситуации и разрешает их, прерывая одну из сцепившихся транзакций и тем самым позволяя другой (другим) продолжить работу. (Какая именно транзакция будет прервана, обычно сложно предсказать, так что рассчитывать на определённое поведение не следует.)

Заметьте, что взаимоблокировки могут вызываться и блокировками на уровне строк (таким образом, они возможны, даже если не применяются явные блокировки). Рассмотрим случай, когда две параллельных транзакции изменяют таблицу. Первая транзакция выполняет:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

При этом она получает блокировку строки с указанным номером счёта. Затем вторая транзакция выполняет:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

Первый оператор `UPDATE` успешно получает блокировку указанной строки и изменяет данные в ней. Однако второй оператор `UPDATE` обнаруживает, что строка, которую он пытается изменить, уже заблокирована, так что он ждёт завершения транзакции, получившей блокировку. Таким образом, вторая транзакция сможет продолжиться только после завершения первой. Теперь первая транзакция выполняет:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

Первая транзакция пытается получить блокировку заданной строки, но ей это не удаётся: эта блокировка уже принадлежит второй транзакции. Поэтому первой транзакции остаётся только ждать завершения второй. В результате первая транзакция блокируется второй, а вторая — первой: происходит взаимоблокировка. Postgres Pro выявляет эту ситуацию и прерывает одну из транзакций.

Обычно лучший способ предотвращения взаимоблокировок — добиться, чтобы все приложения, обращающиеся к базе данных, запрашивали блокировки нескольких объектов единообразно. В данном примере, если бы обе транзакции изменяли строки в одном порядке, взаимоблокировка бы не произошла. Блокировки в транзакции следует упорядочивать так, чтобы первой для какого-либо объекта запрашивалась наиболее ограничивающая из тех, которые для него потребуются. Если заранее обеспечить такой порядок нельзя, взаимоблокировки можно обработать по факту, повторяя прерванные транзакции.

Если ситуация взаимоблокировки не будет выявлена, транзакция, ожидающая блокировки на уровне таблицы или строки, будет ждать её освобождения неограниченное время. Это означает, что приложения не должны оставлять транзакции открытыми долгое время (например, ожидая ввода пользователя).

13.3.5. Рекомендательные блокировки

Postgres Pro также имеет средства создания блокировок, смысл которых определяют сами приложения. Такие блокировки называются *рекомендательными*, так как система не форсирует их использование — правильно их использовать должно само приложение. Рекомендательные блокировки бывают полезны для реализаций стратегий блокирования, плохо вписывающихся в модель MVCC. Например, рекомендательные блокировки часто применяются для исполнения стратегии пессимистичной блокировки, типичной для систем управления данными «плоский файл». Хотя для этого можно использовать и дополнительные флаги в таблицах, рекомендательные блокировки работают быстрее, не приводят к раздуванию таблиц и автоматически ликвидируются сервером в конце сеанса.

В Postgres Pro есть два варианта получить рекомендательные блокировки: на уровне сеанса и на уровне транзакции. Рекомендательная блокировка, полученная на уровне сеанса, удерживается, пока она не будет явно освобождена, или до конца сеанса. В отличие от стандартных рекомендательные блокировки уровня сеанса нарушают логику транзакций — блокировка, полученная в транзакции, даже если произойдёт откат этой транзакции, будет сохраняться в сеансе; аналогично, освобождение блокировки остаётся в силе, даже если транзакция, в которой оно было выполнено, позже прерывается. Вызывающий процесс может запросить блокировку несколько раз; при этом каждому запросу блокировки должен соответствовать запрос освобождения, чтобы она была действительно освобождена. Рекомендательные блокировки на уровне транзакций, напротив, во многом похожи на обычные блокировки: они автоматически освобождаются в конце транзакций и не требуют явного освобождения. Для кратковременного применения блокировок это поведение часто более уместно, чем поведение рекомендательных блокировок на уровне сеанса. Запросы рекомендательных блокировок одного идентификатора на уровне сеанса и на уровне транзакции будут блокировать друг друга вполне предсказуемым образом. Если сеанс уже владеет данной рекомендательной блокировкой, дополнительные запросы её в том же сеансе будут всегда успешны, даже если её ожидают другие сеансы. Это утверждение справедливо вне зависимости от того, на каком уровне (сеанса или транзакции) установлены или запрашиваются новые блокировки.

Как и остальные блокировки в Postgres Pro, все рекомендательные блокировки, связанные с любыми сеансами, можно просмотреть в системном представлении `pg_locks`.

И рекомендательные, и обычные блокировки сохраняются в области общей памяти, размер которой определяется параметрами конфигурации max_locks_per_transaction и max_connections. Важно, чтобы этой памяти было достаточно, так как в противном случае сервер не сможет выдать никакую блокировку. Таким образом, число рекомендательных блокировок, которые может выдать сервер, ограничивается обычно десятками или сотнями тысяч в зависимости от конфигурации сервера.

В определённых случаях при использовании рекомендательных блокировок, особенно в запросах с явными указаниями `ORDER BY` и `LIMIT`, важно учитывать, что получаемые блокировки могут зависеть от порядка вычисления SQL-выражений. Например:

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- опасно!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

В этом примере второй вариант опасен, так как `LIMIT` не обязательно будет применяться перед вызовом функции блокировки. В результате приложение может получить блокировки, на которые оно не рассчитывает и которые оно не сможет освободить (до завершения сеанса). С точки зрения приложения такие блокировки окажутся в подвешенном состоянии, хотя они и будут отображаться в `pg_locks`.

Функции, предназначенные для работы с рекомендательными блокировками, описаны в Подразделе 9.28.10.

13.4. Проверки целостности данных на уровне приложения

Используя транзакции Read Committed, очень сложно обеспечить целостность данных с точки зрения бизнес-логики, так как представление данных смещается с каждым оператором и даже один оператор может не ограничиваться своим снимком состояния в случае конфликта записи.

Хотя транзакция Repeatable Read получает стабильное представление данных в процессе выполнения, с использованием снимков MVCC для проверки целостности данных всё же связаны тонкие моменты, включая так называемые *конфликты чтения/записи*. Если одна транзакция записывает данные, а другая в это же время пытается их прочитать (до или после записи), она не может увидеть результат работы первой. В таком случае создаётся впечатление, что читающая транзакция выполняется первой вне зависимости от того, какая из них была начата или зафиксирована раньше. Если этим всё и ограничивается, нет никаких проблем, но если читающая транзакция также пишет данные, которые читает параллельная транзакция, получается, что теперь эта транзакция будет исполняться, как будто она запущена перед другими вышеупомянутыми. Если же транзакция, которая должна исполняться как последняя, на самом деле зафиксирована первой, в графе упорядоченных транзакций легко может возникнуть цикл. И когда он возникает, проверки целостности не будут работать правильно без дополнительных мер.

Как было сказано в Подразделе 13.2.3, сериализуемые транзакции представляют собой те же транзакции Repeatable Read, но дополненные неблокирующим механизмом отслеживания опасных условий конфликтов чтения/записи. Когда выявляется условие, приводящее к циклу в порядке транзакций, одна из этих транзакций откатывается и этот цикл таким образом разрывается.

13.4.1. Обеспечение согласованности в сериализуемых транзакциях

Если для всех операций чтения и записи, нуждающихся в согласованном представлении данных, используются транзакции уровня изоляции Serializable, это обеспечивает необходимую согласованность без дополнительных усилий. Приложения из других окружений, применяющие сериализуемые транзакции для обеспечения целостности, в Postgres Pro в этом смысле будут «просто работать».

Применение этого подхода избавляет программистов приложений от лишних сложностей, если приложение использует инфраструктуру, которая автоматически повторяет транзакции в случае отката из-за сбоев сериализации. Возможно, `serializable` стоит даже установить в качестве уровня изоляции по умолчанию (`default_transaction_isolation`). Также имеет смысл принять меры для предотвращения использования других уровней изоляции, непреднамеренного или с целью обойти проверки целостности, например проверять уровень изоляции в триггерах.

Рекомендации по увеличению быстродействия приведены в Подразделе 13.2.3.

Предупреждение: сериализуемые транзакции и репликация данных

Защита целостности с применением сериализуемых транзакций пока ещё не поддерживается в режиме горячего резерва (Раздел 25.4) или логических реплик. Поэтому там, где применяется горячий резерв или логическая репликация, следует использовать уровень Repeatable Read и явные блокировки на главном сервере.

13.4.2. Применение явных блокировок для обеспечения согласованности

Когда возможны несериализуемые операции записи, для обеспечения целостности строк и защиты от одновременных изменений, следует использовать `SELECT FOR UPDATE`, `SELECT FOR SHARE` или соответствующий оператор `LOCK TABLE`. (`SELECT FOR UPDATE` и `SELECT FOR SHARE` защищают от параллельных изменений только возвращаемые строки, тогда как `LOCK TABLE` блокирует всю таблицу.) Это следует учитывать, перенося в Postgres Pro приложения из других СУБД.

Мигрируя в Postgres Pro из других СУБД также следует учитывать, что команда `SELECT FOR UPDATE` сама по себе не гарантирует, что параллельная транзакция не изменит или не удалит выбранную строку. Для получения такой гарантии в Postgres Pro нужно именно изменить эту строку, даже если никакие значения в ней менять не требуется. `SELECT FOR UPDATE` *временно блокирует* другие транзакции, не давая им получить ту же блокировку или выполнить команды `UPDATE` или `DELETE`, которые бы повлияли на заблокированную строку, но как только транзакция, владеющая этой блокировкой, фиксируется или откатывается, заблокированная транзакция сможет выполнить конфликтующую операцию, если только для данной строки действительно не был выполнен `UPDATE`, пока транзакция владела блокировкой.

Реализация глобальной целостности с использованием несериализуемых транзакций MVCC требует более вдумчивого подхода. Например, банковскому приложению может потребоваться проверить, равняется ли сумма всех расходов в одной таблице сумме приходов в другой, при том, что обе таблицы активно изменяются. Просто сравнивать результаты двух успешных последовательных команд `SELECT sum(...)` в режиме Read Committed нельзя, так как вторая команда может захватить результаты транзакций, пропущенных первой. Подсчитывая суммы в одной транзакции Repeatable Read, можно получить точную картину только для транзакций, которые были зафиксированы до начала данной, но при этом может возникнуть законный вопрос — будет ли этот результат актуален тогда, когда он будет выдан. Если транзакция Repeatable Read сама вносит какие-то изменения, прежде чем проверять равенство сумм, полезность этой проверки становится ещё более сомнительной, так как при проверке будут учитываться некоторые, но не все изменения, произошедшие после начала транзакции. В таких случаях предусмотрительный разработчик может заблокировать все таблицы, задействованные в проверке, чтобы получить картину действительности, не вызывающую сомнений. Для этого применяется блокировка `SHARE` (или более строгая), которая гарантирует, что в заблокированной таблице не будет незафиксированных изменений, за исключением тех, что внесла текущая транзакция.

Также заметьте, что, применяя явные блокировки для предотвращения параллельных операций записи, следует использовать либо режим Read Committed, либо в режиме Repeatable Read обязательно получать блокировки прежде, чем выполнять запросы. Блокировка, получаемая транзакцией Repeatable Read, гарантирует, что никакая другая транзакция, изменяющая таблицу, не выполняется, но если снимок состояния, полученный транзакцией, предшествует блокировке, он может не включать на данный момент уже зафиксированные изменения. Снимок состояния в транзакции Repeatable Read создаётся фактически на момент начала первой команды выборки или изменения данных (`SELECT`, `INSERT`, `UPDATE`, `DELETE` или `MERGE`), так что получить явные блокировки можно до того, как он будет сформирован.

13.5. Обработка сбоев сериализации

На уровнях изоляции Repeatable Read и Serializable могут выдаваться ошибки с целью предотвращения аномалий сериализации. Как отмечалось ранее, приложения, использующие эти уровни, должны быть готовы повторить транзакции, завершившиеся сбоем из-за ошибок сериализации. Текст такого сообщения об ошибке может варьироваться в зависимости от конкретных обстоятельств, но код SQLSTATE всегда будет `40001` (`serialization_failure`).

Также может иметь смысл повторять запросы в случае ошибок взаимоблокировки. Такие ошибки имеют код SQLSTATE `40P01` (`deadlock_detected`).

В некоторых случаях также целесообразно повторять запросы после ошибок уникальности ключа, которые имеют код SQLSTATE `23505` (`unique_violation`), и ошибок нарушения исключения, которые имеют код SQLSTATE `23P01` (`exclusion_violation`). Например, если приложение выбирает новое значение для столбца первичного ключа, предварительно выяснив, какие значения уже есть, оно может столкнуться с ошибкой уникальности ключа, потому что другой экземпляр этого же приложения одновременно выбрал то же значение. По сути, это ошибка сериализации, но сервер не обнаруживает её как таковую, потому что он не может «видеть» связь между добавляемым значением и предыдущим чтением. Есть также некоторые особые случаи, когда сервер выдаёт ошибку уникального ключа или нарушения исключения, даже если у него в принципе достаточно информации, чтобы определить, что основной причиной является проблема сериализации. Тогда как запросы, вызвавшие ошибки `serialization_failure`, рекомендуется просто повторять, это не распространяется на запросы, завершающиеся ошибками с другими кодами, поскольку такие ошибки могут свидетельствовать о постоянных проблемах, а не о временных сбоях.

Повторять транзакцию важно целиком, включая всю логику, которая решает, какой SQL выдавать и/или какие значения использовать. Postgres Pro не предлагает средства автоматического повторения запросов, так как не может гарантировать их правильность.

Повторение транзакции не гарантирует, что такая транзакция будет завершена успешно; может потребоваться несколько повторных попыток. Когда уровень конкуренции очень высок, для успешного выполнения транзакции может потребоваться много попыток. В случаях, когда конфликт связан с подготовленной транзакцией, разрешить конфликт может быть невозможно, пока подготовленная транзакция не будет зафиксирована или отменена.

13.6. Ограничения

Некоторые команды DDL, в настоящее время это `TRUNCATE` и формы `ALTER TABLE`, перезаписывающие таблицу, не являются безопасными с точки зрения MVCC. Это значит, что после фиксации усечения или перезаписи таблица окажется пустой для всех параллельных транзакций, если они работают со снимком, полученным перед фиксацией такой команды DDL. Это может проявиться только в транзакции, которая не обращалась к таблице до момента начала команды DDL — любая транзакция, которая обращалась к ней раньше, получила бы как минимум блокировку `ACCESS SHARE`, которая заблокировала бы эту команду DDL до завершения транзакции. Поэтому такие команды не приводят ни к каким видимым несоответствиям с содержимым таблицы при последовательных запросах к целевой таблице, хотя возможно видимое несоответствие между содержимым целевой таблицы и другими таблицами в базе данных.

Поддержка уровня изоляции Serializable ещё не реализована для целевых серверов горячего резерва (они описываются в Разделе 25.4). На данный момент самый строгий уровень изоляции, поддерживаемый в режиме горячего резерва, это Repeatable Read. Хотя и тогда, когда главный сервер выполняет запись в транзакциях Serializable, все резервные серверы в итоге достигают согласованного состояния, но транзакция Repeatable Read на резервном сервере иногда может увидеть промежуточное состояние, не соответствующее результату последовательного выполнения транзакций на главном сервере.

Внутренние обращения к системным каталогам осуществляются за рамками уровня изоляции текущей транзакции. Это означает, что создаваемые объекты базы данных, например таблицы, оказываются видимыми для параллельных транзакций уровня Repeatable Read и Serializable, несмотря на то, что строки в них не видны. С другой стороны, запросы, обращающиеся к системным каталогам напрямую, не будут видеть строки, представляющие недавно созданные объекты базы данных, если эти запросы используют более высокие уровни изоляции.

13.7. Блокировки и индексы

Хотя Postgres Pro обеспечивает неблокирующий доступ на чтение/запись к данным таблиц, для индексов в настоящий момент это поддерживается не в полной мере. Postgres Pro управляет доступом к различным типам индексов следующим образом:

Индексы типа B-дерево, GiST и SP-GiST

Для управления чтением/записью используются кратковременные блокировки на уровне страницы, исключительные и разделяемые. Блокировки освобождаются сразу после извлечения или добавления строки индекса. Эти типы индексов обеспечивают максимальное распараллеливание операций, не допуская взаимоблокировок.

Хеш-индексы

Для управления чтением/записью используются блокировки на уровне групп хеша. Блокировки освобождаются после обработки всей группы. Такие блокировки с точки зрения распараллеливания лучше, чем блокировки на уровне индекса, но не исключают взаимоблокировок, так как они сохраняются дольше, чем выполняется одна операция с индексом.

Индексы GIN

Для управления чтением/записью используются кратковременные блокировки на уровне страницы, исключительные и разделяемые. Блокировки освобождаются сразу после извлечения или добавления строки индекса. Но заметьте, что добавление значения в поле с GIN-индексом обычно влечёт добавление нескольких ключей индекса, так что GIN может проделывать целый ряд операций для одного значения.

В настоящее время в многопоточной среде наиболее производительны индексы-B-деревья; и так как они более функциональны, чем хеш-индексы, их рекомендуется использовать в такой среде для приложений, когда нужно индексировать скалярные данные. Если же нужно индексировать не скалярные данные, B-деревья не подходят, и вместо них следует использовать индексы GiST, SP-GiST или GIN.

Глава 14. Оптимизация производительности

Быстродействие запросов зависит от многих факторов. На некоторые из них могут воздействовать пользователи, а другие являются фундаментальными особенностями системы. В этой главе приводятся полезные советы, которые помогут понять их и оптимизировать производительность Postgres Pro.

14.1. Использование `EXPLAIN`

Выполняя любой полученный запрос, Postgres Pro разрабатывает для него *план запроса*. Выбор правильного плана, соответствующего структуре запроса и характеристикам данным, крайне важен для хорошей производительности, поэтому в системе работает сложный *планировщик*, задача которого — подобрать хороший план. Узнать, какой план был выбран для какого-либо запроса, можно с помощью команды `EXPLAIN`. Понимание плана — это искусство, и чтобы овладеть им, нужен определённый опыт, но этот раздел расскажет о самых простых вещах.

Приведённые ниже примеры показаны на тестовой базе данных, которая создаётся для выявления регрессий в исходных кодах PostgreSQL текущей версии. Для неё предварительно выполняется `VACUUM ANALYZE`. Вы должны получить похожие результаты, если возьмёте ту же базу данных и проделаете следующие действия, но примерная стоимость и ожидаемое число строк у вас может немного отличаться из-за того, что статистика команды `ANALYZE` рассчитывается по случайной выборке, а оценки стоимости зависят от конкретной платформы.

В этих примерах используется текстовый формат вывода `EXPLAIN`, принятый по умолчанию, как более компактный и удобный для восприятия человеком. Если вывод `EXPLAIN` нужно передать какой-либо программе для дальнейшего анализа, лучше использовать один из машинно-ориентированных форматов (XML, JSON или YAML).

14.1.1. Азы `EXPLAIN`

Структура плана запроса представляет собой дерево *узлов плана*. Узлы на нижнем уровне дерева — это узлы сканирования, которые возвращают необработанные данные таблицы. Разным типам доступа к таблице соответствуют разные узлы: последовательное сканирование, сканирование индекса и сканирование битовой карты. Источниками строк могут быть не только таблицы, но и например, предложения `VALUES` и функции, возвращающие множества во `FROM`, и они представляются отдельными типами узлов сканирования. Если запрос требует объединения, агрегатных вычислений, сортировки или других операций с исходными строками, над узлами сканирования появляются узлы, обозначающие эти операции. И так как обычно операции могут выполняться разными способами, на этом уровне тоже могут быть узлы разных типов. В выводе команды `EXPLAIN` для каждого узла в дереве плана отводится одна строка, где показывается базовый тип узла плюс оценка стоимости выполнения данного узла, которую сделал для него планировщик. Если для узла выводятся дополнительные свойства, в вывод могут добавляться дополнительные строки, с отступом от основной информации узла. В самой первой строке (основной строке самого верхнего узла) выводится общая стоимость выполнения для всего плана; именно это значение планировщик старается минимизировать.

Взгляните на следующий простейший пример, просто иллюстрирующий формат вывода:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

Этот запрос не содержит предложения `WHERE`, поэтому он должен просканировать все строки таблицы, так что планировщик выбрал план простого последовательного сканирования. Числа, перечисленные в скобках (слева направо), имеют следующий смысл:

- Приблизительная стоимость запуска. Это время, которое проходит, прежде чем начнётся этап вывода данных, например для сортирующего узла это время сортировки.
- Приблизительная общая стоимость. Она вычисляется в предположении, что узел плана выполняется до конца, то есть возвращает все доступные строки. На практике родительский узел может досрочно прекратить чтение строк дочернего (см. приведённый ниже пример с `LIMIT`).
- Ожидаемое число строк, которое должен вывести этот узел плана. При этом так же предполагается, что узел выполняется до конца.
- Ожидаемый средний размер строк, выводимых этим узлом плана (в байтах).

Стоимость может измеряться в произвольных единицах, определяемых параметрами планировщика (см. Подраздел 18.7.2). Традиционно единицей стоимости считается операция чтения страницы с диска; то есть seq_page_cost обычно равен `1.0`, а другие параметры задаётся относительно него. Примеры в этом разделе выполняются со стандартными параметрами стоимости.

Важно понимать, что стоимость узла верхнего уровня включает стоимость всех его потомков. Также важно осознавать, что эта стоимость отражает только те факторы, которые учитывает планировщик. В частности, она не зависит от времени, необходимого для преобразования выходных значений в текстовую форму или их передачи клиенту, хотя оно может составлять значительную часть общего времени выполнения запроса. Тем не менее планировщик игнорирует эту стоимость, так как он всё равно не сможет изменить её, выбрав другой план. (Мы верим в то, что любой правильный план запроса выдаёт один и тот же набор строк.)

Значение `rows` здесь имеет особенность — оно выражает не число строк, обработанных или просканированных узлом плана, а число строк, выданных этим узлом. Часто оно окажется меньше числа просканированных строк в результате применённой к узлу фильтрации по условиям `WHERE`. В идеале, на верхнем уровне это значение будет приблизительно равно числу строк, которое фактически возвращает, изменяет или удаляет запрос.

Возвращаясь к нашему примеру:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

Эти числа получаются очень просто. Выполните:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

и вы увидите, что `tenk1` содержит 345 страниц диска и 10 000 строк. Общая стоимость вычисляется как (число_чтений_диска * seq_page_cost) + (число_просканированных_строк * cpu_tuple_cost). По умолчанию, `seq_page_cost` равно 1.0, а `cpu_tuple_cost` — 0.01, так что приблизительная стоимость запроса равна (345 * 1.0) + (10000 * 0.01) = 445.

Теперь давайте изменим запрос, добавив в него предложение `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..470.00 rows=7000 width=244)
   Filter: (unique1 < 7000)
```

Заметьте, что в выводе `EXPLAIN` показано, что условие `WHERE` применено как «фильтр» к узлу плана Seq Scan (Последовательное сканирование). Это означает, что узел плана проверяет это условие для каждого просканированного им узла и выводит только те строки, которые удовлетворяют ему. Предложение `WHERE` повлияло на оценку числа выходных строк. Однако при сканировании потребуется прочитать все 10 000 строк, поэтому общая стоимость не уменьшилась. На деле она даже немного увеличилась (на 10000 * cpu_operator_cost, если быть точными), отражая дополнительное время, которое потребуется процессору на проверку условия `WHERE`.

Фактическое число строк результата этого запроса будет равно 7 000, но значение `rows` даёт только приблизительное значение. Если вы попытаетесь повторить этот эксперимент, вы можете получить немного другую оценку; более того, она может меняться после каждой команды `ANALYZE`, так как `ANALYZE` получает статистику по случайной выборке таблицы.

Теперь давайте сделаем ограничение более избирательным:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.06..224.98 rows=100 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
         Index Cond: (unique1 < 100)
```

В данном случае планировщик решил использовать план из двух этапов: сначала дочерний узел плана просматривает индекс и находит в нём адреса строк, соответствующих условию индекса, а затем верхний узел собственно выбирает эти строки из таблицы. Выбирать строки по отдельности гораздо дороже, чем просто читать их последовательно, но так как читать придётся не все страницы таблицы, это всё равно будет дешевле, чем сканировать всю таблицу. (Использование двух уровней плана объясняется тем, что верхний узел сортирует адреса строк, выбранных из индекса, в физическом порядке, прежде чем читать, чтобы снизить стоимость отдельных чтений. Слово «bitmap» (битовая карта) в имени узла обозначает механизм, выполняющий сортировку.)

Теперь давайте добавим ещё одно условие в предложение `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..225.20 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
         Index Cond: (unique1 < 100)
```

Добавленное условие `stringu1 = 'xxx'` уменьшает оценку числа результирующих строк, но не стоимость запроса, так как просматриваться будет тот же набор строк, что и раньше. Заметьте, что условие на `stringu1` не добавляется в качестве условия индекса, так как индекс построен только по столбцу `unique1`. Вместо этого оно применяется как фильтр к строкам, полученным по индексу. В результате стоимость даже немного увеличилась, отражая добавление этой проверки.

В некоторых случаях планировщик предпочтёт «простой» план сканирования индекса:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

В плане такого типа строки таблицы выбираются в порядке индекса, в результате чего чтение их обходится дороже, но так как их немного, дополнительно сортировать положения строк не стоит. Вы часто будете встречать этот тип плана в запросах, которые выбирают всего одну строку. Также он часто задействуется там, где условие `ORDER BY` соответствует порядку индекса, так как в этих случаях для выполнения `ORDER BY` не требуется дополнительный шаг сортировки. В этом примере добавленная конструкция `ORDER BY unique1` будет использовать тот же план, потому что индекс уже неявно обеспечивает нужный порядок.

Планировщик может обработать конструкцию `ORDER BY` несколькими способами. Предыдущий пример показывает, что нужный порядок может быть получен неявным образом. Также планировщик может задействовать явную операцию `Sort`:

```
EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;

                            QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=1109.39..1134.39 rows=10000 width=244)
   Sort Key: unique1
   ->  Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

Если подплан гарантирует сортировку по префиксу заданных ключей сортировки, планировщик может применить операцию `Incremental sort` (инкрементальную сортировку):

```
EXPLAIN SELECT * FROM tenk1 ORDER BY hundred, ten LIMIT 100;

                                              QUERY PLAN
-------------------------------------------------------------------
-----------------------------
 Limit  (cost=19.35..39.49 rows=100 width=244)
   ->  Incremental Sort  (cost=19.35..2033.39 rows=10000 width=244)
         Sort Key: hundred, ten
         Presorted Key: hundred
         ->  Index Scan using tenk1_hundred on tenk1  (cost=0.29..1574.20 rows=10000
 width=244)
```

С инкрементальной сортировкой, в отличие от обычной, кортежи могут выдаваться до завершения сортировки всего результата, это в частности позволяет оптимизировать запросы с `LIMIT`. Кроме того, для инкрементальной сортировки может потребоваться меньше памяти, вследствие чего уменьшается вероятность вытеснения сортируемых данных на диск, но с другой стороны, требуется разделять результирующее множество на несколько частей, что влечёт дополнительные накладные расходы.

Если в таблице есть отдельные индексы по разным столбцам, фигурирующим в `WHERE`, планировщик может выбрать сочетание этих индексов (с AND и OR):

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.07..60.11 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.07..25.07 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

Но для этого потребуется обойти оба индекса, так что это не обязательно будет выгоднее, чем просто просмотреть один индекс, а второе условие обработать как фильтр. Измените диапазон и вы увидите, как это повлияет на план.

Следующий пример иллюстрирует эффекты `LIMIT`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.28 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..70.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
```

```
         Filter: (unique1 < 100)
```

Это тот же запрос, что и раньше, но добавили мы в него `LIMIT`, чтобы возвращались не все строки, и планировщик решает выполнять запрос по-другому. Заметьте, что общая стоимость и число строк для узла Index Scan рассчитываются в предположении, что он будет выполняться полностью. Однако узел Limit должен остановиться, получив только пятую часть всех строк, так что его стоимость будет составлять одну пятую от вычисленной ранее, и это и будет итоговой оценкой стоимости запроса. С другой стороны, планировщик мог бы просто добавить в предыдущий план узел Limit, но это не избавило бы от затрат на запуск сканирования битовой карты, а значит, общая стоимость была бы выше 25 единиц.

Давайте попробуем соединить две таблицы по столбцам, которые мы уже использовали:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.50 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.90 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

В этом плане появляется узел соединения с вложенным циклом, на вход которому поступают данные от двух его потомков, узлов сканирования. Эту структуру плана отражает отступ основных строк его узлов. Первый, или «внешний», потомок соединения — узел сканирования битовой карты, похожий на те, что мы видели раньше. Его стоимость и число строк те же, что мы получили бы для запроса `SELECT ... WHERE unique1 < 10`, так как к этому узлу добавлено предложение `WHERE unique1 < 10`. Условие `t1.unique2 = t2.unique2` ещё не учитывается, поэтому оно не влияет на число строк узла внешнего сканирования. Узел соединения с вложенным циклом будет выполнять узел «внутреннего» потомка для каждой строки, полученной из внешнего потомка. Значения столбцов из текущей внешней строки могут использоваться во внутреннем сканировании (в данном случае это значение `t1.unique2`), поэтому мы получаем план и стоимость примерно такие, как и раньше для простого запроса `SELECT ... WHERE t2.unique2 =` *`константа`*. (На самом деле оценочная стоимость немного меньше, в предположении, что при неоднократном сканировании индекса по `t2` положительную роль сыграет кеширование.) В результате стоимость узла цикла складывается из стоимости внешнего сканирования, цены внутреннего сканирования, умноженной на число строк (здесь 10 * 7.90), и небольшой наценки за обработку соединения.

В этом примере число выходных строк соединения равно произведению чисел строк двух узлов сканирования, но это не всегда будет так, потому что в дополнительных условиях `WHERE` могут упоминаться обе таблицы, так что применить их можно будет только в точке соединения, а не в одном из узлов сканирования. Например:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                                         QUERY PLAN
-------------------------------------------------------------------
--------------------------
 Nested Loop  (cost=4.65..49.36 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
```

```
              Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10
 width=244)
              Index Cond: (unique2 < 10)
```

Условие `t1.hundred < t2.hundred` не может быть проверено в индексе `tenk2_unique2`, поэтому оно применяется в узле соединения. Это уменьшает оценку числа выходных строк, тогда как число строк в узлах сканирования не меняется.

Заметьте, что здесь планировщик решил «материализовать» внутреннее отношение соединения, поместив поверх него узел плана Materialize (Материализовать). Это значит, что сканирование индекса `t2` будет выполняться только единожды, при том, что узлу вложенного цикла соединения потребуется прочитать данные десять раз, по числу строк во внешнем соединении. Узел Materialize сохраняет считанные данные в памяти, чтобы затем выдать их из памяти на следующих проходах.

Выполняя внешние соединения, вы можете встретить узлы плана с присоединёнными условиями, как обычными «Filter», так и «Join Filter» (Фильтр соединения). Условия Join Filter формируются из предложения `ON` для внешнего соединения, так что если строка не удовлетворяет условию Join Filter, она всё же выдаётся как строка, дополненная значениями NULL. Обычное же условие Filter применяется после правил внешнего соединения и поэтому полностью исключает строки. Во внутреннем соединении оба этих фильтра работают одинаково.

Если немного изменить избирательность запроса, мы можем получить совсем другой план соединения:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
-------------------------------------------------------------------
-----------------------
 Hash Join  (cost=226.23..709.73 rows=100 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=224.98..224.98 rows=100 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98 rows=100 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100
 width=0)
                     Index Cond: (unique1 < 100)
```

Здесь планировщик выбирает соединение по хешу, при котором строки одной таблицы записываются в хеш-таблицу в памяти, после чего сканируется другая таблица и для каждой её строки проверяется соответствие по хеш-таблице. Обратите внимание, что и здесь отступы отражают структуру плана: результат сканирования битовой карты по `tenk1` подаётся на вход узлу Hash, который конструирует хеш-таблицу. Затем она передаётся узлу Hash Join, который читает строки из узла внешнего потомка и проверяет их по этой хеш-таблице.

Ещё один возможный тип соединения — соединение слиянием:

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
-------------------------------------------------------------------
-----------------------
 Merge Join  (cost=0.56..233.49 rows=10 width=488)
```

```
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..643.28 rows=100
 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..166.28 rows=1000
 width=244)
```

Соединение слиянием требует, чтобы входные данные для него были отсортированы по ключам соединения. В этом примере все входные данные сортируются после сканирования индекса, при котором все строки просматриваются в правильном порядке, однако можно использовать и последовательное сканирование и сортировку. (Выполнение последовательного сканирования и сортировки часто бывает быстрее сканирования индекса, когда нужно отсортировать много строк, так как при сканировании по индексу обращения к диску не упорядочены.)

Один из способов посмотреть различные планы — принудить планировщик не считать выбранную им стратегию самой выгодной, используя флаги, описанные в Подразделе 18.7.1. (Это полезный, хотя и грубый инструмент. См. также Раздел 14.3.) Например, если мы убеждены, что соединение слиянием — не лучший тип соединения в предыдущем примере, мы можем попробовать:

```
SET enable_mergejoin = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
-------------------------------------------------------------------
-----------------------
 Hash Join  (cost=226.23..344.08 rows=10 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on onek t2  (cost=0.00..114.00 rows=1000 width=244)
   ->  Hash  (cost=224.98..224.98 rows=100 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98 rows=100 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100
 width=0)
                     Index Cond: (unique1 < 100)
```

Видно, что планировщик в этом случае считает соединение по хешу примерно на 50% дороже, чем соединение слиянием. Конечно, может возникнуть вопрос — а правильно ли это? Мы можем ответить на него, используя описанную ниже команду `EXPLAIN ANALYZE`.

Некоторые планы запросов включают *подпланы*, которые формируются при выполнении вложенных запросов `SELECT` внутри основных запросов. Такие запросы могут иногда преобразовываться в обычные планы соединения, однако если это невозможно, получаются планы, подобные приведённому ниже:

```
EXPLAIN VERBOSE SELECT unique1
FROM tenk1 t
WHERE t.ten < ALL (SELECT o.ten FROM onek o WHERE o.four = t.four);

                               QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on public.tenk1 t  (cost=0.00..586095.00 rows=5000 width=4)
   Output: t.unique1
   Filter: (ALL (t.ten < (SubPlan 1).col1))
   SubPlan 1
     ->  Seq Scan on public.onek o  (cost=0.00..116.50 rows=250 width=4)
           Output: o.ten
           Filter: (o.four = t.four)
```

Этот искусственный пример свидетельствует о нескольких моментах: значения основного плана могут передаваться подплану (здесь передаётся значение `t.four`), а результаты вложенного запроса `SELECT` доступны для основного плана. Результирующие значения выводятся командой `EXPLAIN` с нотациями типа `(`*`имя_подплана`*`).col`*`N`*, которые ссылаются на *N*-й столбец вывода вложенного запроса `SELECT`.

В примере выше благодаря оператору `ALL` подплан применяется для каждой строки внешнего запроса (с высокой оценочной стоимостью). Чтобы этого избежать, для некоторых запросов может использоваться *подплан с хешированием*:

```
EXPLAIN SELECT *
FROM tenk1 t
WHERE t.unique1 NOT IN (SELECT o.unique1 FROM onek o);

                                         QUERY PLAN
--------------------------------------------------------------------
-------------------------
 Seq Scan on tenk1 t  (cost=61.77..531.77 rows=5000 width=244)
   Filter: (NOT (ANY (unique1 = (hashed SubPlan 1).col1)))
   SubPlan 1
     ->  Index Only Scan using onek_unique1 on onek o  (cost=0.28..59.27 rows=1000
 width=4)
(4 rows)
```

В этом случае подплан применяется один раз, и его вывод загружается в хеш-таблицу в памяти, к которой потом обращается оператор `ANY` внешнего запроса. Чтобы такой подплан работал, нужно, чтобы вложенный запрос `SELECT` не ссылался на переменные внешнего запроса и оператор сравнения в рамках `ANY` поддерживал хеширование.

Если вложенный запрос `SELECT` не только не ссылается на переменные внешнего запроса, но и не может вернуть больше одной строки, он может быть выполнен с применением *инициализирующего плана*:

```
EXPLAIN VERBOSE SELECT unique1
FROM tenk1 t1 WHERE t1.ten = (SELECT (random() * 10)::integer);

                             QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on public.tenk1 t1  (cost=0.02..470.02 rows=1000 width=4)
   Output: t1.unique1
   Filter: (t1.ten = (InitPlan 1).col1)
   InitPlan 1
     ->  Result  (cost=0.00..0.02 rows=1 width=4)
           Output: ((random() * '10'::double precision))::integer
```

Такой план применяется только один раз за время выполнения внешнего плана, а его результаты сохраняются и могут быть повторно использованы для строк внешнего плана. Так, в этом примере оценка `random()` происходит только один раз, и все значения `t1.ten` сравниваются с одним и тем же случайно выбранным целым числом. Без конструкции с вложенным запросом `SELECT` процесс выглядел бы несколько иначе.

14.1.2. EXPLAIN ANALYZE

Точность оценок планировщика можно проверить, используя команду `EXPLAIN` с параметром `ANALYZE`. С этим параметром `EXPLAIN` на самом деле выполняет запрос, а затем выводит фактическое число строк и время выполнения, накопленное в каждом узле плана, вместе с теми же оценками, что выдаёт обычная команда `EXPLAIN`. Например, мы можем получить примерно такой результат:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
```

```
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                                 QUERY PLAN
-------------------------------------------------------------------
---------------------------------------------------------------
 Nested Loop  (cost=4.65..118.50 rows=10 width=488) (actual time=0.017..0.051 rows=10
 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.38 rows=10 width=244) (actual
 time=0.009..0.017 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         Heap Blocks: exact=10
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
 (actual time=0.004..0.004 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.90 rows=1 width=244)
 (actual time=0.003..0.003 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Planning Time: 0.485 ms
 Execution Time: 0.073 ms
```

Заметьте, что значения «actual time» (фактическое время) приводятся в миллисекундах, тогда как оценки `cost` (стоимость) выражаются в произвольных единицах, так что они вряд ли совпадут. Обычно важнее определить, насколько приблизительная оценка числа строк близка к действительности. В этом примере они в точности совпали, но на практике так бывает редко.

В некоторых планах запросов некоторый внутренний узел может выполняться неоднократно. Например, внутреннее сканирование индекса будет выполняться для каждой внешней строки во вложенном цикле верхнего уровня. В таких случаях значение `loops` (циклы) показывает, сколько всего раз выполнялся этот узел, а фактическое время и число строк вычисляется как среднее по всем итерациям. Это делается для того, чтобы полученные значения можно было сравнить с выводимыми приблизительными оценками. Чтобы получить общее время, затраченное на выполнение узла, время одной итерации нужно умножить на значение `loops`. В показанном выше примере мы потратили в общей сложности 0.030 мс на сканирование индекса в `tenk2`.

В ряде случаев `EXPLAIN ANALYZE` выводит дополнительную статистику по выполнению, включающую не только время выполнения узлов и число строк. Для узлов Sort и Hash, например, выводится следующая информация:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                       QUERY PLAN
-------------------------------------------------------------------
-------------------------------------------------------------------------
 Sort  (cost=713.05..713.30 rows=100 width=488) (actual time=2.995..3.002 rows=100
 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 74kB
   ->  Hash Join  (cost=226.23..709.73 rows=100 width=488) (actual time=0.515..2.920
 rows=100 loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual
 time=0.026..1.790 rows=10000 loops=1)
         ->  Hash  (cost=224.98..224.98 rows=100 width=244) (actual time=0.476..0.477
 rows=100 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 35kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.06..224.98 rows=100 width=244)
 (actual time=0.030..0.450 rows=100 loops=1)
```

```
                    Recheck Cond: (unique1 < 100)
                    Heap Blocks: exact=90
                    ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100
 width=0) (actual time=0.013..0.013 rows=100 loops=1)
                          Index Cond: (unique1 < 100)
 Planning Time: 0.187 ms
 Execution Time: 3.036 ms
```

Для узла Sort показывается использованный метод и место сортировки (в памяти или на диске), а также задействованный объём памяти. Для узла Hash выводится число групп и пакетов хеша, а также максимальный объём, который заняла в памяти хеш-таблица. (Если число пакетов больше одного, часть хеш-таблицы будет выгружаться на диск и занимать какое-то пространство, но его объём здесь не показывается.)

Другая полезная дополнительная информация — число строк, удалённых условием фильтра:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                               QUERY PLAN
-------------------------------------------------------------------
--------------------------------------
 Seq Scan on tenk1  (cost=0.00..470.00 rows=7000 width=244) (actual time=0.030..1.995
 rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning Time: 0.102 ms
 Execution Time: 2.145 ms
```

Эти значения могут быть особенно ценны для условий фильтра, применённых к узлам соединения. Строка «Rows Removed» выводится, только когда условие фильтра отбрасывает минимум одну просканированную строку или потенциальную пару соединения, если это узел соединения.

Похожую ситуацию можно наблюдать при сканировании «неточного» индекса. Например, рассмотрим этот план поиска многоугольников, содержащих указанную точку:

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                              QUERY PLAN
-------------------------------------------------------------------
-----------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.09 rows=1 width=85) (actual time=0.023..0.023
 rows=0 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 7
 Planning Time: 0.039 ms
 Execution Time: 0.033 ms
```

Планировщик полагает (и вполне справедливо), что таблица слишком мала для сканирования по индексу, поэтому он выбирает последовательное сканирование, при котором все строки отбрасываются условием фильтра. Но если мы принудим его выбрать сканирование по индексу, мы получим:

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                        QUERY PLAN
-------------------------------------------------------------------
-------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=85) (actual
 time=0.074..0.074 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
```

```
   Rows Removed by Index Recheck: 1
 Planning Time: 0.039 ms
 Execution Time: 0.098 ms
```

Здесь мы видим, что индекс вернул одну потенциально подходящую строку, но затем она была отброшена при перепроверке условия индекса. Это объясняется тем, что индекс GiST является «неточным» для проверок включений многоугольников: фактически он возвращает строки с многоугольниками, перекрывающими точку по координатам, а затем для этих строк нужно выполнять точную проверку.

`EXPLAIN` принимает параметр `BUFFERS` (который также можно применять с `ANALYZE`), включающий ещё более подробную статистику выполнения запроса:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                                                QUERY PLAN
--------------------------------------------------------------------
---------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.07..60.11 rows=10 width=244) (actual
 time=0.105..0.114 rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Heap Blocks: exact=10
   Buffers: shared hit=14 read=3
   ->  BitmapAnd  (cost=25.07..25.07 rows=10 width=0) (actual time=0.100..0.101 rows=0
 loops=1)
         Buffers: shared hit=4 read=3
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
 (actual time=0.027..0.027 rows=100 loops=1)
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
 (actual time=0.070..0.070 rows=999 loops=1)
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=2 read=3
 Planning:
   Buffers: shared hit=3
 Planning Time: 0.162 ms
 Execution Time: 0.143 ms
```

Значения, которые выводятся с параметром `BUFFERS`, помогают понять, на какие части запроса приходится большинство операций ввода-вывода.

Не забывайте, что `EXPLAIN ANALYZE` действительно выполняет запрос, хотя его результаты могут не показываться, а заменяться выводом команды `EXPLAIN`. Поэтому при таком анализе возможны побочные эффекты. Если вы хотите проанализировать запрос, изменяющий данные, но при этом сохранить прежние данные таблицы, вы можете откатить транзакцию после запроса:

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                                QUERY PLAN
--------------------------------------------------------------------
--------------------------------------------------------------
 Update on tenk1  (cost=5.06..225.23 rows=0 width=0) (actual time=1.634..1.635 rows=0
 loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.06..225.23 rows=100 width=10) (actual
 time=0.065..0.141 rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         Heap Blocks: exact=90
```

```
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=100 width=0)
 (actual time=0.031..0.031 rows=100 loops=1)
               Index Cond: (unique1 < 100)
 Planning Time: 0.151 ms
 Execution Time: 1.856 ms

ROLLBACK;
```

Как показано в этом примере, когда выполняется команда `INSERT`, `UPDATE`, `DELETE` или `MERGE`, собственно изменение данных в таблице происходит в узле верхнего уровня Insert, Update, Delete или Merge. Узлы плана более низких уровней выполняют работу по нахождению старых строк и/или вычислению новых данных. Поэтому вверху мы видим тот же тип сканирования битовой карты, что и раньше, только теперь его вывод подаётся узлу Update, который сохраняет изменённые строки. Стоит отметить, что узел, изменяющий данные, может выполняться значительное время (в данном случае это составляет львиную часть всего времени), но планировщик не учитывает эту работу в оценке общей стоимости. Это связано с тем, что эта работа будет одинаковой при любом правильном плане запроса, и поэтому на выбор плана она не влияет.

Когда команда `UPDATE`, `DELETE` или `MERGE` имеет дело с секционированной таблицей или иерархией наследования, вывод может быть таким:

```
EXPLAIN UPDATE gtest_parent SET f1 = CURRENT_DATE WHERE f2 = 101;

                                       QUERY PLAN
-------------------------------------------------------------------
---------------------
 Update on gtest_parent  (cost=0.00..3.06 rows=0 width=0)
   Update on gtest_child gtest_parent_1
   Update on gtest_child2 gtest_parent_2
   Update on gtest_child3 gtest_parent_3
   ->  Append  (cost=0.00..3.06 rows=3 width=14)
         ->  Seq Scan on gtest_child gtest_parent_1  (cost=0.00..1.01 rows=1 width=14)
               Filter: (f2 = 101)
         ->  Seq Scan on gtest_child2 gtest_parent_2  (cost=0.00..1.01 rows=1 width=14)
               Filter: (f2 = 101)
         ->  Seq Scan on gtest_child3 gtest_parent_3  (cost=0.00..1.01 rows=1 width=14)
               Filter: (f2 = 101)
```

В этом примере узлу Update нужно обработать три дочерние таблицы, но не изначально упомянутую в запросе секционированную таблицу (поскольку она никогда не хранит данные). Поэтому формируются три плана сканирования, по одному для каждой таблицы. Ясности ради для узла Update добавляется примечание, показывающее, какие именно таблицы будут изменяться, в том же порядке, в каком они идут в соответствующих внутренних планах.

Под заголовком `Planning time` (Время планирования) команда `EXPLAIN ANALYZE` выводит время, затраченное на построение плана запроса из разобранного запроса и его оптимизацию. Время собственно разбора или перезаписи запроса в него не включается.

Значение `Execution time` (Время выполнения), выводимое командой `EXPLAIN ANALYZE`, включает продолжительность запуска и остановки исполнителя запроса, а также время выполнения всех сработавших триггеров, но не включает время разбора, перезаписи и планирования запроса. Время, потраченное на выполнение триггеров `BEFORE` (если такие имеются) включается во время соответствующих узлов Insert, Update или Delete node; но время выполнения триггеров `AFTER` не учитывается, так как триггеры `AFTER` срабатывают после выполнения всего плана. Общее время, проведённое в каждом триггере (`BEFORE` или `AFTER`), также выводится отдельно. Заметьте, что триггеры отложенных ограничений выполняются только в конце транзакции, так что время их выполнения `EXPLAIN ANALYZE` не учитывает.

Время, показанное для узла верхнего уровня, не включает время, необходимое для преобразования выходных данных запроса в отображаемую форму или отправки этих данных клиенту. При выполнении команды `EXPLAIN ANALYZE` данные никогда не отправляются клиенту, однако при помощи параметра `SERIALIZE` команде можно указывать, чтобы она преобразовывала выходные данные запроса в отображаемую форму и измеряла необходимое для этого время. Оно отображается отдельно и включается в общее время выполнения (значение `Execution Time`).

14.1.3. Ограничения

Время выполнения, измеренное командой `EXPLAIN ANALYZE`, может значительно отличаться от времени выполнения того же запроса в обычном режиме. Тому есть две основных причины. Во-первых, так как при анализе никакие строки результата не передаются клиенту, время передачи по сети не учитывается. Время ввода-вывода тоже не учитывается, за исключением случаев, когда указан параметр `SERIALIZE`. Во-вторых, может быть существенной дополнительная нагрузка, связанная с функциями измерений `EXPLAIN ANALYZE`, особенно в системах, где вызов `gettimeofday()` выполняется медленно. Для измерения этой нагрузки вы можете воспользоваться утилитой pg_test_timing.

Результаты `EXPLAIN` не следует распространять на ситуации, значительно отличающиеся от тех, в которых вы проводите тестирование. В частности, не следует полагать, что выводы, полученные для игрушечной таблицы, будут применимы и для настоящих больших таблиц. Оценки стоимости нелинейны и планировщик может выбирать разные планы в зависимости от размера таблицы. Например, в крайнем случае вся таблица может уместиться в одну страницу диска, и тогда вы почти наверняка получите план последовательного сканирования, независимо от того, есть у неё и индексы или нет. Планировщик понимает, что для обработки таблицы ему в любом случае потребуется прочитать одну страницу, так что нет никакого смысла обращаться к ещё одной странице за индексом. (Мы наблюдали это в показанном выше примере с `polygon_tbl`.)

Бывает, что фактическое и приближённо оценённое значения не совпадают, но в этом нет ничего плохого. Например, это возможно, когда выполнение плана узла прекращается преждевременно из-за указания `LIMIT` или подобного эффекта. Например, для запроса с `LIMIT`, который мы пробовали раньше:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                                            QUERY PLAN
--------------------------------------------------------------------
-------------------------------------------------------------
 Limit  (cost=0.29..14.33 rows=2 width=244) (actual time=0.051..0.071 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..70.50 rows=10 width=244)
 (actual time=0.051..0.070 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning Time: 0.077 ms
 Execution Time: 0.086 ms
```

Оценки стоимости и числа строк для узла Index Scan показываются в предположении, что этот узел будет выполняться до конца. Но в действительности узел Limit прекратил запрашивать строки, как только получил первые две, так что фактическое число строк равно 2 и время выполнения запроса будет меньше, чем рассчитал планировщик. Но это не ошибка, а просто следствие того, что оценённые и фактические значения выводятся по-разному.

Соединения слиянием также имеют свои особенности, которые могут ввести в заблуждение. Соединение слиянием прекратит читать один источник данных, если второй будет прочитан до конца, а следующее значение ключа в первом больше последнего значения во втором. В этом случае пар строк больше не будет, так что сканировать первый источник дальше нет смысла. В результате будут прочитаны не все строки одного потомка и вы получите тот же эффект, что и с `LIMIT`. Кроме того, если внешний (первый) потомок содержит строки с повторяющимися значениями ключа, внутренний (второй) потомок сдвинется назад и повторно выдаст строки для этого значения ключа. `EXPLAIN ANALYZE` считает эти повторяющиеся строки, как если бы это действительно были дополнительные строки внутреннего источника. Когда во внешнем узле много таких повторений ключей, фактическое число строк, подсчитанное для внутреннего узла, может значительно превышать число строк в соответствующей таблице.

Для узлов BitmapAnd (Логическое произведение битовых карт) и BitmapOr (Логическое сложение битовых карт) фактическое число строк всегда равно 0 из-за ограничений реализации.

Обычно `EXPLAIN` выводит подробности для каждого узла плана, сгенерированного планировщиком. Однако бывают ситуации, когда исполнитель может определить, что некоторые узлы не нужно выполнять, так как они не могут выдать никакие строки, с учётом значений параметров, ставших известными уже после планирования. (В настоящее время это может произойти только с дочерними узлами `Append` или `MergeAppend`, сканирующими секционированную таблицу.) В таких ситуациях эти узлы плана не попадают в вывод `EXPLAIN`, а в плане появляется запись `Subplans Removed: N` (Подпланов удалено: N).

14.2. Статистика, используемая планировщиком

14.2.1. Статистика по одному столбцу

Как было показано в предыдущем разделе, планировщик запросов должен оценить число строк, возвращаемых запросов, чтобы сделать правильный выбор в отношении плана запроса. В этом разделе кратко описывается статистика, которую использует система для этих оценок.

В частности, статистика включает общее число записей в каждой таблице и индексе, а также число дисковых блоков, которые они занимают. Эта информация содержится в таблице `pg_class`, в столбцах `reltuples` и `relpages`. Получить её можно, например так:

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      345
 tenk1_hundred        | i       |     10000 |       11
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Здесь мы видим, что `tenk1` содержит 10 000 строк данных и столько же строк в индексах, но объём индексов гораздо меньше таблицы (что неудивительно).

Для большей эффективности `reltuples` и `relpages` не пересчитываются «на лету», так что они обычно содержат несколько устаревшие значения. Их обновляют команды `VACUUM`, `ANALYZE` и несколько команд DDL, такие как `CREATE INDEX`. `VACUUM` и `ANALYZE` могут не сканировать всю таблицу (и обычно так и делают), а только вычислить приращение `reltuples` по части таблицы, так что результат остаётся приблизительным. В любом случае планировщик пересчитывает значения, полученные из `pg_class`, в пропорции к текущему физическому размеру таблицы и таким образом уточняет приближение.

Большинство запросов возвращают не все строки таблицы, а только немногие из них, ограниченные условиями `WHERE`. Поэтому планировщику нужно оценить *избирательность* условий `WHERE`, то есть определить, какой процент строк будет соответствовать каждому условию в предложении `WHERE`. Нужная для этого информация хранится в системном каталоге `pg_statistic`. Значения в `pg_statistic` обновляются командами `ANALYZE` и `VACUUM ANALYZE` и никогда не бывают точными, даже сразу после обновления.

Для исследования статистики лучше обращаться не непосредственно к таблице `pg_statistic`, а к представлению `pg_stats`, предназначенному для облегчения восприятия этой информации. Кроме того, представление `pg_stats` доступно для чтения всем, тогда как `pg_statistic` — только суперпользователям. (Это сделано для того, чтобы непривилегированные пользователи не могли ничего узнать о содержимом таблиц других людей из статистики. Представление `pg_stats` устроено так, что оно показывает строки только для тех таблиц, которые может читать данный пользователь.) Например, мы можем выполнить:

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+------------------------------------
 name    | f         | -0.5681108 | I- 580                        Ramp+
         |           |            | I- 880                        Ramp+
         |           |            | Sp Railroad                       +
         |           |            | I- 580                            +
         |           |            | I- 680                        Ramp+
         |           |            | I- 80                         Ramp+
         |           |            | 14th                          St  +
         |           |            | I- 880                            +
         |           |            | Mac Arthur                    Blvd+
         |           |            | Mission                       Blvd+
...
 name    | t         |    -0.5125 | I- 580                        Ramp+
         |           |            | I- 880                        Ramp+
         |           |            | I- 580                            +
         |           |            | I- 680                        Ramp+
         |           |            | I- 80                         Ramp+
         |           |            | Sp Railroad                       +
         |           |            | I- 880                            +
         |           |            | State Hwy 13                  Ramp+
         |           |            | I- 80                             +
         |           |            | State Hwy 24                  Ramp+
...
 thepath | f         |          0 |
 thepath | t         |          0 |
(4 rows)
```

Заметьте, что для одного столбца показываются две строки: одна соответствует полной иерархии наследования, построенной для таблицы `road` (`inherited=t`), и другая относится непосредственно к таблице `road` (`inherited=f`). (Для краткости в примере показаны только первые десять самых частых значений столбца `name`.)

Объём информации, сохраняемой в `pg_statistic` командой `ANALYZE`, в частности максимальное число записей в массивах `most_common_vals` (самые популярные значения) и `histogram_bounds` (границы гистограмм) для каждого столбца, можно ограничить на уровне столбцов с помощью команды `ALTER TABLE SET STATISTICS` или глобально, установив параметр конфигурации default_statistics_target. В настоящее время ограничение по умолчанию равно 100 записям. Увеличивая этот предел, можно повысить точность оценок планировщика, особенно для столбцов с нерегулярным распределением данных, ценой роста размера `pg_statistic` и незначительного увеличения времени расчёта этих оценок. И напротив, для столбцов с простым распределением данных может быть достаточно меньшего предела.

Подробнее использование статистики планировщиком описывается в Главе 65.

14.2.2. Расширенная статистика

Часто наблюдается картина, когда медленное выполнение запросов объясняется плохим выбором плана из-за того, что несколько столбцов, фигурирующих в условиях запроса, оказываются связанными. Обычно планировщик полагает, что несколько условий не зависят друг от друга, а это предположение оказывается неверным, когда значения этих столбцов коррелируют. Обычная статистика, которая по природе своей строится по отдельным столбцам, не может выявить корреляции между столбцами. Однако Postgres Pro имеет возможность вычислять *многовариантную статистику*, которая может собирать необходимую для этого информацию.

Так как число возможных комбинаций столбцов очень велико, автоматически вычислять многовариантную статистику непрактично. Вместо этого можно создать *объекты расширенной статистики*, чаще называемые просто *объектами статистики*, чтобы сервер собирал статистику по некоторым наборам столбцов, представляющим интерес.

Объекты статистики создаются командой `CREATE STATISTICS`. При создании такого объекта просто добавляется запись в каталоге, выражающая востребованность этой статистики. Собственно сбор данных выполняется процедурой `ANALYZE` (запускаемой вручную или автоматически в фоновом процессе). Изучить собранные значения можно в каталоге `pg_statistic_ext_data`.

Команда `ANALYZE` вычисляет расширенную статистику по той же выборке строк таблицы, которая используется и для вычисления обычной статистики по отдельным столбцам. Так как размер выборки увеличивается с увеличением целевого ограничения статистики для таблицы или любых её столбцов (как описано в предыдущем разделе), при большем целевом ограничении обычно получается более точная расширенная статистика, но и времени на её вычисление требуется больше.

В следующих подразделах описываются виды расширенной статистики, поддерживаемые в настоящее время.

14.2.2.1. Функциональные зависимости

Простейший вид расширенной статистики отслеживает *функциональные зависимости* (это понятие используется в определении нормальных форм баз данных). Мы называем столбец `b` функционально зависимым от столбца `a`, если знания значения `a` достаточно для определения значения `b`, то есть не существует двух строк с одинаковыми значениями `a`, но разными значениями `b`. В полностью нормализованной базе данных функциональные зависимости должны существовать только в первичных ключах и суперключах. Однако на практике многие наборы данных не нормализуются полностью по разным причинам; например, денормализация часто производится намеренно по соображениям производительности.

Существование функциональных зависимостей напрямую влияет на точность оценок в определённых запросах. Если запрос содержит условия как по независимым, так и по зависимым столбцам, условия по зависимым столбцам дополнительно не сокращают размер результата. Однако без знания о функциональной зависимости планировщик запросов будет полагать, что все условия независимы, и недооценит размер результата.

Для информирования планировщика о функциональных зависимостях команда `ANALYZE` может собирать показатели зависимостей между столбцами. Оценить степень зависимости между всеми наборами столбцов обошлось бы непозволительно дорого, поэтому сбор данных ограничивается только теми группами столбцов, которые фигурируют вместе в объекте статистики, определённом со свойством `dependencies`. Во избежание ненужных издержек при выполнении `ANALYZE` и последующем планировании запросов статистику с `dependencies` рекомендуется создавать только для групп сильно коррелирующих столбцов.

Взгляните на пример сбора статистики функциональной зависимости:

```
CREATE STATISTICS stts (dependencies) ON city, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxddependencies
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxddependencies
```

```
---------+---------+------------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

В показанном случае столбец 1 (код zip) полностью определяет столбец 5 (city), так что коэффициент равен 1.0, тогда как город (столбец city) определяет код ZIP только в 42% всех случаев, что означает, что многие города (58%) представлены несколькими кодами ZIP.

При вычислении избирательности запроса, в котором задействованы функционально зависимые столбцы, планировщик корректирует оценки избирательности по условиям, используя коэффициенты зависимостей, чтобы не допустить недооценки размера результата.

14.2.2.1.1. Ограничения функциональных зависимостей

Функциональные зависимости в настоящее время применяются только при рассмотрении простых условий с равенствами, сравнивающих значения столбцов с константами, и условиями `IN` с константами. Они не используются для улучшения оценок при проверке равенства двух столбцов или сравнении столбца с выражением, а также в условиях с диапазоном, условиях `LIKE` или любых других видах условий.

Рассматривая функциональные зависимости, планировщик предполагает, что условия по задействованным столбцам совместимы и таким образом избыточны. Если условия несовместимы, правильной оценкой должен быть ноль строк, но эта возможность не рассматривается. Например, с таким запросом

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

планировщик отбросит условие с `city`, так как оно не влияет на избирательность, что верно. Однако он сделает то же предположение и в таком случае:

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

хотя на самом деле этому запросу будет удовлетворять ноль строк. Но статистика функциональной зависимости не даёт достаточно информации, чтобы прийти к такому заключению.

Во многих практических ситуациях это предположение обычно удовлетворяется; например, графический интерфейс приложения для последующего формирования запроса может не допускать выбор несовместимого сочетания города и кода ZIP. Но когда это не так, статистика функциональной зависимости может не подойти.

14.2.2.2. Многовариантное число различных значений

Статистика по одному столбцу содержит число различных значений в каждом отдельном столбце. Оценки числа различных значений в сочетании нескольких столбцов (например, в `GROUP BY a, b`) часто оказываются ошибочными, когда планировщик имеет статистические данные только по отдельным столбцам, что приводит к выбору плохих планов.

Для улучшения таких оценок операция `ANALYZE` может собирать статистику по различным значениям для группы столбцов. Как и ранее, это непрактично делать для каждой возможной группы столбцов, так что данные собираются только по тем группам столбцов, которые указаны в определении объекта статистики, создаваемого со свойством `ndistinct`. Данные будут собираться по всем возможным сочетаниям из двух или нескольких столбцов из перечисленных в определении.

В продолжение предыдущего примера, количества различных значений в таблице ZIP-кодов могут выглядеть так:

```
CREATE STATISTICS stts2 (ndistinct) ON city, state, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxdndistinct AS nd
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts2';
-[ RECORD 1 ]--------------------------------------------------------
```

```
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

Как видно, есть три комбинации столбцов, имеющих 33178 различных значений: код ZIP и штат; код ZIP и город; код ZIP, город и штат (то, что все эти числа равны, ожидаемый факт, так как сам по себе код ZIP в этой таблице уникален). С другой стороны, сочетание города и штата даёт только 27435 различных значений.

Объект статистики `ndistinct` рекомендуется создавать только для тех сочетаний столбцов, которые действительно используются при группировке, и только когда неправильная оценка числа групп может привести к выбору плохих планов. В противном случае усилия, потраченные на выполнение `ANALYZE`, будут напрасными.

14.2.2.3. Многовариантные списки MCV

Ещё один тип статистики, сохраняемой для каждого столбца, представляют списки частых значений. Такие списки позволяют получать очень точную оценку для отдельных столбцов, но для запросов, содержащих условия с несколькими столбцами, полученная по ним оценка может быть значительно искажена.

Для улучшения таких оценок операция `ANALYZE` может собирать списки MCV по комбинациям столбцов. Подобно статистике функциональных зависимостей и различных значений, такую статистику непрактично собирать для каждой возможной группировки столбцов. В данном случае это ещё более актуально, так как списки MCV (в отличие от двух упомянутых статистик) содержат распространённые значения столбцов. Поэтому данные для них собираются только по тем группам столбцов, которые фигурируют в объекте статистики, определённом с указанием `mcv`.

В продолжение предыдущего примера, список MCV для таблицы ZIP-кодов может выглядеть следующим образом (в отличие от более простых типов статистики, для его анализа требуется применить функцию):

```
CREATE STATISTICS stts3 (mcv) ON city, state FROM zipcodes;

ANALYZE zipcodes;

SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts3';

 index |         values         | nulls | frequency | base_frequency
-------+------------------------+-------+-----------+----------------
     0 | {Washington, DC}       | {f,f} |  0.003467 |        2.7e-05
     1 | {Apo, AE}              | {f,f} |  0.003067 |        1.9e-05
     2 | {Houston, TX}          | {f,f} |  0.002167 |       0.000133
     3 | {El Paso, TX}          | {f,f} |     0.002 |       0.000113
     4 | {New York, NY}         | {f,f} |  0.001967 |       0.000114
     5 | {Atlanta, GA}          | {f,f} |  0.001633 |        3.3e-05
     6 | {Sacramento, CA}       | {f,f} |  0.001433 |        7.8e-05
     7 | {Miami, FL}            | {f,f} |    0.0014 |          6e-05
     8 | {Dallas, TX}           | {f,f} |  0.001367 |        8.8e-05
     9 | {Chicago, IL}          | {f,f} |  0.001333 |        5.1e-05
   ...
(99 rows)
```

Выводимая информация показывает, что наиболее распространённую комбинацию города и штата образует Washington и DC, с частотой около 0.35% (в объёме выборки). Базовая частота этой комбинации (вычисленная из частот значений в отдельных столбцах) составляет всего 0.0027%, то есть эта оценка оказывается заниженной на два порядка.

Объекты статистики MCV рекомендуется создавать только для тех сочетаний столбцов, которые действительно используются в условиях вместе, и только когда неправильная оценка числа групп может привести к выбору плохих планов. В противном случае усилия, потраченные на выполнение `ANALYZE` и планирование, будут напрасными.

14.3. Управление планировщиком с помощью явных предложений `JOIN`

Поведением планировщика в некоторой степени можно управлять, используя явный синтаксис `JOIN`. Понять, когда и почему это бывает нужно, поможет небольшое введение.

В простом запросе с соединением, например таком:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

планировщик может соединять данные таблицы в любом порядке. Например, он может разработать план, в котором сначала A соединяется с B по условию `WHERE a.id = b.id`, а затем C соединяется с получившейся таблицей по другому условию `WHERE`. Либо он может соединить B с C, а затем с A результатом соединения. Он также может соединить сначала A с C, а затем результат с B — но это будет не эффективно, так как ему придётся сформировать полное декартово произведение A и C из-за отсутствия в предложении `WHERE` условия, подходящего для оптимизации соединения. (В Postgres Pro исполнитель запросов может соединять только по две таблицы, поэтому для получения результата нужно выбрать один из этих способов.) При этом важно понимать, что все эти разные способы соединения дают одинаковые по смыслу результаты, но стоимость их может различаться многократно. Поэтому планировщик должен изучить их все и найти самый эффективный способ выполнения запроса.

Когда запрос включает только две или три таблицы, возможны всего несколько вариантов их соединения. Но их число растёт экспоненциально с увеличением числа задействованных таблиц. Если число таблиц больше десяти, уже практически невозможно выполнить полный перебор всех вариантов, и даже для шести или семи таблиц планирование может занять недопустимо много времени. Когда таблиц слишком много, планировщик Postgres Pro переключается с полного поиска на алгоритм *генетического* вероятностного поиска в ограниченном числе вариантов. (Порог для этого переключения задаётся параметром выполнения geqo_threshold.) Генетический поиск выполняется быстрее, но не гарантирует, что найденный план будет наилучшим.

Когда запрос включает внешние соединения, планировщик имеет меньше степеней свободы, чем с обычными (внутренними) соединениями. Например, рассмотрим запрос:

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Хотя ограничения в этом запросе очень похожи на показанные в предыдущем примере, смысл его отличается, так как результирующая строка должна выдаваться для каждой строки A, даже если для неё не находится соответствия в соединении B и C. Таким образом, здесь планировщик не может выбирать порядок соединения: он должен соединить B с C, а затем соединить A с результатом. Соответственно, и план этого запроса построится быстрее, чем предыдущего. В других случаях планировщик сможет определить, что можно безопасно выбрать один из нескольких способов соединения. Например, для запроса:

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

можно соединить A либо с B, либо с C. В настоящее время только `FULL JOIN` полностью ограничивает порядок соединения. На практике в большинстве запросов с `LEFT JOIN` и `RIGHT JOIN` порядком можно управлять в некоторой степени.

Синтаксис явного внутреннего соединения (`INNER JOIN`, `CROSS JOIN` или лаконичный `JOIN`) по смыслу равнозначен перечислению отношений в предложении `FROM`, так что он никак не ограничивает порядок соединений.

Хотя большинство видов `JOIN` не полностью ограничивают порядок соединения, в Postgres Pro можно принудить планировщик обрабатывать все предложения `JOIN` как ограничивающие этот порядок. Например, следующие три запроса логически равнозначны:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Но если мы укажем планировщику соблюдать порядок `JOIN`, на планирование второго и третьего уйдёт меньше времени. Когда речь идёт только о трёх таблицах, выигрыш будет незначительным, но для множества таблиц это может быть очень эффективно.

Чтобы планировщик соблюдал порядок внутреннего соединения, выраженный явно предложениями `JOIN`, нужно присвоить параметру выполнения join_collapse_limit значение 1. (Другие допустимые значения обсуждаются ниже.)

Чтобы сократить время поиска, необязательно полностью ограничивать порядок соединений, в `JOIN` можно соединять элементы как в обычном списке `FROM`. Например, рассмотрите следующий запрос:

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

Если `join_collapse_limit` = 1, планировщик будет вынужден соединить А с В раньше, чем результат с другими таблицами, но в дальнейшем выборе вариантов он не ограничен. В данном примере число возможных вариантов соединения уменьшается в 5 раз.

Упрощать для планировщика задачу перебора вариантов таким способом — это полезный приём, помогающий не только выбрать сократить время планирования, но и подтолкнуть планировщик к хорошему плану. Если планировщик по умолчанию выбирает неудачный порядок соединения, вы можете заставить его выбрать лучший, применив синтаксис `JOIN`, конечно если вы сами его знаете. Эффект подобной оптимизации рекомендуется подтверждать экспериментально.

На время планирования влияет и другой, тесно связанный фактор — решение о включении подзапросов в родительский запрос. Пример такого запроса:

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

Такая же ситуация может возникнуть с представлением, содержащим соединение; вместо ссылки на это представление будет вставлено его выражение `SELECT` и в результате получится запрос, похожий на показанный выше. Обычно планировщик старается включить подзапрос в родительский запрос и получить таким образом:

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

Часто это позволяет построить лучший план, чем при планировании подзапросов по отдельности. (Например, внешние условия `WHERE` могут быть таковы, что при соединении сначала X с A будет исключено множество строк A, а значит формировать логический результат подзапроса полностью не потребуется.) Но в то же время тем самым мы увеличиваем время планирования; две задачи соединения трёх элементов мы заменяем одной с пятью элементами. Так как число вариантов увеличивается экспоненциально, сложность задачи увеличивается многократно. Планировщик пытается избежать проблем поиска с огромным числом вариантов, рассматривая подзапросы отдельно, если в предложении `FROM` родительского запроса оказывается больше чем `from_collapse_limit` элементов. Изменяя этот параметр выполнения, можно подобрать оптимальное соотношение времени планирования и качества плана.

Параметры from_collapse_limit и join_collapse_limit называются похоже, потому что они делают практически одно и то же: первый параметр определяет, когда планировщик будет «сносить» в предложение `FROM` подзапросы, а второй — явные соединения. Обычно `join_collapse_limit` устанавливается равным `from_collapse_limit` (чтобы явные соединения и подзапросы обрабатывались одинаково) или 1 (если требуется управлять порядком соединений). Но вы можете задать другие значения, чтобы добиться оптимального соотношения времени планирования и времени выполнения запросов.

14.4. Наполнение базы данных

Довольно часто в начале или в процессе использования базы данных возникает необходимость загрузить в неё большой объём данных. В этом разделе приведены рекомендации, которые помогут сделать это максимально эффективно.

14.4.1. Отключите автофиксацию транзакций

Выполняя серию команд `INSERT`, выключите автофиксацию транзакций и зафиксируйте транзакцию только один раз в самом конце. (В обычном SQL это означает, что нужно выполнить `BEGIN` до, и `COMMIT` после этой серии. Некоторые клиентские библиотеки могут делать это автоматически, в таких случаях нужно убедиться, что это так.) Если вы будете фиксировать каждое добавление по отдельности, Postgres Pro придётся проделать много действий для каждой добавляемой строки. Выполнять все операции в одной транзакции хорошо ещё и потому, что в случае ошибки добавления одной из строк произойдёт откат к исходному состоянию и вы не окажетесь в сложной ситуации с частично загруженными данными.

14.4.2. Используйте `COPY`

Используйте `COPY`, чтобы загрузить все строки одной командой вместо серии `INSERT`. Команда `COPY` оптимизирована для загрузки большого количества строк; хотя она не так гибка, как `INSERT`, но при загрузке больших объёмов данных она влечёт гораздо меньше накладных расходов. Так как `COPY` — это одна команда, применяя её, нет необходимости отключать автофиксацию транзакций.

В случаях, когда `COPY` не подходит, может быть полезно создать подготовленный оператор `INSERT` с помощью `PREPARE`, а затем выполнять `EXECUTE` столько раз, сколько потребуется. Это позволит избежать накладных расходов, связанных с разбором и анализом каждой команды `INSERT`. В разных интерфейсах это может выглядеть по-разному; за подробностями обратитесь к описанию «подготовленных операторов» в документации конкретного интерфейса.

Заметьте, что с помощью `COPY` большое количество строк практически всегда загружается быстрее, чем с помощью `INSERT`, даже если используется `PREPARE` и серия операций добавления заключена в одну транзакцию.

`COPY` работает быстрее всего, если она выполняется в одной транзакции с командами `CREATE TABLE` или `TRUNCATE`. В таких случаях записывать WAL не нужно, так как в случае ошибки файлы, содержащие загружаемые данные, будут всё равно удалены. Однако это замечание справедливо, только когда параметр wal_level равен `minimal`, так как в противном случае все команды должны записывать свои изменения в WAL.

14.4.3. Удалите индексы

Если вы загружаете данные в только что созданную таблицу, быстрее всего будет загрузить данные с помощью `COPY`, а затем создать все необходимые для неё индексы. На создание индекса для уже существующих данных уйдёт меньше времени, чем на последовательное его обновление при добавлении каждой строки.

Если вы добавляете данные в существующую таблицу, может иметь смысл удалить индексы, загрузить таблицу, а затем пересоздать индексы. Конечно, при этом надо учитывать, что временное отсутствие индексов может отрицательно повлиять на скорость работы других пользователей. Кроме того, следует дважды подумать, прежде чем удалять уникальные индексы, так как без них соответствующие проверки ключей не будут выполняться.

14.4.4. Удалите ограничения внешних ключей

Как и с индексами, проверки, связанные с ограничениями внешних ключей, выгоднее выполнять «массово», а не для каждой строки в отдельности. Поэтому может быть полезно удалить ограничения внешних ключей, загрузить данные, а затем восстановить прежние ограничения. И в этом случае тоже приходится выбирать между скоростью загрузки данных и риском допустить ошибки в отсутствие ограничений.

Более того, когда вы загружаете данные в таблицу с существующими ограничениями внешнего ключа, для каждой новой строки добавляется запись в очередь событий триггера (так как именно срабатывающий триггер проверяет такие ограничения для строки). При загрузке многих миллионов строк очередь событий триггера может занять всю доступную память, что приведёт к недопустимой нагрузке на файл подкачки или даже к сбою команды. Таким образом, загружая большие объёмы данных, может быть не просто желательно, а *необходимо* удалять, а затем восстанавливать внешние ключи. Если же временное отключение этого ограничения неприемлемо, единственно возможным решением может быть разделение всей операции загрузки на меньшие транзакции.

14.4.5. Увеличьте `maintenance_work_mem`

Ускорить загрузку больших объёмов данных можно, увеличив параметр конфигурации maintenance_work_mem на время загрузки. Это приведёт к увеличению быстродействия `CREATE INDEX` и `ALTER TABLE ADD FOREIGN KEY`. На скорость самой команды `COPY` это не повлияет, так что этот совет будет полезен, только если вы применяете какой-либо из двух вышеописанных приёмов.

14.4.6. Увеличьте `max_wal_size`

Также массовую загрузку данных можно ускорить, изменив на время загрузки параметр конфигурации max_wal_size. Загружая большие объёмы данных, Postgres Pro вынужден увеличивать частоту контрольных точек по сравнению с обычной (которая задаётся параметром `checkpoint_timeout`), а значит и чаще сбрасывать «грязные» страницы на диск. Временно увеличив `max_wal_size`, можно уменьшить частоту контрольных точек и связанных с ними операций ввода-вывода.

14.4.7. Отключите архивацию WAL и потоковую репликацию

Для загрузки больших объёмов данных в среде, где используется архивация WAL или потоковая репликация, быстрее будет сделать копию базы данных после загрузки данных, чем обрабатывать множество операций изменений в WAL. Чтобы отключить передачу изменений через WAL в процессе загрузки, отключите архивацию и потоковую репликацию, назначьте параметру wal_level значение `minimal`, archive_mode — `off`, а max_wal_senders — 0. Но имейте в виду, что изменённые параметры вступят в силу только после перезапуска сервера, а все ранее созданные резервные копии не будут пригодны для резервного сервера и восстановления архива, что может привести к потере данных.

Это не только поможет сэкономить время архивации и передачи WAL, но и непосредственно ускорит некоторые команды, потому что они не записывают в WAL ничего, если в `wal_level` установлен уровень `minimal` и текущая подтранзакция (или транзакция верхнего уровня) создала и опустошила таблицу или индекс, куда затем вносятся изменения. (Они могут гарантировать безопасность данных при сбое, не записывая их в WAL, а только выполнив `fsync` в конце, что будет гораздо дешевле.)

14.4.8. Выполните в конце `ANALYZE`

Всякий раз, когда распределение данных в таблице значительно меняется, настоятельно рекомендуется выполнять `ANALYZE`. Эта рекомендация касается и загрузки в таблицу большого объёма данных. Выполнив `ANALYZE` (или `VACUUM ANALYZE`), вы тем самым обновите статистику по данной таблице для планировщика. Когда планировщик не имеет статистики или она не соответствует действительности, он не сможет правильно планировать запросы, что приведёт к снижению быстродействия при работе с соответствующими таблицами. Заметьте, что если включён демон автоочистки, он может запускать `ANALYZE` автоматически; подробнее об этом можно узнать в Подразделе 23.1.3 и Подразделе 23.1.6.

14.4.9. Несколько замечаний относительно pg_dump

В скриптах загрузки данных, которые генерирует pg_dump, автоматически учитываются некоторые, но не все из этих рекомендаций. Чтобы загрузить данные, которые выгрузил pg_dump, максимально быстро, вам нужно будет выполнить некоторые дополнительные действия вручную. (Заметьте, что эти замечания относятся только к *восстановлению* данных, но не к *выгрузке* их. Следующие рекомендации применимы вне зависимости от того, загружается ли архивный файл pg_dump в psql или в pg_restore.)

По умолчанию pg_dump использует команду `COPY` и когда она выгружает полностью схему и данные, в сгенерированном скрипте она сначала предусмотрительно загружает данные, а потом создаёт индексы и внешние ключи. Так что в этом случае часть рекомендаций выполняется автоматически. Вам остаётся учесть только следующие:

- Установите подходящие (то есть превышающие обычные) значения для `maintenance_work_mem` и `max_wal_size`.
- Если вы используете архивацию WAL или потоковую репликацию, по возможности отключите их на время восстановления. Для этого перед загрузкой данных, присвойте параметру `archive_mode` значение `off`, `wal_level` — `minimal`, а `max_wal_senders` — 0. Закончив восстановление, верните их обычные значения и сделайте свежую базовую резервную копию.
- Поэкспериментируйте с режимами параллельного копирования и восстановления команд pg_dump и pg_restore, и подберите оптимальное число параллельных заданий. Параллельное копирование и восстановление данных, управляемое параметром `-j`, должно дать значительный выигрыш в скорости по сравнению с последовательным режимом.
- Если это возможно в вашей ситуации, восстановите все данные в рамках одной транзакции. Для этого передайте параметр `-1` или `--single-transaction` команде psql или pg_restore. Но учтите, что в этом режиме даже незначительная ошибка приведёт к откату всех изменений и часы восстановления будут потрачены зря. В зависимости от того, насколько взаимосвязаны данные, предпочтительнее может быть вычистить их вручную. Команды `COPY` будут работать максимально быстро, когда они выполняются в одной транзакции и архивация WAL выключена.
- Если на сервере баз данных установлено несколько процессоров, полезным может оказаться параметр `--jobs` команды pg_restore. С его помощью можно выполнить загрузку данных и создание индексов параллельно.
- После загрузки данных запустите `ANALYZE`.

При выгрузке данных без схемы тоже используется команда `COPY`, но индексы, как обычно и внешние ключи, при этом не удаляются и не пересоздаются. [1] Поэтому, загружая только данные, вы сами должны решить, нужно ли для ускорения загрузки удалять и пересоздавать индексы и внешние ключи. При этом будет так же полезно увеличить параметр `max_wal_size`, но не `maintenance_work_mem`; его стоит менять, только если вы впоследствии пересоздаёте индексы и внешние ключи вручную. И не забудьте выполнить `ANALYZE` после; подробнее об этом можно узнать в Подразделе 23.1.3 и Подразделе 23.1.6.

14.5. Оптимизация, угрожающая стабильности

Стабильность — это свойство базы данных, гарантирующее, что результат зафиксированных транзакций будет сохранён даже в случае сбоя сервера или отключения питания. Однако обеспечивается стабильность за счёт значительной дополнительной нагрузки. Поэтому, если вы можете отказаться от такой гарантии, Postgres Pro можно ускорить ещё больше, применив следующие методы оптимизации. Кроме явно описанных исключений, даже с такими изменениями конфигурации при сбое программного ядра СУБД гарантия стабильности сохраняется; риск потери или разрушения данных возможен только в случае внезапного сбоя операционной системы.

- Поместите каталог данных кластера БД в файловую систему, размещённую в памяти (т. е. в RAM-диск). Так вы исключите всю активность ввода-вывода, связанную с базой данных, если только размер базы данных не превышает объём свободной памяти (возможно, с учётом файла подкачки).
- Выключите fsync; сбрасывать данные на диск не нужно.

---

[1] Вы можете отключить внешние ключи, используя параметр `--disable-triggers` — но при этом нужно понимать, что тем самым вы не просто отложите, а полностью выключите соответствующие проверки, что позволит вставить недопустимые данные.

- Выключите synchronous_commit; нет необходимости принудительно записывать WAL на диск при фиксации каждой транзакции. Но учтите, это может привести к потере транзакций (хотя данные останутся согласованными) в случае сбоя *базы данных*.
- Выключите full_page_writes; защита от частичной записи страниц не нужна.
- Увеличьте max_wal_size и checkpoint_timeout; это уменьшит частоту контрольных точек, хотя объём `/pg_wal` при этом вырастет.
- Создавайте нежурналируемые таблицы для оптимизации записи в WAL (но учтите, что такие таблицы не защищены от сбоя).

Глава 15. Параллельный запрос

Postgres Pro может вырабатывать такие планы запросов, которые будут задействовать несколько CPU, чтобы получить ответ на запросы быстрее. Эта возможность называется распараллеливанием запросов. Для многих запросов параллельное выполнение не даёт никакого выигрыша, либо из-за ограничений текущей реализации, либо из-за принципиальной невозможности построить параллельный план, который был бы быстрее последовательного. Однако для запросов, в которых это может быть полезно, распараллеливание часто даёт очень значительное ускорение. Многие такие запросы могут выполняться в параллельном режиме как минимум вдвое быстрее, а некоторые — быстрее в четыре и даже более раз. Обычно наибольший выигрыш можно получить с запросами, обрабатывающими большой объём данных, но возвращающими пользователю всего несколько строк. В этой главе достаточно подробно рассказывается, как работают параллельные запросы и в каких ситуациях их можно использовать, чтобы пользователи, желающие применять их, понимали, чего ожидать.

15.1. Как работают параллельно выполняемые запросы

Когда оптимизатор определяет, что параллельное выполнение будет наилучшей стратегией для конкретного запроса, он создаёт план запроса, включающий *узел Gather* (Сбор) или *Gather Merge* (Сбор со слиянием). Взгляните на простой пример:

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

Во всех случаях узел `Gather` или `Gather Merge` будет иметь ровно один дочерний план, представляющий часть общего плана, выполняемую в параллельном режиме. Если узел `Gather` или `Gather Merge` располагается на самом верху дерева плана, в параллельном режиме будет выполняться весь запрос. Если он находится где-то в другом месте плана, параллельно будет выполняться только часть плана ниже него. В приведённом выше примере запрос обращается только к одной таблице, так что помимо узла `Gather` есть только ещё один узел плана; и так как этот узел является потомком узла `Gather`, он будет выполняться в параллельном режиме.

Используя EXPLAIN, вы можете узнать количество исполнителей, выбранное планировщиком для данного запроса. Когда при выполнении запроса достигается узел `Gather`, процесс, обслуживающий сеанс пользователя, запрашивает фоновые рабочие процессы в этом количестве. Количество исполнителей, которое может попытаться задействовать планировщик, ограничивается значением max_parallel_workers_per_gather. Общее число фоновых рабочих процессов, которые могут существовать одновременно, ограничивается параметрами max_worker_processes и max_parallel_workers. Таким образом, вполне возможно, что параллельный запрос будет выполняться меньшим числом рабочих процессов, чем планировалось, либо вообще без дополнительных рабочих процессов. Оптимальность плана может зависеть от числа доступных рабочих процессов, так что их нехватка может повлечь значительное снижение производительности. Если это наблюдается часто, имеет смысл увеличить `max_worker_processes` и `max_parallel_workers`, чтобы одновременно могло работать больше процессов, либо наоборот уменьшить `max_parallel_workers_per_gather`, чтобы планировщик запрашивал их в меньшем количестве.

Каждый фоновый рабочий процесс, успешно запущенный для данного параллельного запроса, будет выполнять параллельную часть плана. Ведущий процесс также будет выполнять эту часть плана, но он несёт дополнительную ответственность: он должен также прочитать все кортежи, выданные рабочими процессами. Когда параллельная часть плана выдаёт лишь небольшое количество кортежей, ведущий часто ведёт себя просто как один из рабочих процессов, ускоряя выполнение запроса. И напротив, когда параллельная часть плана выдаёт множество кортежей, ведущий может быть почти всё время занят чтением кортежей, выдаваемых другими рабочими процессами, и выполнять другие шаги обработки, связанные с узлами плана выше узла `Gather` или `Gather Merge`. В таких случаях ведущий процесс может вносить лишь минимальный вклад в выполнение параллельной части плана.

Когда над параллельной частью плана оказывается узел `Gather Merge`, а не `Gather`, это означает, что все процессы, выполняющие части параллельного плана, выдают кортежи в отсортированном порядке, и что ведущий процесс выполняет слияние с сохранением порядка. Узел же `Gather`, напротив, получает кортежи от подчинённых процессов в произвольном удобном ему порядке, нарушая порядок сортировки, который мог существовать.

15.2. Когда может применяться распараллеливание запросов?

Планировщик запросов может отказаться от построения параллельных планов запросов в любом случае под влиянием нескольких параметров. Чтобы он строил параллельные планы запросов при каких-бы то ни было условиях, описанные далее параметры необходимо настроить указанным образом.

- max_parallel_workers_per_gather должен иметь значение, большее нуля. Это особый вариант более общего ограничения на суммарное число используемых рабочих процессов, задаваемого параметром `max_parallel_workers_per_gather`.

В дополнение к этому, система должна работать не в однопользовательском режиме. Так как в этом режиме вся СУБД работает как один процесс, фоновые рабочие процессы в нём недоступны.

Даже если принципиально возможно построить параллельные планы выполнения, планировщик не будет строить такой план для определённого запроса, если имеет место одно из следующих обстоятельств:

- Запрос выполняет запись данных или блокирует строки в базе данных. Если запрос содержит операцию, изменяющую данные либо на верхнем уровне, либо внутри CTE, для такого запроса не будут строиться параллельные планы. Исключение составляют следующие команды, которые создают новую таблицу, наполняют её и при этом могут использовать параллельный план для команды `SELECT`, выполняемой внутри запроса:
  - `CREATE TABLE ... AS`
  - `SELECT INTO`
  - `CREATE MATERIALIZED VIEW`
  - `REFRESH MATERIALIZED VIEW`
- Запрос может быть приостановлен в процессе выполнения. В ситуациях, когда система решает, что может иметь место частичное или дополнительное выполнение, план параллельного выполнения не строится. Например, курсор, созданный предложением DECLARE CURSOR, никогда не будет использовать параллельный план. Подобным образом, цикл PL/pgSQL вида `FOR x IN query LOOP .. END LOOP` никогда не будет использовать параллельный план, так как система параллельных запросов не сможет определить, может ли безопасно выполняться код внутри цикла во время параллельного выполнения запроса.
- В запросе используются функции, помеченные как `PARALLEL UNSAFE`. Большинство системных функций безопасны для параллельного выполнения (`PARALLEL SAFE`), но пользовательские функции по умолчанию помечаются как небезопасные (`PARALLEL UNSAFE`). Эта характеристика функции рассматривается в Разделе 15.4.
- Запрос работает внутри другого запроса, уже параллельного. Например, если функция, вызываемая в параллельном запросе, сама выполняет SQL-запрос, последний запрос никогда не

будет выполняться параллельно. Это ограничение текущей реализации, но убирать его вряд ли следует, так как это может привести к использованию одним запросом чрезмерного количества процессов.

Даже когда для определённого запроса построен параллельный план, при некоторых условиях этот план нельзя будет выполнить в параллельном режиме. В этих случаях ведущий процесс выполнит часть плана ниже узла `Gather` самостоятельно, как если бы узла `Gather` вовсе не было. Это произойдёт только при выполнении одного из следующих условий:

- Невозможно получить ни одного фонового рабочего процесса из-за ограничения общего числа этих процессов значением max_worker_processes.
- Невозможно получить ни одного фонового рабочего процесса из-за ограничения общего числа таких процессов для параллельного выполнения значением max_parallel_workers.
- Клиент передаёт сообщение Execute с ненулевым количеством выбираемых кортежей. За подробностями обратитесь к описанию расширенного протокола запросов. Так как libpq в настоящее время не позволяет передавать такие сообщения, это возможно только с клиентом, задействующим не libpq. Если это происходит часто, имеет смысл установить в max_parallel_workers_per_gather 0 в сеансах, для которых это актуально, чтобы система не пыталась строить планы, которые могут быть неэффективны при последовательном выполнении.

15.3. Параллельные планы

Так как каждый рабочий процесс выполняет параллельную часть плана до конца, нельзя просто взять обычный план запроса и запустить его в нескольких исполнителях. В этом случае все исполнители выдавали бы полные копии выходного набора результатов, так что запрос выполнится не быстрее, чем обычно, а его результаты могут быть некорректными. Вместо этого параллельной частью плана должно быть то, что для оптимизатора представляется как *частичный план*; то есть такой план, при выполнении которого в отдельном процессе будет получено только подмножество выходных строк, а каждая требующаяся строка результата будет гарантированно выдана ровно одним из сотрудничающих процессов. Вообще говоря, это означает, что сканирование нижележащей таблицы запроса должно проводиться с учётом распараллеливания.

15.3.1. Параллельные сканирования

В настоящее время поддерживаются следующие виды сканирований таблицы, рассчитанные на параллельное выполнение.

- При *параллельном последовательном сканировании* блоки таблицы будут разделены на диапазоны, распределяемые между взаимодействующими процессами. Каждый рабочий процесс сначала завершает сканирование полученного диапазона блоков, а затем запрашивает следующий диапазон.
- При *параллельном сканировании кучи по битовой карте* один процесс выбирается на роль ведущего. Этот процесс производит сканирование одного или нескольких индексов и строит битовую карту, показывающую, какие блоки таблицы нужно посетить. Затем эти блоки разделяются между взаимодействующими процессами как при параллельном последовательном сканировании. Другими словами, сканирование кучи выполняется в параллельном режиме, а сканирование нижележащего индекса — нет.
- При *параллельном сканировании по индексу* или *параллельном сканировании только индекса* взаимодействующие процессы читают данные из индекса по очереди. В настоящее время параллельное сканирование индекса поддерживается только для индексов-B-деревьев. Каждый процесс будет выбирать один блок индекса с тем, чтобы просканировать и вернуть все кортежи, на которые он ссылается; другие процессы могут в то же время возвращать кортежи для другого блока индекса. Результаты параллельного сканирования B-дерева каждый рабочий процесс возвращает в отсортированном порядке.

В будущем может появиться поддержка параллельного выполнения и для других вариантов сканирования, например, сканирования индексов, отличных от B-дерева.

15.3.2. Параллельные соединения

Как и в непараллельном плане, целевая таблица может соединяться с одной или несколькими другими таблицами с использованием вложенных циклов, соединения по хешу или соединения слиянием. Внутренней стороной соединения может быть любой вид непараллельного плана, который в остальном поддерживается планировщиком, при условии, что он безопасен для выполнения в параллельном исполнителе. Внутренней стороной может быть и параллельный план, в зависимости от типа соединения.

- В *соединении с вложенным циклом* внутренняя сторона всегда непараллельная. Хотя она выполняется полностью, это эффективно, если с внутренней стороны производится сканирование индекса, так как внешние кортежи, а значит и циклы, находящие значения в индексе, разделяются по параллельным процессам.
- При *соединении слиянием* с внутренней стороны всегда будет непараллельный план и, таким образом, он будет выполняться полностью. Это может быть неэффективно, особенно если потребуется произвести сортировку, так как работа и конечные данные будут повторяться в каждом параллельном процессе.
- При *соединении по хешу* (непараллельном, без префикса «parallel») внутреннее соединение выполняется полностью в каждом параллельном процессе, и в результате они строят одинаковые копии хеш-таблицы. Это может быть неэффективно при большой хеш-таблице или дорогостоящем плане. В *параллельном соединении по хешу* с внутренней стороны выполняется *параллельное хеширование*, при котором работа по построению общей хеш-таблицы разделяется между параллельными процессами.

15.3.3. Параллельное агрегирование

Postgres Pro поддерживает параллельное агрегирование, выполняя агрегирование в два этапа. Сначала каждый процесс, задействованный в параллельной части запроса, выполняет шаг агрегирования, выдавая частичный результат для каждой известной ему группы. В плане это отражает узел `Partial Aggregate`. Затем эти промежуточные результаты передаются ведущему через узел `Gather` или `Gather Merge`. И наконец, ведущий заново агрегирует результаты всех рабочих процессов, чтобы получить окончательный результат. Это отражает в плане узел `Finalize Aggregate`.

Так как узел `Finalize Aggregate` выполняется в ведущем процессе, запросы, выдающие достаточно большое количество групп по отношению к числу входных строк, будут расцениваться планировщиком как менее предпочтительные. Например, в худшем случае количество групп, выявленных узлом `Finalize Aggregate`, может равняться числу входных строк, обработанных всеми рабочими процессами на этапе `Partial Aggregate`. Очевидно, что в такой ситуации использование параллельного агрегирования не даст никакого выигрыша производительности. Планировщик запросов учитывает это в процессе планирования, так что выбор параллельного агрегирования в подобных случаях очень маловероятен.

Параллельное агрегирование поддерживается не во всех случаях. Чтобы оно поддерживалось, агрегатная функция должна быть безопасной для распараллеливания и должна иметь комбинирующую функцию. Если переходное состояние агрегатной функции имеет тип `internal`, она должна также иметь функции сериализации и десериализации. За подробностями обратитесь к CREATE AGGREGATE. Параллельное агрегирование не поддерживается, если вызов агрегатной функции содержит предложение `DISTINCT` или `ORDER BY`. Также оно не поддерживается для сортирующих агрегатов или когда запрос включает предложение `GROUPING SETS`. Оно может использоваться только когда все соединения, задействованные в запросе, также входят в параллельную часть плана.

15.3.4. Параллельное присоединение

Когда требуется объединить строки из различных источников в единый набор результатов, в Postgres Pro используются узлы плана `Append` или `MergeAppend`. Это обычно происходит при реализации `UNION ALL` или при сканировании секционированной таблицы. Данные узлы могут применяться как в параллельных, так и в обычных планах. Однако в параллельных планах планировщик может заменить их на узел `Parallel Append`.

Если в параллельном плане используется узел `Append`, все задействованные процессы выполняют очередной дочерний план совместно, пока он не будет завершён, и лишь затем, примерно в одно время, переходят к выполнению следующего дочернего плана. Когда же применяется `Parallel Append`, исполнитель старается равномерно распределить между задействованными процессами все дочерние планы, чтобы они выполнялись параллельно. Это позволяет избежать конкуренции и не тратить ресурсы на запуск дочернего плана для тех процессов, которые не будут его выполнять.

Кроме того, в отличие от обычного узла `Append`, использование которого внутри параллельного плана допускается только для частичных дочерних планов, узел `Parallel Append` может обрабатывать как частичные, так и не частичные дочерние планы. Для сканирования не частичного плана будет использоваться только один процесс, поскольку его многократное сканирование приведёт лишь к дублированию результатов. Таким образом, для планов, объединяющих несколько наборов результатов, можно достичь параллельного выполнения на высоком уровне, даже когда эффективные частичные планы отсутствуют. Например, рассмотрим запрос к секционированной таблице, который может быть эффективно реализован только с помощью индекса, не поддерживающего параллельное сканирование. Планировщик может выбрать узел `Parallel Append` для параллельного объединения нескольких обычных планов `Index Scan`; в этом случае каждое сканирование индекса будет выполняться до полного завершения одним процессом, но при этом разные сканирования будут осуществляться параллельно.

Отключить данную функциональность можно с помощью enable_parallel_append.

15.3.5. Советы по параллельным планам

Если для запроса ожидается параллельный план, но такой план не строится, можно попытаться уменьшить parallel_setup_cost или parallel_tuple_cost. Разумеется, этот план может оказаться медленнее последовательного плана, предпочитаемого планировщиком, но не всегда. Если вы не получаете параллельный план даже с очень маленькими значениями этих параметров (например, сбросив оба их в ноль), может быть какая-то веская причина тому, что планировщик запросов не может построить параллельный план для вашего запроса. За информацией о возможных причинах обратитесь к Разделу 15.2 и Разделу 15.4.

Когда выполняется параллельный план, вы можете применить `EXPLAIN (ANALYZE, VERBOSE)`, чтобы просмотреть статистику по каждому узлу плана в разрезе рабочих процессов. Это может помочь определить, равномерно ли распределяется работа между всеми узлами плана, и на более общем уровне понимать характеристики производительности плана.

15.4. Безопасность распараллеливания

Планировщик классифицирует операции, вовлечённые в выполнение запроса, как либо *безопасные для распараллеливания*, либо *ограниченно распараллеливаемые*, либо *небезопасные для распараллеливания*. Безопасной для распараллеливания операцией считается такая, которая не мешает параллельному выполнению запроса. Ограниченно распараллеливаемой операцией считается такая, которая не может выполняться в параллельном рабочем процессе, но может выполняться в ведущем процессе, когда запрос выполняется параллельно. Таким образом, ограниченно параллельные операции никогда не могут оказаться ниже узла `Gather` или `Gather Merge`, но могут встречаться в других местах плана, содержащего такой узел. Небезопасные для распараллеливания операции не могут выполняться в параллельных запросах, даже в ведущем процессе. Когда запрос содержит что-либо небезопасное для распараллеливания, параллельное выполнение для такого запроса полностью исключается.

Ограниченно распараллеливаемыми всегда считаются следующие операции:

- Сканирование общих табличных выражений (CTE).
- Сканирование временных таблиц.
- Сканирование сторонних таблиц, если только обёртка сторонних данных не предоставляет функцию `IsForeignScanParallelSafe`, которая допускает распараллеливание.
- Узлы плана, которые ссылаются на связанный `SubPlan`.

15.4.1. Пометки параллельности для функций и агрегатов

Планировщик не может автоматически определить, является ли пользовательская обычная или агрегатная функция безопасной для распараллеливания, так как это потребовало бы предсказания действия каждой операции, которую могла бы выполнять функция. В общем случае это равнозначно решению проблемы остановки, а значит, невозможно. Даже для простых функций, где это в принципе возможно, мы не пытаемся это делать, так как это будет слишком дорогой и потенциально неточной процедурой. Вместо этого, все определяемые пользователем функции полагаются небезопасными для распараллеливания, если явно не отмечено обратное. Когда используется CREATE FUNCTION или ALTER FUNCTION, функции можно назначить отметку `PARALLEL SAFE`, `PARALLEL RESTRICTED` или `PARALLEL UNSAFE`, отражающую её характер. В команде CREATE AGGREGATE для параметра `PARALLEL` можно задать `SAFE`, `RESTRICTED` или `UNSAFE` в виде соответствующего значения.

Обычные и агрегатные функции должны помечаться небезопасными для распараллеливания (`PARALLEL UNSAFE`), если они пишут в базу данных, изменяют состояние транзакции (кроме использования подтранзакций для восстановления после сбоя), обращаются к последовательностям или производят постоянные изменения параметров. Подобным образом, функции должны помечаться как ограниченно распараллеливаемые (`PARALLEL RESTRICTED`), если они обращаются к временным таблицам, состоянию клиентского подключения, курсорам, подготовленным операторам или разнообразному локальному состоянию обслуживающего процесса, которое система не может синхронизировать между рабочими процессами. Например, по этой причине ограниченно параллельными являются функции `setseed` и `random`.

В целом, если функция помечена как безопасная, когда на самом деле она небезопасна или ограниченно безопасна, или если она помечена как ограниченно безопасная, когда на самом деле она небезопасная, такая функция может выдавать ошибки или возвращать неправильные ответы при использовании в параллельном запросе. Функции на языке C могут теоретически проявлять полностью неопределённое появление при некорректной пометке, так как система никаким образом не может защитить себя от произвольного кода C, но чаще всего результат будет не хуже, чем с любой другой функцией. В случае сомнений, вероятно, лучше всего будет помечать функции как небезопасные (`UNSAFE`).

Если функция, выполняемая в параллельном рабочем процессе, затребует блокировки, которыми не владеет ведущий, например, обращаясь к таблице, не упомянутой в запросе, эти блокировки будут освобождены по завершении процесса, а не в конце транзакции. Если вы разрабатываете функцию с таким поведением, и эта особенность выполнения оказывается критичной, пометьте такую функцию как `PARALLEL RESTRICTED`, чтобы она выполнялась только в ведущем процессе.

Заметьте, что планировщик запросов не рассматривает возможность отложенного выполнения ограниченно распараллеливаемых обычных или агрегатных функций, задействованных в запросе, для получения лучшего плана. Поэтому, например, если предложение `WHERE`, применяемое к конкретной таблице, является ограниченно параллельным, планировщик запросов исключит возможность сканирования этой таблицы в параллельной части плана. В некоторых случаях возможно (и, вероятно, более эффективно) включить сканирование этой таблицы в параллельную часть запроса и отложить вычисление предложения `WHERE`, чтобы оно происходило над узлом `Gather`, но планировщик этого не делает.

Часть III. Администрирование сервера

В этой части документации освещаются темы, представляющие интерес для администратора баз данных Postgres Pro. В частности, здесь рассматривается установка и настройка сервера, управление пользователями и базами данных, а также задачи обслуживания. С этими темами следует ознакомиться всем, кто эксплуатирует сервер Postgres Pro (даже для личных целей, а тем более в производственной среде).

Материал даётся примерно в том порядке, в каком его следует читать начинающему пользователю. Главы самодостаточны и при желании могут быть прочитаны по отдельности. Информация представлена в повествовательном стиле и разделена по темам. Если же вас интересует формальное и полное описание команд, рекомендуем обратиться к Части VI.

Первые несколько глав написаны так, чтобы их можно было понять без предварительных знаний, так что начинающие пользователи, которым нужно установить собственный сервер, могут начать свой путь с них. Остальные главы части посвящены настройке сервера и управлению им; в этом материале подразумевается, что читатель знаком с основными принципами использования СУБД Postgres Pro. За дополнительной информацией мы рекомендуем обратиться к Части I и Части II.

Глава 16. Установка из двоичных пакетов

16.1. Установка Postgres Pro Standard

Postgres Pro работает только в ОС на базе Linux и поставляется в виде двоичных пакетов. Все двоичные дистрибутивы Postgres Pro состоят из нескольких пакетов. Структура пакетов отличается от структуры ванильного PostgreSQL и поддерживает следующие режимы установки:

- **Быстрая установка и настройка**. Пакет `postgrespro-std-17` устанавливает и настраивает все компоненты, необходимые для получения готовой к использованию конфигурации, как клиентских, так и серверных компонентов. Выберите этот вариант, если вы планируете установить только один экземпляр Postgres Pro и вас не беспокоят возможные конфликты с другими продуктами на базе PostgreSQL.

  Важно

  При установке пакета `postgrespro-std-17` могут быть удалены существующие инсталляции продуктов на базе Postgres Pro и PostgreSQL. И напротив, установленный пакет Postgres Pro может быть автоматически удалён, если вы впоследствии развернёте другой продукт на базе PostgreSQL. Данный пакет не нужно устанавливать в процессе обновления или миграции, а также если вы планируете разворачивать в одной системе несколько инсталляций.

- **Расширенная установка**. Вы можете выбрать любые пакеты, требующиеся для ваших целей, включая пакеты для разработчиков. Этот вариант требует ручной настройки, поэтому для него необходимо хорошо разбираться в Linux и понимать архитектуру PostgreSQL. Это единственно возможный вариант, если вы планируете реализовать один из следующих сценариев использования Postgres Pro:
  - Установка одновременно нескольких версий Postgres Pro либо установка вместе с другими продуктами на базе PostgreSQL.
  - Обновление с другой версии или миграция с другого продукта на базе PostgreSQL.
  - Управление сервером Postgres Pro с использованием программных средств обеспечения высокой степени доступности, таких как pacemaker, вместо стандартных системных механизмов управления службами.

Минимальные системные требования для установки Postgres Pro, создания кластера баз данных и запуска сервера:

- Процессор с частотой 1 ГГц
- 1 ГБ ОЗУ
- 1 ГБ свободного дискового пространства

Для вспомогательных компонентов или данных потребуется дополнительное дисковое пространство.

16.1.1. Поддерживаемые дистрибутивы Linux

Примечание

Это последний выпуск Postgres Pro Standard, в котором поддерживается Ubuntu 24.10. Со следующего выпуска поддержка этой операционной системы прекратится.

Примечание

Следующий выпуск Postgres Pro Standard будет последним с поддержкой Ubuntu 25.04.

Примечание

Следующий выпуск будет последним, в котором сертифицированная редакция Postgres Pro Standard поддерживает CentOS 8 и RHEL 8.

Двоичные пакеты Postgres Pro предоставляются для следующих систем на базе Linux:

- Системы Red Hat Enterprise Linux (RHEL) и производные от них: CentOS 8, Rocky Linux 8/9/10, Red Hat Enterprise Linux 8/9/10, Oracle Linux 8/9/10, РОСА «ХРОМ» 2021.2, РЕД ОС МУРОМ 7.3/8, AlmaLinux 9/10
- Системы на базе Debian: Debian 11/12/13, Ubuntu 22.04/24.04/24.10/25.04, Astra Linux 1.7/1.8/3.8
- Альт 10/11, Альт СП релиз 10
- SUSE Linux Enterprise Server (SLES) 15
- МосОС 15
- Системы Linux для архитектуры процессоров Эльбрус: Альт 9/10 для e2kv3/e2kv4, Альт 8.2 СП для e2kv3/e2kv4, Альт СП релиз 10, Astra Linux «Ленинград» 8.1
- Системы Linux для архитектуры ARM: Альт 9/10/11, Альт СП 8.2 и выше, Astra Linux «Новороссийск», Debian 11/12/13, РЕД ОС МУРОМ 7.3/8, RHEL 8/9/10, Ubuntu 22.04/24.04

Примечание

Двоичные пакеты Postgres Pro используют библиотеку tzdata, предоставляемую операционной системой, поэтому важно, чтобы в системе была установлена последняя её версия. В случае использования устаревшей библиотеки tzdata значения времени в вашей базе данных могут быть некорректными.

16.1.2. Подключение репозитория пакетов

Перед установкой Postgres Pro из двоичных пакетов (с помощью менеджера пакетов вашей операционной системы) необходимо подключить репозиторий пакетов Postgres Pro.

Для подключения репозитория выполните следующие действия:

1. Обратитесь к специалистам технической поддержки Postgres Pro Standard, чтобы загрузить Bash-скрипт `pgpro-repo-add.sh`. Вам также предоставят реквизиты для доступа к репозиторию.

   Этот скрипт подключит репозиторий, чтобы менеджер пакетов вашей операционной системы мог использовать его.

2. Убедитесь, что утилита wget установлена и имеет доступ в Интернет.

   Скрипту необходима утилита wget и доступ в Интернет, чтобы проверить корректность реквизитов в ходе подключения репозитория.

   ### Примечание

   Для доступа в Интернет невозможно использовать SOCKS5-прокси, поскольку в wget отсутствует поддержка прокси этого типа.

3. (Необязательно) При необходимости проверьте содержимое файла со скриптом.
4. Запустите скрипт от имени пользователя `root` (например, используя sudo).
5. Введите реквизиты для доступа к репозиторию, когда скрипт их запросит.

> **Важно**
>
> Если реквизиты не были предоставлены или они некорректны, скрипт предложит подключить репозиторий без указания реквизитов.
>
> Если вы решите подключить репозиторий без указания реквизитов, то, чтобы получить доступ к репозиторию, вам потребуется указать реквизиты вручную.

Теперь все готово к установке Postgres Pro Standard с помощью менеджера пакетов вашей операционной системы.

16.1.3. Быстрая установка и настройка

Если вам нужно установить только один экземпляр Postgres Pro и вы не собираетесь использовать никакие другие продукты на базе PostgreSQL в вашей системе, вы можете использовать режим быстрой установки. Типичная процедура установки в этом случае выглядит так:

1. Подключите репозиторий пакетов, предназначенный для вашей операционной системы.
2. Установите пакет `postgrespro-std-17`. При этом по зависимостям установятся все требуемые компоненты, будет создана база данных по умолчанию, запущен сервер баз данных и настроен автозапуск сервера при загрузке системы, а все предоставляемые программы станут доступными в пути `PATH`.

> **Примечание**
>
> По умолчанию конфигурация базы данных настраивается под устанавливаемый продукт Postgres Pro. Если вам нужна другая настройка сервера базы данных, выберите расширенную установку и воспользуйтесь аргументом `tune` команды `pg-setup initdb`.

После завершения установки вы можете запустить psql от имени пользователя `postgres` и подключиться к только что созданной базе данных, находящейся в каталоге данных `/var/lib/pgpro/std-17/data`.

Так как база данных по умолчанию создаётся скриптом pg-setup, путь к каталогу данных сохраняется в файле `/etc/default/postgrespro-std-17`. Все последующие команды pg-setup, а также любые команды, управляющие службой Postgres Pro, будут нацелены именно на эту базу данных. Кроме того, в указанном файле можно изменить значение `PG_OOM_ADJUST_VALUE` для дочерних процессов postmaster (за подробностями обратитесь к Подразделу 17.4.4).

16.1.4. Расширенная установка

Разделение дистрибутива на несколько пакетов позволяет произвести инсталляцию по-разному для различных применений: для серверов баз данных, клиентских систем или рабочих станций разработчиков. Такие инсталляции необходимо настраивать вручную, но это даёт дополнительную гибкость при использовании продукта. Вы можете установить несколько версий Postgres Pro параллельно, а также вместе с другими продуктами PostgreSQL. В частности, это может потребоваться при осуществлении обновления или при миграции с другого продукта на базе PostgreSQL.

Для осуществления расширенной установки выполните следующие действия:

1. Подключите репозиторий пакетов, предназначенный для вашей операционной системы.
2. Выберите пакеты Postgres Pro, требующиеся для ваших целей, и установите их с помощью стандартных для вашего дистрибутива Linux команд. Доступные пакеты перечислены в Таблице 16.1.

   В результате все файлы будут установлены в каталог `/opt/pgpro/std-17`.
3. Запустите от имени root утилиту pg-wrapper, чтобы добавить и клиентские, и серверные установленные программы в путь поиска `PATH`, а также включить страницы man по SQL в файл конфигурации страниц man. Эта утилита входит в состав пакета `postgrespro-std-17-client`.

   ```
   /opt/pgpro/std-17/bin/pg-wrapper links update
   ```

   Подробнее о разрешении возможных конфликтов рассказывается в pg-wrapper.
4. Если вы устанавливаете пакет `postgrespro-std-17-server`, выполните следующую процедуру настройки сервера:

   a. Создайте начальную базу данных, запустив вспомогательный скрипт pg-setup от имени root с ключом `initdb`:

   ```
   /opt/pgpro/std-17/bin/pg-setup initdb [параметры_initdb]
   ```

   Здесь:

   - Аргумент `tune` выбирает вариант конфигурации базы данных.
   - *`параметры_initdb`* — обычные параметры initdb.

   ### Примечание

   По умолчанию pg-setup инициализирует кластер баз данных со включёнными контрольными суммами. Если для вас это нежелательно, укажите `--no-data-checksums`.

   Скрипт `pg-setup` выполняет административные операции с базами данных от имени пользователя `postgres`. Если вы не укажете никакие параметры initdb, база по умолчанию создаётся в каталоге `/var/lib/pgpro/std-17/data` с параметрами локализации, определяемыми переменной окружения `LANG` текущего сеанса. Переменные окружения `LC_*` при этом игнорируются.

   Так как база данных по умолчанию создаётся скриптом pg-setup, путь к каталогу данных сохраняется в файле `/etc/default/postgrespro-std-17`. Все последующие команды pg-setup, а также любые команды, управляющие службой Postgres Pro, будут нацелены именно на эту базу данных. Кроме того, в указанном файле можно изменить значение `PG_OOM_ADJUST_VALUE` для дочерних процессов postmaster (за подробностями обратитесь к Подразделу 17.4.4).

   b. Запустите сервер с помощью `pg-setup` от имени root следующим образом:

   ```
   /opt/pgpro/std-17/bin/pg-setup service start
   ```

   Как и сервер ванильного PostgreSQL, сервер Postgres Pro работает под именем пользователя `postgres`.

   ### Примечание

   По умолчанию автоматический запуск сервера отключён, чтобы вы могли вручную управлять восстановлением базы данных после перезагрузки системы. Однако вы можете настроить и автоматический запуск сервера Postgres Pro. За подробностями обратитесь к Подразделу 16.1.4.2.

16.1.4.1. Выбор устанавливаемых пакетов

В следующей таблице перечислены все имеющиеся пакеты Postgres Pro Standard.

Таблица 16.1. Пакеты Postgres Pro Standard

Пакет	Описание
`postgrespro-std-17`	Пакет верхнего уровня, устанавливающий и настраивающий Postgres Pro для серверных и клиентских систем. Не используйте этот пакет при обновлении или миграции. **Важно** При установке пакета `postgrespro-std-17` могут быть удалены существующие инсталляции продуктов на базе Postgres Pro и PostgreSQL. И напротив, установленный пакет Postgres Pro может быть автоматически удалён, если вы впоследствии развернёте другой продукт на базе PostgreSQL.
`postgrespro-std-17-client`	Стандартные клиентские приложения, такие как psql и pg_dump.
`postgrespro-std-17-libs`	Общие библиотеки, требующиеся для развёртывания клиентских приложений, включая libpq; библиотеки времени выполнения для обработчика ECPG.
`postgrespro-std-17-server`	Сервер Postgres Pro и серверный язык программирования PL/pgSQL.
`postgrespro-std-17-contrib`	Дополнительные расширения и программы, разворачиваемые на серверах баз данных.
`postgrespro-std-17-devel`	Заголовочные файлы и библиотеки для разработки клиентских приложений и серверных расширений. В системах на базе Debian этот пакет называется `postgrespro-std-17-dev`.
`postgrespro-std-17-plperl`	Реализация языка Perl для программирования на стороне сервера (см. Главу 43).
`postgrespro-std-17-plpython3`	Реализация языка Python 3 для программирования на стороне сервера (см. Главу 44).
`postgrespro-std-17-pltcl`	Реализация языка Tcl для программирования на стороне сервера (см. Главу 42).
`postgrespro-std-17-docs`	Документация на английском языке.
`postgrespro-std-17-docs-ru`	Документация на русском языке.
`postgrespro-std-17-test`	Тестовые скрипты для сервера. Этот пакет представлен только для систем на базе RHEL и SUSE.
`postgrespro-std-17-jit`	Этот пакет обеспечивает поддержку JIT-компиляции.

Пакет	Описание
	Этот пакет предоставляется только для архитектуры x86_64 и только для поддерживаемых систем Debian и Ubuntu, Astra Linux 1.7/1.8, поддерживаемых систем Альт, CentOS 8, Rocky Linux 8/9, Oracle Linux 8/9, AlmaLinux 9, SLES 15 и RHEL 8/9. Чтобы узнать об использовании JIT больше, обратитесь к Главе 29.
`mamonsu`	mamonsu — агент мониторинга для сбора метрик Postgres Pro и операционной системы.
`pg-portal-modify-std-17`	Расширение, позволяющее модифицировать курсоры Postgres Pro.
`pg-probackup-std-17`	Утилита pg_probackup.
`pgpro-controldata`	Утилита pgpro_controldata, показывающая управляющую информацию кластера БД PostgreSQL/Postgres Pro и параметры совместимости кластера и/или сервера.
`pgpro-pwr-std-17`	Расширение pgpro_pwr, позволяющее получать отчёты по нагрузке, полезные для выявления наиболее ресурсоёмких операций в базе данных.
`pgpro-stats-std-17`	Расширение pgpro_stats, собирающее статистику выполнения SQL-операторов, подсчитывающее статистику событий ожидания и предоставляющее прочие полезные метрики.

Также Postgres Pro включает библиотеки и утилиты, необходимые для предоставляемых пакетов и сервера Postgres Pro. Эти пакеты предоставляются только для дистрибутивов, в которых отсутствует требуемая версия таких библиотек.

Таблица 16.2. Сторонние библиотеки и утилиты, собранные для Postgres Pro Standard

Библиотека/утилита	Описание
`liblz4`	Библиотека для быстрого сжатия данных без потерь.
`freetds`	Библиотека для реализации протокола TDS. Используется расширением `tds-fdw` и включает в себя `freetds-devel`, `freetds-doc` и `freetds-libs`. Предоставляется для RHEL 8/9.
`libunwind`	Библиотека для раскрутки стека, используемая параметром `crash_info`. Предоставляется для RHEL 8.
`libbacktrace`	Библиотека для обратной трассировки стека, используемая параметром `crash_info`. Предоставляется для РЕД ОС 7.3/8 и RHEL 8/9.
`libsybdb5`	Библиотека для реализации протокола TDS. Предоставляется для SLES 15.
`perl-IO-Tty, perl-IPC-Run`	Две библиотеки Perl, используемые в рамках TAP-тестирования и доступные в пакете `postgrespro-std-17-devel` для разработчиков расширений и инженеров. `perl-IO-Tty` предо-

Библиотека/утилита	Описание
	ставляется для РЕД ОС МУРОМ 7.3/8, RHEL 8/9, а `perl-IPC-Run` — для РЕД ОС МУРОМ 7.3/8, RHEL 8/9, SLES 15.

Кроме этого, имеются отдельные пакеты с несколькими дополнительными модулями, собранные специально для совместимости с Postgres Pro Standard:

Таблица 16.3. Сторонние пакеты, собранные для Postgres Pro Standard

Пакет	Описание
`hypopg-std-17`	Расширение hypopg, обеспечивающее поддержку гипотетических индексов в Postgres Pro.
`ldap2pg`	Утилита ldap2pg для автоматизации процесса создания, обновления и удаления ролей и пользователей Postgres Pro из корпоративного каталога.
`odbc-postgrespro-std-17`	Драйвер открытого соединения с базами данных (open database connectivity, ODBC) для доступа к системам управления базами данных. В качестве альтернативы можно использовать `odbc-postgresql`, однако это может привести к конфликтам с устаревшими библиотеками ванильной версии PostgreSQL, которые не поддерживают функциональность новых библиотек.
`oracle-fdw-std-17`	Расширение oracle_fdw, предоставляющее обёртку сторонних данных для доступа к базам данных Oracle. Этот пакет предоставляется только для операционных систем на платформе Intel.
`orafce-std-17`	В этом пакете представлены реализации некоторых функций, которые существуют в СУБД Oracle, но отсутствуют в Postgres Pro (или работают немного по-другому).
`pg-filedump-std-17`	Утилита pg_filedump, представляющая файлы кучи и индексов, а также управляющие файлы Postgres Pro в понятном человеку виде.
`pg-repack-std-17`	pg_repack — расширение Postgres Pro и утилита для реорганизации таблиц.
`pgbouncer`	pgbouncer — пул соединений для Postgres Pro.
`pgpro-pgbadger`	pgbadger — анализатор журналов Postgres Pro, генерирующий подробные отчёты и графики.
`pgvector-std-17`	Расширение pgvector, предоставляющее поиск сходства векторов в Postgres Pro.
`pldebugger-std-17`	Набор разделяемых библиотек, реализующих API для отладки функций PL/pgSQL в Postgres Pro.
`plv8-std-17`	Разделяемая библиотека PLV8, добавляющая в Postgres Pro поддержку процедурного языка Javascript в реализации V8.

Пакет	Описание
	Этот пакет предоставляется только для систем Debian и Ubuntu, Альт 10/11, SLES 15, RHEL 8/9 и РЕД ОС МУРОМ 7.3.
`tds-fdw-std-17`	Расширение tds_fdw, предоставляющее обёртку сторонних данных для подключения к Microsoft SQL Server и другим базам данных, использующим протокол TDS (Tabular Data Stream, Поток табличных данных).
`wal2json-std-17`	Расширение wal2json для логического декодирования, которое преобразует изменения базы данных из журнала предзаписи (WAL) в формат JSON.

Кроме того, Postgres Pro предоставляет отдельные пакеты с отладочной информацией для некоторых операционных систем:

- В системах на базе Debian это пакет `postgrespro-std-17-dbg`.
- В системах на базе RHEL это пакет `postgrespro-std-17-debuginfo`.
- В ОС Альт Линукс для всех пакетов с двоичными файлами имеются соответствующие пакеты `-debuginfo`.

Для серверных инсталляций требуются как минимум следующие пакеты:

- `postgrespro-std-17-server`
- `postgrespro-std-17-client`
- `postgrespro-std-17-libs`

Для использования дополнительных расширений Postgres Pro вы должны также установить пакет `postgrespro-std-17-contrib`. В системах на базе Debian пакет `postgrespro-std-17-server` зависит от пакета `postgrespro-std-17-contrib`, поэтому последний должен всегда устанавливаться вместе с сервером.

Для клиентских инсталляций обычно достаточно установить пакеты `postgrespro-std-17-client` и `postgrespro-std-17-libs`. Если вы используете независимые приложения и вам не нужны стандартные клиентские утилиты, такие как psql, вы можете установить только пакет `postgrespro-std-17-libs`.

Для рабочих станций разработчиков необходимый минимум составляют следующие пакеты:

- `postgrespro-std-17-libs`
- `postgrespro-std-17-devel`/ `postgrespro-std-17-dev`

Также вы можете установить и настроить сервер с тестовой базой данных. Подробнее о настройке, которая может потребоваться дополнительно, рассказывается в Подразделе 16.1.5.

16.1.4.2. Настройка автоматического запуска сервера

Если вы производите расширенную установку, автоматический запуск сервера по умолчанию отключается. После создания базы данных по умолчанию вы можете настроить автоматический запуск сервера при загрузке системы, используя имеющиеся в вашей операционной системе механизмы управления службами или сторонние средства обеспечения высокой степени доступности. Для облегчения этой задачи в пакете `postgrespro-std-17-server` поставляется скрипт pg-setup, помещаемый в каталог `/opt/pgpro/std-17/bin`.

Для настройки автозапуска сервера запустите скрипт pg-setup со следующими параметрами:

```
pg-setup service enable
```

Если требуется, вы можете отключить автозапуск сервера, используя тот же скрипт:

```
pg-setup service disable
```

Вы также можете использовать системные средства управления службами, непосредственно задействуя скрипты для систем инициализации SysV init.d и systemd, предоставляемые в пакете `postgrespro-std-17-server`. В зависимости от дистрибутива Linux Postgres Pro поддерживает следующие механизмы управления службами:

Дистрибутив Linux	**Предоставляемые скрипты**
Системы на базе RHEL, SLES 15	Файл службы systemd
Debian, Ubuntu, Альт 10/11	И файл службы systemd, и скрипт init.d в стиле SysV

Для включения автоматического запуска сервера средствами systemd выполните следующую команду:

```
systemctl enable postgrespro-std-17
```

Для использования скрипта init.d в стиле SysV:

- В системах Альт Линукс выполните следующую команду:

  ```
  systemctl enable postgrespro-std-17
  ```

- В системах Debian воспользуйтесь скриптом `update-rc.d`. За подробностями обратитесь к соответствующей странице man.

16.1.5. Установка на рабочих станциях разработчиков

Хотя может быть достаточно установить пакеты `postgrespro-std-17-libs` и `postgrespro-std-17-devel`/`postgrespro-std-17-dev`, обычно на машинах разработчиков удобно иметь и установленный сервер. Для быстрой установки вы можете воспользоваться пакетом `postgrespro-std-17`, который автоматически настроит предоставляемые клиентские и серверные программы, а также создаст начальную базу данных. Однако если вы планируете использовать одновременно несколько продуктов на базе PostgreSQL, следуйте инструкциям по расширенной установке, приведённым в Подразделе 16.1.4.

Чтобы скомпилировать программы с библиотеками Postgres Pro, используя утилиту pg_config, поставляемую с Postgres Pro, необходимо, чтобы её путь был указан в переменной `PATH` до путей каких-либо других версий pg_config. Заметьте, что в системах на базе RHEL pg_config не добавляется в `PATH` автоматически. Если в вашей системе нет других версий pg_config, вы можете воспользоваться утилитой pg-wrapper, поставляемой в пакете `postgrespro-std-17-client`, для создания символической ссылки на pg_config в стандартном каталоге исполняемых файлов.

Для компиляции программ с использованием pkg-config добавьте путь `/opt/pgpro/std-17/lib/pkgconfig/` в переменную среды `PKG_CONFIG_PATH`.

Если вы хотите компилировать расширения Postgres Pro с поддержкой встраивания JIT, выполните дополнительные требования:

- Установите пакет LLVM для разработки и компилятор Clang. Вы должны выбрать пакеты той же версии, какая использовалась для сборки пакета `postgrespro-std-17-jit`, устанавливаемом на сервере. Определить версию, требующуюся для текущего выпуска Postgres Pro, вы можете по значению `CLANG` в файле `/opt/pgpro/std-17/lib/pgxs/src/Makefile.global`.
- Выполняя команды `make` и `make install`, добавьте параметр `with-llvm=yes` для компиляции и установки файлов с битовым кодом для вашего расширения. По умолчанию компиляции битового кода отключена, так как она зависит от доступности компилятора Clang.

16.1.5.1. Использование сторонних программ с Postgres Pro

Чтобы использовать сервер Postgres Pro с клиентской программой из стороннего продукта, вы можете установить версию библиотек PostgreSQL, с которыми компилировалась эта программа. Например, если программа поставляется в составе ванильного PostgreSQL, вам может потребоваться установить пакеты `libpq` или `postgresql-libs`, предоставляемые для вашего дистрибутива Linux. В этом случае программа, возможно, не будет использовать некоторые новые возможности сервера Postgres Pro, но скорее всего она и не поддерживает их.

Если вы предпочитаете использовать библиотеки Postgres Pro со сторонней программой или хотели бы задействовать новую возможность, которая не требует изменения самого клиентского приложения, например, проверку подлинности SCRAM, вы можете перекомпилировать вашу программу с библиотеками Postgres Pro.

> **Важно**
>
> Если программа скомпилирована с одной версией libpq, а используется с другой, её стабильная работа не гарантируется.

Если вы создаёте пакеты `.rpm` или `.deb` для вашей программы, рекомендуется сделать следующее:

1. Добавьте путь `/opt/pgpro/std-17/bin` в переменную `PATH` в сборочных скриптах (в файле `.spec` или `debian/rules`).
2. Добавьте `postgrespro-std-17-dev` в теги `BuildDepends` или `BuildRequires` описания пакета вашей программы.

Тем самым вы обеспечите вызов в процессе создания пакетов нужной версии pg_config при каждой пересборке пакета исходного кода.

16.1.6. Настройка нескольких экземпляров Postgres Pro

Чтобы настроить в Linux несколько экземпляров сервера Postgres Pro с разными каталогами данных, нужно проделать следующее:

1. Установите и настройте Postgres Pro, следуя указаниям в Подразделе 16.1.3 или Подразделе 16.1.4.
2. После создания начальной базы данных запустите initdb и укажите путь к другому каталогу данных и любые другие параметры, требующиеся для инициализации другого экземпляра сервера.
3. Задайте другие порты для экземпляров серверов в соответствующих файлах `postgresql.conf`, чтобы не допустить конфликтов.
4. Если требуется, настройте автоматический запуск сервера следующим образом:
   a. Создайте копию файла `/etc/init.d/postgrespro-std-17` или `/lib/systemd/system/postgrespro-std-17.service` с другим именем и поменяйте в ней путь к каталогу данных.
   b. Включите автоматический запуск сервера, используя скрипты автозапуска, предоставленные для вашего системного механизма управления службами, вместо `pg-setup`, как описано в Подразделе 16.1.4.2. Убедитесь в том, что вы используете переименованные копии скриптов, созданные на шаге 1.

16.1.7. Замечания о совместимости с антивирусами

Настоятельно не рекомендуется использовать антивирусное ПО в системах, где работает Postgres Pro, так как антивирус создаёт дополнительную нагрузку и может оказывать непредсказуемое влияние на работу базы данных, что чревато проблемами с производительностью и надёжностью. Если вам необходимо использовать антивирусное ПО, обязательно добавьте в исключения антивируса следующие каталоги, так как они не содержат исполняемых файлов:

- Каталог `PGDATA` с основными данными кластера (обычно `/var/lib/pgpro/std-17/data` или другой каталог, заданный в параметрах initdb)
- Каталоги созданных табличных пространств

16.2. Установка дополнительно поставляемых модулей

Postgres Pro поставляется с набором дополнительных серверных расширений, или модулей. В Linux они распространяются в отдельном пакете `postgrespro-contrib`. В Windows они устанавливаются вместе с серверными компонентами.

Установив двоичные файлы, вы должны развернуть в базе данных дополнительные нужные вам расширения. В большинстве случае для этого достаточно выполнить команду CREATE EXTENSION. Однако для некоторых расширений также требуется, чтобы при запуске сервера загружались определённые разделяемые библиотеки. Если вы хотите использовать такие расширения, вам нужно настроить параметр

```
shared_preload_libraries = 'lib1, lib2, lib3'
```

в файле `postgresql.conf` вашего экземпляра Postgres Pro и перезапустить сервер, а затем выполнить команду `CREATE EXTENSION`.

Более конкретные инструкции по установке и настройке определённого расширения вы можете найти в его документации в разделе Приложение F.

Чтобы получить список расширений, доступных в вашей инсталляции Postgres Pro, просмотрите системный каталог `pg_available_extensions`.

16.2.1. Установка новых расширений в сертифицированной редакции

Работа в среде, требующей высокого уровня безопасности, накладывает некоторые ограничения. Поскольку работать от имени суперпользователя запрещается из-за неограниченных прав доступа, выполнение операций можно поручить обычным пользователям (например, администратору СУБД).

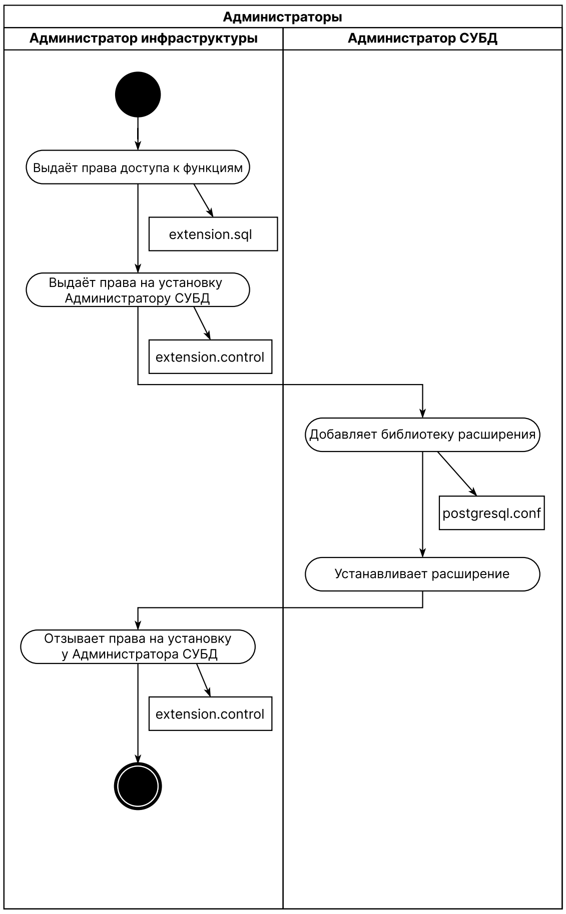
Устанавливать доверенные расширения и работать с ними можно с правами обычных пользователей, а прочие расширения требуют повышенного уровня безопасности. Установку некоторых расширений в среде с высоким уровнем безопасности, например pg_proaudit, следует осуществлять по специальной процедуре.

В этой процедуре участвуют администратор инфраструктуры и администратор СУБД:

- Администратор инфраструктуры отвечает за общую безопасность системы и не управляет Postgres Pro.
- Администратор СУБД, представленный в Postgres Pro ролью `PGPRO_DBMS_ADMIN`, отвечает за конфигурацию, настройку и администрирование СУБД Postgres Pro.

Действия, которые администраторам следует выполнить для установки нового расширения, показаны на Рисунке 16.1.

Рисунок 16.1. Действия администраторов для установки расширения



16.2.1.1. Администратор инфраструктуры: необходимые действия

Файлы конфигурации расширений в каталоге `share/extension` разрешено изменять только администратору инфраструктуры.

Файл `.sql` расширения содержит объявление SQL-функций. По умолчанию вызывать их могут только суперпользователи. В ходе процедуры установки расширения администратор инфраструктуры выполняет следующие действия:

1. Добавляет предложения `GRANT` в файл `.sql`, как показано ниже, что позволяет администратору СУБД использовать эти функции без необходимости запрашивать доступ у суперпользователя:

   ```
   -- Создание новых версий объектов
   CREATE FUNCTION pg_proaudit_show()
   RETURNS TABLE(db_name text,
     event_type text,
     object_type text,
     object_oid oid,
     role_name text)
   AS 'MODULE_PATHNAME', 'pg_proaudit_show_conf'
   LANGUAGE C VOLATILE;
   REVOKE ALL ON FUNCTION pg_proaudit_show() FROM public;
   ```

   Администратор инфраструктуры предоставляет права обычному пользователю:

   ```
   -- Создание новых версий объектов
   CREATE FUNCTION pg_proaudit_show()
   RETURNS TABLE(db_name text,
     event_type text,
   ```

```
    object_type text,
    object_oid oid,
    role_name text)
  AS 'MODULE_PATHNAME', 'pg_proaudit_show_conf'
  LANGUAGE C VOLATILE;
  REVOKE ALL ON FUNCTION pg_proaudit_show() FROM public;
  GRANT ALL ON FUNCTION pg_proaudit_show() TO PGPRO_DBMS_ADMIN;
```

2. Позволяет обычному пользователю установить расширение, изменяя или добавляя свойство `trusted` в файле `.control` расширения, и тем самым предоставляя временное разрешение на установку:

   ```
   trusted = true
   ```

Для использования обёрток сторонних данных ролью `PGPRO_DBMS_ADMIN` требуется специальное разрешение безопасности. Чтобы предоставить такое разрешение, администратор инфраструктуры добавляет команду `GRANT USAGE ON FOREIGN DATA WRAPPER` в соответствующий файл `.sql` расширения. Ниже приведён пример для postgres_fdw:

```
GRANT USAGE ON FOREIGN DATA WRAPPER postgres_fdw TO PGPRO_DBMS_ADMIN;
```

16.2.1.2. Администратор СУБД: необходимые действия

Администратору СУБД разрешено изменять файлы конфигурации Postgres Pro, за исключением `pg_hba.conf`, в котором хранится информация, относящаяся к безопасности. Изменять файл конфигурации `pg_hba.conf` разрешено только администратору инфраструктуры. В рамках процедуры установки расширения администратор СУБД выполняет следующие действия:

1. Добавляет файл соответствующей библиотеки в переменную `shared_preload_libraries` в файле конфигурации `postgresql.conf` и перезагружает сервер баз данных для применения изменений.
2. Создаёт расширение, используя команду CREATE EXTENSION.

16.2.1.3. Администратор инфраструктуры: последний шаг

Чтобы обеспечить безопасность во время использования расширения, администратор инфраструктуры возвращает свойство `trusted` в исходное состояние:

- Если свойство не было указано, удаляет его.
- Если свойство имело значение `FALSE`, возвращает ему значение `FALSE`.

16.3. Миграция на Postgres Pro

Разные основные версии Postgres Pro, а также разные продукты на базе PostgreSQL, имеющие одинаковую основную версию, могут иметь несовместимые на двоичном уровне базы данных, так что просто заменить программу сервера и продолжить использовать его в общем случае нельзя. Чтобы преобразовать базы данных с более старой основной версии, вам потребуется выполнить выгрузку/загрузку данных с помощью pg_dumpall или использовать pg_upgrade.

Инструкции по обновлению для определённой версии вы найдёте в Замечаниях к выпуску для этой версии Postgres Pro.

Глава 17. Подготовка к работе и сопровождение сервера

В этой главе рассказывается, как установить и запустить сервер баз данных, и о том, как он взаимодействует с операционной системой.

Указания в этой главе даны в предположении, что вы работаете с PostgreSQL в чистом виде, без дополнительной инфраструктуры, например, с экземпляром, собранным из исходного кода согласно инструкциям из предыдущих глав. Если вы используете сервер PostgreSQL, распространяемый производителем в виде готового продукта, вероятнее всего, производитель специально позаботился о том, чтобы сервер баз данных устанавливался и запускался принятым в системе образом. Подробности вы можете узнать в документации используемого вами продукта.

17.1. Учётная запись пользователя Postgres Pro

Как и любую другую серверную службу, доступную для внешнего мира, Postgres Pro рекомендуется запускать под именем отдельного пользователя. Эта учётная запись должна владеть только данными, которыми управляет сервер, и разделять её с другими службами не следует. (Например, будет неразумным выбрать в качестве такого пользователя `nobody`.) В частности, рекомендуется не назначать этого пользователя владельцем исполняемых файлов Postgres Pro, чтобы их нельзя было подменить в случае компрометации серверного процесса.

При установке пакетов PostgreSQL, поставляемого в виде продукта, подходящая учётная запись обычно создаётся автоматически.

Для создания пользователя в Unix-подобной системе следует искать команду `useradd` или `adduser`. В качестве имени пользователя часто используется postgres, и именно это имя предполагается в данной документации, но вы можете выбрать и другое, если захотите.

17.2. Создание кластера баз данных

Прежде чем вы сможете работать с базами данных, вы должны проинициализировать область хранения баз данных на диске. Мы называем это хранилище *кластером баз данных*. (В SQL применяется термин «кластер каталога».) Кластер баз данных представляет собой набор баз, управляемых одним экземпляром работающего сервера. После инициализации кластер будет содержать базу данных с именем `postgres`, предназначенную для использования по умолчанию утилитами, пользователями и сторонними приложениями. Сам сервер баз данных не требует наличия базы `postgres`, но многие внешние вспомогательные программы рассчитывают на её существование. При инициализации в каждом кластере создаются ещё две базы: `template1` и `template0`. Как можно понять из их имён, они применяются впоследствии в качестве шаблонов создаваемых баз данных; использовать их в качестве рабочих не следует. (За информацией о создании новых баз данных в кластере обратитесь к Главе 21.)

С точки зрения файловой системы кластер баз данных представляет собой один каталог, в котором будут храниться все данные. Мы называем его *каталогом данных* или *областью данных*. Где именно хранить данные, вы абсолютно свободно можете выбирать сами. Какого-либо стандартного пути не существует, но часто данные размещаются в `/usr/local/pgsql/data` или в `/var/lib/pgsql/data`. Прежде чем с каталогом данных можно будет работать, его нужно инициализировать, используя программу initdb, которая устанавливается в составе Postgres Pro.

Если вы используете PostgreSQL в виде готового продукта, в нём могут быть приняты определённые соглашения о расположении каталога данных, и может также предоставляться скрипт для создания этого каталога данных. В этом случае следует воспользоваться этим скриптом, а не запускать непосредственно `initdb`. За подробностями обратитесь к документации используемого вами продукта.

Чтобы инициализировать кластер баз данных вручную, запустите `initdb`, передав в параметре `-D` путь к желаемому расположению данных кластера в файловой системе, например:

```
$ initdb -D /usr/local/pgsql/data
```

Заметьте, что эту команду нужно выполнять от имени пользователя Postgres Pro, о котором говорится в предыдущем разделе.

Подсказка

В качестве альтернативы параметра `-D` можно установить переменную окружения `PGDATA`.

Также можно запустить команду `initdb`, воспользовавшись программой pg_ctl , примерно так:

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

Этот вариант может быть удобнее, если вы используете `pg_ctl` для запуска и остановки сервера (см. Раздел 17.3), так как `pg_ctl` будет единственной командой, с помощью которой вы будете управлять экземпляром сервера баз данных.

Команда `initdb` попытается создать указанный вами каталог, если он не существует. Конечно, она не сможет это сделать, если `initdb` не будет разрешено записывать в родительский каталог. Вообще рекомендуется, чтобы пользователь Postgres Pro был владельцем не только каталога данных, но и родительского каталога, так что такой проблемы быть не должно. Если же и нужный родительский каталог не существует, вам нужно будет сначала создать его, используя права root, если вышестоящий каталог защищён от записи. Таким образом, процедура может быть такой:

```
root# mkdir /usr/local/pgsql
root# chown postgres /usr/local/pgsql
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

Команда `initdb` не будет работать, если указанный каталог данных уже существует и содержит файлы; это мера предохранения от случайной перезаписи существующей инсталляции.

Так как каталог данных содержит все данные базы, очень важно защитить его от неавторизованного доступа. Для этого `initdb` лишает прав доступа к нему всех пользователей, кроме пользователя Postgres Pro и, возможно, его группы. Если группе разрешается доступ, то только для чтения. Это позволяет непривилегированному пользователю, входящему в одну группу с владельцем кластера, делать резервные копии данных кластера или выполнять другие операции, для которых достаточно доступа только для чтения.

Заметьте, чтобы корректно разрешить или запретить доступ группы к данным существующего кластера, необходимо выключить кластер и установить соответствующий режим для всех каталогов и файлов до запуска Postgres Pro. В противном случае в каталоге данных возможно смешение режимов. Для кластеров, к которым имеет доступ только владелец, требуется установить режим `0700` для каталогов и `0600` для файлов, а для кластеров, в которых также разрешается чтение группой, режим `0750` для каталогов и `0640` для файлов.

Однако даже когда содержимое каталога защищено, если проверка подлинности клиентов настроена по умолчанию, любой локальный пользователь может подключиться к базе данных и даже стать суперпользователем. Если вы не доверяете другим локальным пользователям, мы рекомендуем использовать один из параметров команды `initdb`: `-W`, `--pwprompt` или `--pwfile`, — и назначить пароль суперпользователя баз данных. Кроме того, воспользуйтесь параметром `-A scram-sha-256`, чтобы отключить разрешённый по умолчанию режим аутентификации `trust`; либо измените сгенерированный файл `pg_hba.conf` после выполнения `initdb`, но *перед* тем, как запустить сервер в первый раз. (Возможны и другие разумные подходы — применить режим проверки подлинности `peer` или ограничить подключения на уровне файловой системы. За дополнительными сведениями обратитесь к Главе 19.)

Команда `initdb` также устанавливает для кластера баз данных локаль по умолчанию. Обычно она просто берёт параметры локали из текущего окружения и применяет их к инициализируемой базе данных. Однако можно выбрать и другую локаль для базы данных; за дополнительной информацией обратитесь к Разделу 22.1. Команда `initdb` задаёт порядок сортировки по умолчанию для применения в определённом кластере баз данных, и хотя новые базы данных могут создаваться с иным порядком сортировки, порядок в базах-шаблонах, создаваемых initdb, можно изменить, только если удалить и пересоздать их. Также учтите, что при использовании локалей, отличных от `C` и `POSIX`, возможно снижение производительности. Поэтому важно правильно выбрать локаль с самого начала.

Команда `initdb` также задаёт кодировку символов по умолчанию для кластера баз данных. Обычно она должна соответствовать кодировке локали. За подробностями обратитесь к Разделу 22.3.

Для локалей, отличных от `C` и `POSIX`, порядок сортировки символов зависит от системной библиотеки локализации, а он, в свою очередь, влияет на порядок ключей в индексах. Поэтому кластер нельзя перевести на несовместимую версию библиотеки ни путём восстановления снимка, ни через двоичную репликацию, ни перейдя на другую операционную систему или обновив её версию.

17.2.1. Использование дополнительных файловых систем

Во многих инсталляциях кластеры баз данных создаются не в «корневом» томе, а в отдельных файловых системах (томах). Если вы решите сделать так же, то не следует выбирать в качестве каталога данных самый верхний каталог дополнительного тома (точку монтирования). Лучше всего создать внутри каталога точки монтирования каталог, принадлежащий пользователю Postgres Pro, а затем создать внутри него каталог данных. Это исключит проблемы с разрешениями, особенно для таких операций, как pg_upgrade, и при этом гарантирует чистое поведение в случае, если дополнительный том окажется отключён.

17.2.2. Файловые системы

Вообще говоря, для Postgres Pro может использоваться любая файловая система с семантикой POSIX. Пользователи предпочитают различные файловые системы по самым разным причинам, в частности, по соображениям производительности, изученности или поддержки поставщиком. Как показывает практика, в результате лишь смены файловой системы или корректировки её параметров при прочих равных не следует ожидать значительного изменения производительности или поведения.

17.2.2.1. NFS

Каталог данных Postgres Pro может размещаться и в файловой системе NFS. Postgres Pro не подстраивается специально под NFS, что означает, что с NFS он работает точно так же, как и с локально подключёнными устройствами. Postgres Pro не использует такую функциональность файловых систем, которая имеет свои особенности в NFS, например блокировки файлов.

Единственное убедительное требование — используя NFS с Postgres Pro, монтируйте эту файловую систему в режиме `hard`. При использовании режима `hard` процессы могут «зависать» на неопределённое время в случае сетевых проблем, поэтому могут потребоваться особые меры контроля. В режиме `soft` системные вызовы будут прерываться в случаях перебоев в сети, но Postgres Pro не повторяет вызовы, прерванные таким образом, и это будет проявляться в ошибках ввода-вывода.

Использовать параметр монтирования `sync` не обязательно. Поведения режима `async` достаточно, так как Postgres Pro вызывает `fsync` в нужные моменты для сброса кеша записи (так же, как и с локальной файловой системой). Однако параметр `sync` настоятельно рекомендуется использовать при экспортировании файловой системы на *сервере* NFS в тех ОС, где он поддерживается (в основном это касается Linux). В противном случае не гарантируется, что в результате выполнения `fsync` или аналогичного вызова NFS-клиентом данные действительно окажутся в надёжном хранилище на сервере, вследствие чего возможно повреждение данных, как и при выключенном параметре fsync. Значения по умолчанию параметров монтирования и экспортирования меняются от производителя к производителю и от версии к версии, поэтому рекомендуется перепроверить их или, возможно, явно задать нужные значения во избежание неоднозначности.

В некоторых случаях внешнее устройство хранение может быть подключено по NFS или посредством низкоуровневого протокола, например iSCSI. В последнем случае такое хранилище представляется в виде блочного устройства, и на нём можно создать любую файловую систему. При этом администратору не придётся иметь дело со странностями NFS, но надо понимать, что сложности управления удалённым хранилищем в таком случае просто перемещаются на другие уровни.

17.3. Запуск сервера баз данных

Важно

При установке двоичных пакетов в системах Linux базы данных по умолчанию располагаются в каталоге `/var/lib/pgpro/std-17/data`, если явно не будет задан другой каталог. За подробностями обратитесь к Разделу 16.1.

Чтобы кто-либо смог обратиться к базе данных, необходимо сначала запустить сервер баз данных. Программа сервера называется `postgres`.

Если вы используете PostgreSQL в виде готового продукта, в нём наверняка реализована возможность запуска сервера в виде фонового задания так, как это принято в вашей операционной системе. Использовать предоставленную продуктом инфраструктуру для запуска сервера гораздо проще, чем пытаться разобраться, как это сделать самостоятельно. За подробностями обратитесь к документации используемого вами продукта.

Самый прямолинейный вариант запуска сервера вручную — просто выполнить непосредственно `postgres`, указав расположение каталога данных в ключе `-D`, например:

```
$ postgres -D /usr/local/pgsql/data
```

В результате сервер продолжит работу на переднем плане. Запускать эту команду следует под именем учётной записи Postgres Pro. Без параметра `-D` сервер попытается использовать каталог данных, указанный в переменной окружения `PGDATA`. Если и эта переменная не определена, сервер не запустится.

Однако обычно лучше запускать `postgres` в фоновом режиме. Для этого можно применить обычный синтаксис, принятый в оболочке Unix:

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

Важно где-либо сохранять информацию, которую выводит сервер в каналы stdout и stderr, как показано выше. Это полезно и для целей аудита, и для диагностики проблем. (Более глубоко работа с файлами журналов рассматривается в Разделе 23.3.)

Программа `postgres` также принимает несколько других параметров командной строки. За дополнительными сведениями обратитесь к справочной странице postgres и к следующей Главе 18.

Такой вариант запуска довольно быстро может оказаться неудобным. Поэтому для упрощения подобных задач предлагается вспомогательная программа pg_ctl. Например:

```
pg_ctl start -l logfile
```

запустит сервер в фоновом режиме и направит выводимые сообщения сервера в указанный файл журнала. Параметр `-D` для неё имеет то же значение, что и для программы `postgres`. С помощью `pg_ctl` также можно остановить сервер.

Обычно возникает желание, чтобы сервер баз данных сам запускался при загрузке операционной системы. Скрипты автозапуска для разных систем разные, но в составе PostgreSQL предлагается несколько типовых скриптов в каталоге `contrib/start-scripts`. Для установки такого скрипта в систему требуются права root.

В различных системах приняты разные соглашения о порядке запуска служб в процессе загрузки. Во многих системах для этого используется файл `/etc/rc.local` или `/etc/rc.d/rc.local`. В других применяются каталоги `init.d` или `rc.d`. Однако при любом варианте запускаться сервер должен от имени пользователя Postgres Pro, но *не root* или какого-либо другого пользователя. Поэтому команду запуска обычно следует записывать в форме `su postgres -c '...'`. Например:

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

Ниже приведены более конкретные предложения для нескольких основных ОС. (Вместо указанных нами шаблонных значений необходимо подставить правильный путь к каталогу данных и фактическое имя пользователя.)

- Для запуска во FreeBSD воспользуйтесь файлом *contrib/start-scripts/freebsd* в дереве исходного кода PostgreSQL.
- В OpenBSD, добавьте в файл `/etc/rc.local` следующие строки:

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
      su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -
  D /usr/local/pgsql/data'
      echo -n ' postgresql'
  fi
  ```

- В системах Linux можно добавить

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
  ```

  в `/etc/rc.d/rc.local` или `/etc/rc.local`.

  Используя systemd, вы можете применить следующий файл описания службы (например, `/etc/systemd/system/postgresql.service`):

  ```
  [Unit]
  Description=Postgres Pro database server
  Documentation=man:postgres(1)
  After=network-online.target
  Wants=network-online.target

  [Service]
  Type=notify
  User=postgres
  ExecStart=/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
  ExecReload=/bin/kill -HUP $MAINPID
  KillMode=mixed
  KillSignal=SIGINT
  TimeoutSec=infinity

  [Install]
  WantedBy=multi-user.target
  ```

  Для использования `Type=notify` требуется, чтобы сервер был скомпилирован с указанием `configure --with-systemd`.

  Особого внимания заслуживает значение тайм-аута. На момент написания этой документации по умолчанию в systemd принят тайм-аут 90 секунд, так что процесс, не сообщивший о своей готовности за это время, будет уничтожен. Но серверу Postgres Pro при запуске может потребоваться выполнить восстановление после сбоя, так что переход в состояние готовности может занять гораздо больше времени. Предлагаемое значение `infinity` отключает логику тайм-аута.

- В NetBSD можно использовать скрипт запуска для FreeBSD или для Linux, в зависимости от предпочтений.
- В Solaris, создайте файл с именем `/etc/init.d/postgresql`, содержащий следующую стоку:

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/
  data"
  ```

Затем создайте символическую ссылку на него в каталоге `/etc/rc3.d` с именем `S99postgresql`.

Когда сервер работает, идентификатор его процесса (PID) сохраняется в файле `postmaster.pid` в каталоге данных. Это позволяет исключить запуск нескольких экземпляров сервера с одним каталогом данных, а также может быть полезно для выключения сервера.

17.3.1. Сбои при запуске сервера

Есть несколько распространённых причин, по которым сервер может не запуститься. Чтобы понять, чем вызван сбой, просмотрите файл журнала сервера или запустите сервер вручную (не перенаправляя его потоки стандартного вывода и ошибок) и проанализируйте выводимые сообщения. Ниже мы рассмотрим некоторые из наиболее частых сообщений об ошибках более подробно.

```
LOG:  could not bind IPv4 address "127.0.0.1": Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds
 and retry.
FATAL:  could not create any TCP/IP sockets
```

Это обычно означает именно то, что написано: вы пытаетесь запустить сервер на том же порту, на котором уже работает другой. Однако если сообщение ядра не `Address already in use` или подобное, возможна и другая проблема. Например, при попытке запустить сервер с номером зарезервированного порта будут выданы такие сообщения:

```
$ postgres -p 666
LOG:  could not bind IPv4 address "127.0.0.1": Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds
 and retry.
FATAL:  could not create any TCP/IP sockets
```

Следующее сообщение:

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

может означать, что установленный для вашего ядра предельный размер разделяемой памяти слишком мал для рабочей области, которую пытается создать Postgres Pro (в данном примере 4011376640 байт). Такая ситуация возможна, только если в `shared_memory_type` выбран вариант `sysv`. В этом случае можно попытаться запустить сервер с меньшим числом буферов (shared_buffers) или переконфигурировать ядро и увеличить допустимый размер разделяемой памяти. Вы также можете увидеть это сообщение при попытке запустить несколько серверов на одном компьютере, если запрошенный ими объём памяти в сумме превышает установленный в ядре предел.

Сообщение:

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

не означает, что у вас закончилось место на диске. Это значит, что установленное в вашем ядре предельное число семафоров System V меньше, чем количество семафоров, которое пытается создать Postgres Pro. Как и в предыдущем случае можно попытаться обойти эту проблему, запустив сервер с меньшим числом допустимых подключений (max_connections), но в конце концов вам придётся увеличить этот предел в ядре.

Настройка средств IPC в стиле System V описывается в Подразделе 17.4.1.

17.3.2. Проблемы с подключениями клиентов

Хотя ошибки подключений, возможные на стороне клиента, довольно разнообразны и зависят от приложений, всё же несколько проблем могут быть связаны непосредственно с тем, как был запущен сервер. Описание ошибок, отличных от описанных ниже, следует искать в документации соответствующего клиентского приложения.

```
psql: ошибка: не удалось подключиться к серверу "server.joe.com" (123.123.123.123) по
 порту 5432: Connection refused
        Он действительно работает по адресу "server.joe.com" (123.123.123.123) и
 принимает TCP-соединения (порт 5432)?
```

Это общая проблема «я не могу найти сервер и начать взаимодействие с ним». Показанное выше сообщение говорит о попытке установить подключение по TCP/IP. Очень часто объясняется это тем, что сервер просто забыли настроить для работы по протоколу TCP/IP.

Кроме того, при попытке установить подключение к локальному серверу через Unix-сокет можно получить такое сообщение:

```
psql: ошибка: не удалось подключиться к серверу через сокет "/tmp/.s.PGSQL.5432": No
 such file or directory
        Он действительно работает локально и принимает соединения через этот сокет?
```

Если сервер действительно работает, проверьте, что указываемый клиентом путь сокета (здесь `/tmp`) соответствует значению unix_socket_directories.

Сообщение об ошибке подключения всегда содержит адрес сервера или путь к сокету, что помогает понять, куда именно подключается клиент. Если сервер на самом деле не принимает подключения по этому адресу, обычно выдаётся сообщение ядра `Connection refused` (В соединении отказано) или `No such file or directory` (Нет такого файла или каталога), приведённое выше. (Важно понимать, что `Connection refused` в данном контексте *не* означает, что сервер получил запрос на подключение и отверг его. В этом случае были бы выданы другие сообщения, например, показанные в Разделе 19.15.) Другие сообщения об ошибках, например `Connection timed out` (Тайм-аут соединения) могут сигнализировать о более фундаментальных проблемах, например, о нарушениях сетевых соединений или о блокировании подключения брандмауэром.

17.4. Управление ресурсами ядра

Postgres Pro иногда может исчерпывать некоторые ресурсы операционной системы до предела, особенно при запуске нескольких копий сервера в одной системе или при работе с очень большими базами. В этом разделе описываются ресурсы ядра, которые использует Postgres Pro, и подходы к решению проблем, связанных с ограниченностью этих ресурсов.

17.4.1. Разделяемая память и семафоры

Postgres Pro требует, чтобы операционная система предоставляла средства межпроцессного взаимодействия (IPC), в частности, разделяемую память и семафоры. Системы семейства Unix обычно предоставляют функции IPC в стиле «System V» или функции IPC в стиле «POSIX» или и те, и другие. В Windows эти механизмы реализованы по-другому, но здесь это не рассматривается.

По умолчанию Postgres Pro запрашивает очень небольшой объём разделяемой памяти System V и намного больший объём анонимной разделяемой памяти `mmap`. Возможен также вариант использования одной большой области памяти System V (см. shared_memory_type). Помимо этого при запуске сервера создаётся значительное количество семафоров (в стиле System V или POSIX). В настоящее время семафоры POSIX используются в системах Linux и FreeBSD, а на других платформах используются семафоры System V.

Функции IPC в стиле System V обычно сталкиваются с лимитами на уровне системы. Когда Postgres Pro превышает один из этих лимитов, сервер отказывается запускаться, но должен выдать полезное сообщение, говорящее об ошибке и о том, что с ней делать. (См. также Подраздел 17.3.1.) Соответствующие параметры ядра в разных системах называются аналогично (они перечислены в Таблице 17.1), но устанавливаются по-разному. Ниже предлагаются способы их изменения для некоторых систем.

Таблица 17.1. Параметры IPC в стиле System V

Имя	Описание	Значения, необходимые для запуска одного экземпляра Postgres Pro
`SHMMAX`	Максимальный размер сегмента разделяемой памяти (в байтах)	как минимум 1 КБ, но значение по умолчанию обычно гораздо больше
`SHMMIN`	Минимальный размер сегмента разделяемой памяти (в байтах)	1
`SHMALL`	Общий объём доступной разделяемой памяти (в байтах или страницах)	если в байтах, то же, что и `SHMMAX`; если в страницах, то `ceil(SHMMAX/PAGE_SIZE)` , плюс потребность других приложений
`SHMSEG`	Максимальное число сегментов разделяемой памяти для процесса	требуется только 1 сегмент, но значение по умолчанию гораздо больше
`SHMMNI`	Максимальное число сегментов разделяемой памяти для всей системы	как `SHMSEG` плюс потребность других приложений
`SEMMNI`	Максимальное число идентификаторов семафоров (т. е., их наборов)	как минимум `ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 7) / 19)` плюс потребность других приложений
`SEMMNS`	Максимальное число семафоров для всей системы	`ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 7) / 19) * 20` плюс потребность других приложений
`SEMMSL`	Максимальное число семафоров в наборе	не меньше 20
`SEMMAP`	Число записей в карте семафоров	см. текст
`SEMVMX`	Максимальное значение семафора	не меньше 1000 (по умолчанию оно обычно равно 32767; без необходимости менять его не следует)

Postgres Pro запрашивает небольшой блок разделяемой памяти System V (обычно 48 байт на 64-битной платформе) для каждой копии сервера. В большинстве современных операционных систем такой объём выделяется без проблем. Однако если запускать много копий сервера или явно настроить сервер для использования больших объёмов разделяемой памяти System V (см. shared_memory_type и dynamic_shared_memory_type), может понадобиться увеличить значение `SHMALL`, задающее общий объём разделяемой памяти System V, доступный для всей системы. Заметьте, что `SHMALL` во многих системах задаётся в страницах, а не в байтах.

Менее вероятны проблемы с минимальным размером сегментов разделяемой памяти (`SHMMIN`), который для Postgres Pro не должен превышать примерно 32 байт (обычно это всего 1 байт). Максимальное число сегментов для всей системы (`SHMMNI`) или для одного процесса (`SHMSEG`) тоже обычно не влияет на работоспособность сервера, если только это число не равно нулю.

Когда Postgres Pro использует семафоры System V, он занимает по одному семафору на одно разрешённое подключение (max_connections), на разрешённый рабочий процесс автоочистки (autovacuum_max_workers), разрешённый процесс-передатчик WAL (max_wal_senders) и фоновый процесс (max_worker_processes), в наборах по 19. В каждом таком наборе есть также 20-й семафор, содержащий «магическое число», позволяющий обнаруживать коллизии с наборами семафоров других приложений. Максимальное число семафоров в системе задаётся параметром `SEMMNS`, который, следовательно, должен быть равен как минимум сумме `max_connections`, `autovacuum_max_workers`, `max_wal_senders` и `max_worker_processes`, плюс один дополнительный на каждые 19 семафоров подключений и рабочих процессов (см. формулу в Таблице 17.1). Параметр `SEMMNI` определяет максимальное число наборов семафоров, которые могут существовать в системе в один момент времени. Таким образом, его значение должно быть не меньше чем `ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 7) / 19)`. В качестве временного решения проблем, которые вызываются этими ограничениями, но обычно сопровождаются некорректными сообщениями функции `semget`, например, «No space left on device» (На устройстве не осталось места) можно уменьшить число разрешённых соединений.

В некоторых случаях может потребоваться увеличить `SEMMAP` как минимум до уровня `SEMMNS`. Если в системе есть такой параметр (а во многих системах его нет), он определяет размер карты ресурсов семафоров, в которой выделяется запись для каждого непрерывного блока семафоров. Когда набор семафоров освобождается, эта запись либо добавляется к существующей соседней записи, либо регистрируется как новая запись в карте. Если карта переполняется, освобождаемые семафоры теряются (до перезагрузки). Таким образом, фрагментация пространства семафоров может со временем привести к уменьшению числа доступных семафоров.

Другие параметры, связанные с «аннулированием операций» с семафорами, например, `SEMMNU` и `SEMUME`, на работу Postgres Pro не влияют.

При использовании семафоров POSIX требуемое их количество не отличается от количества для System V, то есть по одному семафору на разрешённое подключение (max_connections), на разрешённый рабочий процесс автоочистки (autovacuum_max_workers), разрешённый процесс-передатчик WAL (max_wal_senders) и фоновый процесс (max_worker_processes). На платформах, где предпочитается этот вариант, отсутствует определённый лимит ядра на количество семафоров POSIX.

FreeBSD

Параметры разделяемой памяти по умолчанию вполне приемлемы, если вы не выберете в `shared_memory_type` вариант `sysv`. Семафоры System V на этой платформе не используются.

Значения параметров IPC по умолчанию можно изменить, используя возможности `sysctl` или `loader`. С помощью `sysctl` можно задать следующие параметры:

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
```

Чтобы эти изменения сохранялись после перезагрузки, измените `/etc/sysctl.conf`.

Если вы выбрали в `shared_memory_type` вариант `sysv`, возможно, вы захотите настроить ядро так, чтобы разделяемая память System V всегда находилась в ОЗУ и никогда не выгружалась в пространство подкачки. Это можно сделать, установив с помощью `sysctl` параметр `kern.ipc.shm_use_phys`.

Если вы запускаете сервер в «камере» FreeBSD, установите для параметра `sysvshm` значение `new`, чтобы у сервера было собственное отдельное пространство имён разделяемой памяти System V. (До версии 11.0 во FreeBSD требовалось разрешать общий доступ из камер к пространству имён IPC ведущего узла и принимать меры для недопущения конфликтов.)

NetBSD

Параметры разделяемой памяти по умолчанию вполне приемлемы, если вы не выберете в `shared_memory_type` вариант `sysv`. Обычно имеет смысл увеличить `kern.ipc.semmni` и `kern.ipc.semmns`, так как их значения по умолчанию в NetBSD слишком малы.

Параметры IPC можно изменить, воспользовавшись командой `sysctl`, например:

```
$ sysctl -w kern.ipc.semmni=100
```

Чтобы эти параметры сохранялись после перезагрузки, измените `/etc/sysctl.conf`.

Если вы выбрали в `shared_memory_type` вариант `sysv`, возможно, вы захотите настроить ядро так, чтобы разделяемая память System V всегда находилась в ОЗУ и никогда не выгружалась в пространство подкачки. Это можно сделать, установив с помощью `sysctl` параметр `kern.ipc.shm_use_phys`.

OpenBSD

Параметры разделяемой памяти по умолчанию вполне приемлемы, если вы не выберете в `shared_memory_type` вариант `sysv` Обычно имеет смысл увеличить `kern.seminfo.semmni` и `kern.seminfo.semmns`, так как их значения по умолчанию в OpenBSD слишком малы.

Параметры IPC можно изменить, воспользовавшись командой `sysctl`, например:

```
$ sysctl kern.seminfo.semmni=100
```

Чтобы эти параметры сохранялись после перезагрузки, измените `/etc/sysctl.conf`.

Linux

Параметры разделяемой памяти по умолчанию вполне приемлемы, если вы не выберете в `shared_memory_type` вариант `sysv`. И даже в этом случае их потребуется увеличить только для старых ядер, в которых эти параметры по умолчанию имеют маленькие значения. Семафоры System V на этой платформе не используются.

Параметры разделяемой памяти можно изменить, воспользовавшись командой `sysctl`. Например, так можно выделить 16 ГБ:

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

Чтобы сохранить эти изменения после перезагрузки, воспользуйтесь файлом `/etc/sysctl.conf`.

macOS

Параметры разделяемой памяти и семафоров по умолчанию вполне приемлемы, если вы не выберете в `shared_memory_type` вариант `sysv`.

Для настройки разделяемой памяти в macOS рекомендуется создать файл `/etc/sysctl.conf` и записать в него присваивания переменных следующим образом:

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

Заметьте, что в некоторых версиях macOS, *все пять* параметров разделяемой памяти должны быть установлены в `/etc/sysctl.conf`, иначе их значения будут проигнорированы.

Значение `SHMMAX` должно быть кратно 4096.

`SHMALL` на этой платформе измеряется в страницах (по 4 КБ).

Все параметры, кроме `SHMMNI` можно изменить «на лету», воспользовавшись командой sysctl. Но тем не менее лучше задавать выбранные вами значения в `/etc/sysctl.conf`, чтобы они сохранялись после перезагрузки.

Solaris
illumos

Параметры разделяемой памяти по умолчанию вполне приемлемы для большинства применений Postgres Pro. По умолчанию Solaris устанавливает в `SHMMAX` четверть объёма ОЗУ. Чтобы выбрать другое значение, задайте соответствующий параметр проекта, связанного с пользователем `postgres`. Например, выполните от имени `root` такую команду:

```
projadd -c "Postgres Pro DB User" -K "project.max-shm-memory=(privileged,8GB,deny)"
 -U postgres -G postgres user.postgres
```

Эта команда создаёт проект `user.postgres` и устанавливает максимальный объём разделяемой памяти для пользователя `postgres` равным 8 ГБ. Это изменение вступает в силу при следующем входе этого пользователя или при перезапуске Postgres Pro (не перезагрузке конфигурации). При этом подразумевается, что Postgres Pro выполняется пользователем `postgres` в группе `postgres`. Перезагружать систему после этой команды не нужно.

Для серверов баз данных, рассчитанных на большое количество подключений, рекомендуется также изменить следующие параметры:

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

Кроме того, если Postgres Pro у вас выполняется внутри зоны, может понадобиться также увеличить лимиты на использование ресурсов зоны. Получить дополнительную информацию о `проектах` и команде `prctl` можно в *Руководстве системного администратора* (System Administrator's Guide), «Главе 2: Проекты и задачи» (Chapter2: Projects and Tasks).

17.4.2. RemoveIPC в systemd

Если используется systemd, необходимо позаботиться о том, чтобы ресурсы IPC (включая разделяемую память) не освобождались преждевременно операционной системой. Это особенно актуально при сборке и установке Postgres Pro из исходного кода. Пользователей дистрибутивных пакетов Postgres Pro это касается в меньшей степени, так как пользователь `postgres` обычно создаётся как системный пользователь.

Параметр `RemoveIPC` в `logind.conf` определяет, должны ли объекты IPC удаляться при полном выходе пользователя из системы. На системных пользователей это не распространяется. Этот параметр по умолчанию включён в стандартной сборке systemd, но в некоторых дистрибутивах операционных систем он по умолчанию отключён.

Обычно негативный эффект включения этого параметра проявляется в том, что объекты разделяемой памяти, используемые для параллельного выполнения запросов, удаляются без видимых причин, что приводит к появлению ошибок и предупреждений при попытке открыть и удалить их, например:

```
WARNING:  could not remove shared memory segment "/PostgreSQL.1450751626": No such file
 or directory
```

(ПРЕДУПРЕЖДЕНИЕ: ошибка при удалении сегмента разделяемой памяти "/PostgreSQL.1450751626": Нет такого файла или каталога) Различные типы объектов IPC (разделяемая память/семафоры, System V/POSIX) обрабатываются в systemd несколько по-разному, поэтому могут наблюдаться ситуации, когда некоторые ресурсы IPC не удаляются так, как другие. Однако полагаться на эти тонкие различия не рекомендуется.

Событие «выхода пользователя из системы» может произойти при выполнении задачи обслуживания или если администратор войдёт под именем `postgres`, а затем выйдет, либо случится что-то подобное, так что предотвратить это довольно сложно.

Какой пользователь является «системным», определяется во время компиляции systemd, исходя из значения `SYS_UID_MAX` в `/etc/login.defs`.

Скрипт упаковывания и развёртывания сервера должен предусмотрительно создавать пользователя `postgres` как системного пользователя, используя команды `useradd -r`, `adduser --system` или равнозначные.

Если же учётная запись пользователя была создана некорректно и изменить её невозможно, рекомендуется задать

```
RemoveIPC=no
```

в `/etc/systemd/logind.conf` или другом подходящем файле конфигурации.

> **Внимание**
>
> Необходимо предпринять минимум одно из этих двух действий, иначе сервер Postgres Pro будет очень нестабильным.

17.4.3. Ограничения ресурсов

В Unix-подобных операционных системах существуют различные типы ограничений ресурсов, которые могут влиять на работу сервера Postgres Pro. Особенно важны ограничения на число процессов для пользователя, число открытых файлов и объём памяти для каждого процесса. Каждое из этих ограничений имеет «жёсткий» и «мягкий» предел. Мягкий предел действительно ограничивает использование ресурса, но пользователь может увеличить его значение до жёсткого предела. Изменить жёсткий предел может только пользователь root. За изменение этих параметров отвечает системный вызов `setrlimit`. Управлять этими ресурсами в командной строке позволяет встроенная команда `ulimit` (в оболочках Bourne) и `limit` (csh). В системах семейства BSD различными ограничениями ресурсов, устанавливаемыми при входе пользователя, управляет файл `/etc/login.conf`. За подробностями обратитесь к документации операционной системы. Для Postgres Pro интерес представляют параметры `maxproc`, `openfiles` и `datasize`. Они могут задаваться, например так:

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(Здесь `-cur` обозначает мягкий предел. Чтобы задать жёсткий предел, нужно заменить это окончание на `-max`.)

Ядро также может устанавливать общесистемные ограничения на использование некоторых ресурсов.

- В Linux максимальное число открытых файлов, которое поддерживает ядро, определяется параметром ядра `fs.file-max`. Изменить этот предел можно, воспользовавшись командой `sysctl -w fs.file-max=N`. Чтобы эти изменения сохранялись после перезагрузки, следует добавить присваивание в файл `/etc/sysctl.conf`. Максимальное число файлов для одного процесса задаётся при компиляции ядра; за дополнительными сведения обратитесь к `/usr/src/linux/Documentation/proc.txt`.

Сервер Postgres Pro использует для обслуживания каждого подключения отдельный процесс, так что возможное число процессов должно быть не меньше числа разрешённых соединений плюс число процессов, требуемых для остальной системы. Это обычно не проблема, но когда в одной системе работает множество серверов, предел может быть достигнут.

В качестве максимального числа открытых файлов по умолчанию обычно выбираются «социально-ориентированные» значения, позволяющие использовать одну систему нескольким пользователям так, чтобы ни один из них не потреблял слишком много системных ресурсов. Если вы запускаете в системе несколько серверов, это должно вполне устраивать, но на выделенных машинах может возникнуть желание увеличить этот предел.

С другой стороны, некоторые системы позволяют отдельным процессам открывать очень много файлов и если это делают сразу несколько процессов, они могут легко исчерпать общесистемный предел. Если вы столкнётесь с такой ситуацией, но не захотите менять общесистемное ограничение, вы можете ограничить использование открытых файлов сервером Postgres Pro, установив параметр конфигурации max_files_per_process.

Ещё одно ограничение в ядре, с которым можно столкнуться, когда устанавливается большое количество клиентских подключений, — максимальная длина очереди подключений к сокету. Если количество запросов на подключение за короткий промежуток времени превышает этот максимум, некоторые из них будут отклонены до того, как сервер Postgres Pro сможет их обработать, при этом клиенты получат неинформативное сообщение об ошибке подключения типа «Resource temporarily unavailable» (Ресурс временно недоступен) или «Connection refused» (Не удалось подключиться). Предел длины очереди на многих платформах по умолчанию составляет 128. Чтобы увеличить его, настройте соответствующий параметр ядра через sysctl и перезапустите сервер Postgres Pro. Этот параметр называется `net.core.somaxconn` в Linux, `kern.ipc.soacceptqueue` в последних версиях FreeBSD и `kern.ipc.somaxconn` в macOS и на других платформах BSD.

17.4.4. Чрезмерное выделение памяти в Linux

В Linux механизм виртуальной памяти по умолчанию работает не оптимально для Postgres Pro. Вследствие того, что ядро выделяет память в чрезмерном объёме, оно может уничтожить главный управляющий процесс Postgres Pro (postmaster), если при выделении памяти процессу Postgres Pro или другому процессу виртуальная память будет исчерпана.

Когда это происходит, вы можете получить примерно такое сообщение ядра (где именно искать это сообщение, можно узнать в документации вашей системы):

```
Out of Memory: Killed process 12345 (postgres).
```

Это сообщение говорит о том, что процесс `postgres` был уничтожен из-за нехватки памяти. Хотя существующие подключения к базе данных будут работать по-прежнему, новые подключения приниматься не будут. Чтобы восстановить работу сервера, Postgres Pro придётся перезапустить.

Один из способов обойти эту проблему — запускать Postgres Pro на компьютере, где никакие другие процессы не займут всю память. Если физической памяти недостаточно, решить проблему также можно, увеличив объём пространства подкачки, так как уничтожение процессов при нехватке памяти происходит только когда заканчивается и физическая память, и место в пространстве подкачки.

Если памяти не хватает по вине самого Postgres Pro, эту проблему можно решить, изменив конфигурацию сервера. В некоторых случаях может помочь уменьшение конфигурационных параметров, связанных с памятью, а именно `shared_buffers`, `work_mem` и `hash_mem_multiplier`. В других случаях проблема может возникать, потому что разрешено слишком много подключений к самому серверу баз данных. Чаще всего в такой ситуации стоит уменьшить число подключений `max_connections` и организовать внешний пул соединений.

«Чрезмерное выделение» памяти можно предотвратить, изменив поведение ядра. Хотя при этом *OOM killer* (уничтожение процессов при нехватке памяти) всё равно может вызываться, вероятность такого уничтожения значительно уменьшается, а значит поведение системы становится более стабильным. Для этого нужно включить режим строгого выделения памяти, воспользовавшись `sysctl`:

```
sysctl -w vm.overcommit_memory=2
```

либо поместив соответствующую запись в `/etc/sysctl.conf`. Возможно, вы также захотите изменить связанный параметр `vm.overcommit_ratio`. За подробностями обратитесь к документации ядра *https://www.kernel.org/doc/Documentation/vm/overcommit-accounting*.

Другой подход, который можно применить (возможно, вместе с изменением `vm.overcommit_memory`), заключается в исключении процесса postmaster из числа возможных жертв при нехватке памяти. Для этого нужно задать для свойства *поправка очков OOM* этого процесса значение `-1000`. Проще всего это можно сделать, выполнив

```
echo -1000 > /proc/self/oom_score_adj
```

в скрипте запуска Postgres Pro непосредственно перед тем, как запускать `postgres`. Заметьте, что делать это надо под именем root, иначе ничего не изменится; поэтому проще всего вставить эту команду в стартовый скрипт, принадлежащий пользователю root. Если вы делаете это, вы также должны установить в данном скрипте эти переменные окружения перед запуском `postgres`:

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
```

```
export PG_OOM_ADJUST_VALUE=0
```

С такими параметрами дочерние процессы главного будут запускаться с обычной, нулевой поправкой очков OOM, так что при необходимости механизм OOM сможет уничтожать их. Вы можете задать и другое значение для `PG_OOM_ADJUST_VALUE`, если хотите, чтобы дочерние процессы исполнялись с другой поправкой OOM. (`PG_OOM_ADJUST_VALUE` также можно опустить, в этом случае подразумевается нулевое значение.) Если вы не установите `PG_OOM_ADJUST_FILE`, дочерние процессы будут работать с той же поправкой очков OOM, которая задана для главного процесса, что неразумно, так всё это делается как раз для того, чтобы главный процесс оказался на особом положении.

17.4.5. Огромные страницы в Linux

Использование огромных страниц (huge pages) снижает накладные расходы при работе с большими непрерывными блоками памяти, что характерно для Postgres Pro, особенно при большом объёме shared_buffers. Чтобы такие страницы можно было задействовать в Postgres Pro, ядро должно быть собрано с параметрами `CONFIG_HUGETLBFS=y` и `CONFIG_HUGETLB_PAGE=y`. Также вам понадобится настроить ОС, чтобы она могла выделить достаточное количество огромных страниц нужного размера. Вычисляемый параметр shared_memory_size_in_huge_pages показывает количество требуемых огромных страниц. Узнать значение этого параметра можно до запуска сервера через команду `postgres`, например:

```
$ postgres -D $PGDATA -C shared_memory_size_in_huge_pages
3170
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
$ ls /sys/kernel/mm/hugepages
hugepages-1048576kB  hugepages-2048kB
```

В этом примере размер по умолчанию составляет 2 МБ, но задав в параметре huge_page_size 2 МБ или 1 ГБ явным образом, вы получите в `shared_memory_size_in_huge_pages` пересчитанное количество страниц. В данном примере серверу потребуется `3170` огромных страниц, но можно запросить и больше, если огромные страницы будут использоваться и другими программами в этой системе. Выбранное значение можно задать так:

```
# sysctl -w vm.nr_hugepages=3170
```

Не забудьте добавить этот параметр в `/etc/sysctl.conf`, чтобы он действовал и после перезагрузки. Если же размер огромных страниц отличается от подразумеваемого по умолчанию, их количество можно задать так:

```
# echo 3170 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

Также можно задать эти параметры во время загрузки ОС, установив соответствующие параметры ядра, например `hugepagesz=2M hugepages=3170`.

Иногда ядро не может сразу выделить запрошенное количество огромных страниц из-за фрагментации, поэтому может потребоваться повторить эту команду или перезагрузить систему. (Немедленно после перезагрузки должен быть свободен больший объём памяти для преобразования в огромные страницы.) Чтобы проверить текущую ситуацию с размещением огромных страниц определённого размера, выполните:

```
$ cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

Также может потребоваться дать пользователю операционной системы, запускающему сервер БД, право использовать огромные страницы, установив его группу в `vm.hugetlb_shm_group` с помощью sysctl, и/или разрешить блокировать память, выполнив `ulimit -l`.

По умолчанию Postgres Pro пытается использовать огромные страницы стандартного размера, а в противном случае переходит к обычным страницам. Чтобы задействовать огромные страницы принудительно, можно установить для huge_pages значение `on` в `postgresql.conf`. Заметьте, что с таким значением Postgres Pro не сможет запуститься, если не получит достаточного количества огромных страниц.

Более подробно о механизме огромных страниц в Linux можно узнать в документации ядра: *https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt*.

17.5. Выключение сервера

Сервер баз данных можно отключить несколькими способами. На практике они все сводятся к отправке сигнала управляющему процессу `postgres`.

Если вы используете PostgreSQL в виде готового продукта и запускаете сервер, применяя предусмотренные в этом продукте средства, то и останавливать сервер вы должны, применяя те же средства. За подробностями обратитесь к документации используемого вами продукта.

Непосредственно управляя сервером, вы можете выбрать тот или иной вариант отключения, посылая разные сигналы главному процессу `postgres`:

SIGTERM

> Запускает так называемое *умное выключение*. Получив SIGTERM, сервер перестаёт принимать новые подключения, но позволяет всем существующим сеансам закончить работу в штатном режиме. Сервер будет отключён только после завершения всех сеансов. Если получая этот сигнал, сервер находится в процессе восстановления, восстановление и потоковая репликация будут прерваны только после завершения всех обычных сеансов.

SIGINT

> Запускает *быстрое выключение*. Сервер запрещает новые подключения и посылает всем работающим серверным процессам сигнал SIGTERM, в результате чего их транзакции прерываются и сами процессы завершаются. Управляющий процесс ждёт, пока будут завершены все эти процессы, и затем завершается сам.

SIGQUIT

> Запускает *немедленное выключение*. Сервер отправляет всем дочерним процессам сигнал SIGQUIT и ждёт их завершения. Если какие-либо из них не завершаются в течение 5 секунд, им посылается SIGKILL. Главный процесс сервера завершается, как только будут завершены все дочерние процессы, не выполняя обычную процедуру остановки БД. В результате при последующем запуске будет запущен процесс восстановления (воспроизведения изменений из журнала). Такой вариант выключения рекомендуется только в экстренных ситуациях.

Удобную возможность отправлять эти сигналы, отключающие сервер, предоставляет программа pg_ctl. Кроме того, в системах, отличных от Windows, соответствующий сигнал можно отправить с помощью команды `kill`. PID основного процесса `postgres` можно узнать, воспользовавшись программой `ps`, либо прочитав файл `postmaster.pid` в каталоге данных. Например, можно выполнить быстрое выключение так:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

Важно

Для выключения сервера не следует использовать сигнал SIGKILL. При таком выключении сервер не сможет освободить разделяемую память и семафоры. Кроме того, при уничтожении главного процесса `postgres` сигналом SIGKILL, он не успеет передать этот сигнал своим дочерним процессам, так что может потребоваться завершать и их вручную.

Чтобы завершить отдельный сеанс, не прерывая работу других сеансов, воспользуйтесь функцией `pg_terminate_backend()` (см. Таблицу 9.94) или отправьте сигнал SIGTERM дочернему процессу, обслуживающему этот сеанс.

17.6. Обновление кластера Postgres Pro

В этом разделе рассказывается, как обновить ваш кластер базы данных с одной версии Postgres Pro на другую.

Текущие номера версий Postgres Pro состоят из трёх цифровых групп: две из них берутся из версии PostgreSQL, а третья — номер релиза Postgres Pro. Например, в номере версии 12.11.2 число 12 обозначает основную версию, 11 — дополнительную, а 2 — номер выпуска Postgres Pro, то есть это второй выпуск после выхода PostgreSQL версии 12.11. Для выпусков Postgres Pro до версии 10.0 номера состояли из четырёх чисел, например 9.5.3.2. Тогда версия Postgres Pro образовывалась группой из трёх чисел версии PostgreSQL(например, 9.5.3), а четвёртая обозначала номер выпуска Postgres Pro, например цифра 2 означала, что это был второй выпуск Postgres Pro после выхода PostgreSQL версии 9.5.3.

Выпуски Postgres Pro одной дополнительной версии обычно совместимы с предыдущими и последующими выпусками. Если вы обновляете выпуск Postgres Pro на базе той же дополнительной версии, например с 12.11.1 до 12.11.2, обычно достаточно просто установить новый выпуск в текущий каталог инсталляции. Если же вы обновляете Postgres Pro на базе другой версии PostgreSQL, например с 12.11.1 до 12.15.1, *вам следует обратить особое внимание на раздел «Миграция»* в замечаниях к выпуску (Приложение E). Хотя вы можете перейти с одной версии на другую, пропустив промежуточные версии, обязательно ознакомьтесь с замечаниями к каждому выпуску, в том числе для каждой пропускаемой версии.

При обновлении *основных* версий Postgres Pro внутренний формат данных может меняться, что усложняет процедуру обновления. Традиционный способ переноса данных в новую основную версию — выгрузить данные из старой версии, а затем восстановить базу данных (это не самый быстрый вариант). Выполнить обновление быстрее позволяет pg_upgrade. Также для обновления можно использовать репликацию, как описано ниже.

Изменения основной версии обычно приносят какие-либо видимые пользователю несовместимости, которые могут требовать доработки приложений. Все подобные изменения описываются в замечаниях к выпуску (Приложение E), включая все необходимые для миграции инструкции в соответствующем разделе.

Осторожные пользователи обычно тестируют свои клиентские приложения с новой версией, прежде чем переходить на неё полностью; поэтому часто имеет смысл установить рядом старую и новую версии. При тестировании обновления основной версии Postgres Pro изучите следующие области возможных изменений:

Администрирование

Средства и функции, предоставляемые администраторам для наблюдения и управления сервером, часто меняются и совершенствуются в каждой новой версии.

SQL

В этой области чаще наблюдается появление новых возможностей команд SQL, чем изменение поведения существующих, если только об этом не говорится в замечаниях к выпуску.

API библиотек

Обычно библиотеки типа libpq только расширяют свою функциональность, если об обратном так же не говорится в замечаниях к выпуску.

Системные каталоги

Изменения в системных каталогах обычно влияют только на средства управления базами данных.

API сервера для кода на C

Сюда относятся изменения в API серверных функций, которые написаны на языке программирования C. Такие изменения затрагивают код, обращающийся к служебным функциям глубоко внутри сервера.

17.6.1. Обновление данных с применением pg_dumpall

Один из вариантов обновления заключается в выгрузке данных из одной основной версии Postgres Pro и загрузке их в другую — для этого необходимо использовать средство *логического* копирования, например pg_dumpall; копирование на уровне файловой системы не подходит. (В самом сервере есть проверки, которые не дадут использовать каталог данных от несовместимой версии Postgres Pro, так что если попытаться запустить с существующим каталогом данных неправильную версию сервера, никакого вреда не будет.)

Для создания копии рекомендуется применять программы pg_dump и pg_dumpall от *новой* версии Postgres Pro, чтобы воспользоваться улучшенными функциями, которые могли в них появиться. Текущие версии этих программ способны читать данные любой версии сервера, начиная с 9.2.

В следующих указаниях предполагается, что сервер установлен в каталоге `/usr/local/pgsql`, а данные находятся в `/usr/local/pgsql/data`. Вам нужно подставить свои пути.

1. При запуске резервного копирования убедитесь в том, что в базе данных не производятся изменения. Изменения не повлияют на целостность полученной копии, но изменённые данные, само собой, в неё не попадут. Если потребуется, измените разрешения в файле `/usr/local/pgsql/data/pg_hba.conf` (или подобном), чтобы подключиться к серверу могли только вы. За дополнительными сведениями об управлении доступом обратитесь к Главе 19.

   Чтобы получить копию всех ваших данных, введите:

   ```
   pg_dumpall > выходной_файл
   ```

   Для создания резервной копии вы можете воспользоваться программой pg_dumpall от текущей версии сервера; за подробностями обратитесь к Подразделу 24.1.2. Однако для лучшего результата стоит попробовать pg_dumpall из Postgres Pro Standard 17.6.1, так как в эту версию вошли исправления ошибок и усовершенствования, по сравнению с предыдущими версиями. Хотя этот совет может показаться абсурдным, ведь новая версия ещё не установлена, ему стоит последовать, если вы планируете установить новую версию рядом со старой. В этом случае вы сможете выполнить установку как обычно, а перенести данные позже. Это также сократит время обновления.

2. Остановите старый сервер:

   ```
   pg_ctl stop
   ```

   В системах, где Postgres Pro запускается при загрузке, должен быть скрипт запуска, с которым можно сделать то же самое. Например, в Red Hat Linux может сработать такой вариант:

   ```
   /etc/rc.d/init.d/postgresql stop
   ```

   Подробнее запуск и остановка сервера описаны в Главе 17.

3. При восстановлении из резервной копии удалите или переименуйте старый каталог, где был установлен сервер, если его имя не привязано к версии. Разумнее будет переименовать каталог, а не удалять его, чтобы его можно было восстановить в случае проблем. Однако учтите, что этот каталог может занимать много места на диске. Переименовать каталог можно, например так:

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

   (Этот каталог нужно переименовывать (перемещать) как единое целое, чтобы относительные пути в нём не изменились.)

4. Установите новую версию Postgres Pro Standard.

5. При необходимости создайте новый кластер баз данных. Помните, что следующие команды нужно выполнять под именем специального пользователя БД (вы уже действуете под этим именем, если производите обновление).

   ```
   /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
   ```

6. Перенесите изменения, внесённые в предыдущие версии `pg_hba.conf` и `postgresql.conf`.

7. Запустите сервер баз данных, так же применяя учётную запись специального пользователя БД:

   `/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data`

8. Наконец, восстановите данные из резервной копии, выполнив:

   `/usr/local/pgsql/bin/psql -d postgres -f` ***`выходной_файл`***

   (При этом будет использоваться *новый* psql.)

Минимизировать время отключения сервера можно, установив новый сервер в другой каталог и запустив параллельно оба сервера, старый и новый, с разными портами. Затем можно будет перенести данные примерно так:

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

17.6.2. Обновление данных с применением pg_upgrade

Модуль pg_upgrade позволяет обновить инсталляцию Postgres Pro с одной основной версии на другую непосредственно на месте. Такое обновление может выполняться за считанные минуты, особенно в режиме `--link`. Для него требуются примерно те же подготовительные действия, что и для варианта с pg_dumpall: запустить/остановить сервер, выполнить initdb. Все эти действия описаны в документации pg_upgrade.

17.6.3. Обновление данных с применением репликации

Методы логической репликации также могут применяться для создания резервного сервера с обновлённой версией Postgres Pro. Это возможно благодаря тому, что логическая репликация поддерживается между разными основными версиями Postgres Pro. Резервный сервер может располагаться как на том же компьютере, так и на другом. Как только синхронизация с главным сервером (где работает старая версия Postgres Pro) будет завершена, можно сделать главным новый сервер, а старый экземпляр базы данных просто отключить. При таком переключении обновление возможно осуществить, прервав работу сервера всего на несколько секунд.

Этот вариант обновления можно осуществить, используя как встроенные средства логической репликации, так и внешние системы логической репликации, такие как pglogical, Slony, Londiste и Bucardo.

17.7. Защита от подмены сервера

Когда сервер работает, злонамеренный пользователь не может подставить свой сервер вместо него. Однако если сервер отключён, локальный пользователь может подменить нормальный сервер, запустив свой собственный. Поддельный сервер сможет читать пароли и запросы клиентов, хотя не сможет вернуть никакие данные, так как каталог `PGDATA` будет защищён от чтения посторонними пользователями. Такая подмена возможна потому, что любой пользователь может запустить сервер баз данных; клиент, со своей стороны, не может обнаружить подмену, если его не настроить дополнительно.

Один из способов предотвратить подмену для `локальных` подключений — использовать каталог Unix-сокетов (unix_socket_directories), в который сможет писать только проверенный локальный пользователь. Это не позволит злонамеренному пользователю создать в этом каталоге свой файл сокета. Если вас беспокоит, что некоторые приложения при этом могут обращаться к файлу сокета в `/tmp` и, таким образом, всё же будут уязвимыми, создайте при загрузке операционной системы символическую ссылку `/tmp/.s.PGSQL.5432`, указывающую на перемещённый файл сокета. Возможно, вам также придётся изменить скрипт очистки `/tmp`, чтобы он не удалял эту ссылку.

Также клиенты могут защитить `локальные` подключения, установив в параметре `requirepeer` имя пользователя, который должен владеть серверным процессом, подключённым к сокету.

Для защиты от подмены TCP-соединений можно либо использовать сертификаты SSL и проверять сертификат сервера со стороны клиентов, либо применять шифрование GSSAPI (или и то, и другое при использовании независимых подключений).

Для защиты от подмены соединения с SSL сервер надо настроить так, чтобы он принимал только подключения `hostssl` (см. Раздел 19.1) и имел ключ и сертификаты SSL (см. Раздел 17.9). Тогда TCP-клиент должен будет подключаться к серверу с параметром `sslmode=verify-ca` или `verify-full` и у него должен быть установлен соответствующий корневой сертификат (см. Подраздел 32.19.1). Как вариант, можно применять системный пул ЦС, предоставляемый реализацией SSL. В этом случае для безопасности принудительно применяется режим `sslmode=verify-full`, так как получение сертификатов, подписываемых публичным ЦС, является обычной практикой.

Для защиты от подмены соединения с сервером во время использования сетевой аутентификации по паролю с методом scram-sha-256 следует удостовериться, что подключение к серверу происходит по SSL-протоколу и с одним из методов, защищающих от подмены соединения, которые описаны в предыдущем абзаце. Кроме того, реализация SCRAM в libpq не защищает на протяжении всего процесса обмена информацией при аутентификации, однако использование параметра соединения `channel_binding=require` позволяет минимизировать вероятность подмены соединения. Злоумышленник, использующий подставной сервер для перехвата данных при обмене по SCRAM, может прибегнуть к офлайн-анализу, чтобы попытаться определить хешированный пароль клиента.

Для защиты от подмены соединения с GSSAPI сервер надо настроить так, чтобы он принимал только подключения `hostgssenc` (см. Раздел 19.1) и для них использовалась аутентификация `gss`. TCP-клиент в этом случае должен подключаться с параметром `gssencmode=require`.

17.8. Возможности шифрования

Postgres Pro обеспечивает шифрование на разных уровнях и даёт гибкость в выборе средств защиты данных в случае кражи сервера, от недобросовестных администраторов или в небезопасных сетях. Шифрование может также требоваться для защиты конфиденциальных данных, например, медицинских сведений или финансовых транзакций.

Шифрование паролей

Пароли пользователей базы данных хранятся в виде хешей (алгоритм хеширования определяется параметром password_encryption), так что администратор не может узнать, какой именно пароль имеет пользователь. Если шифрование SCRAM или MD5 применяется и при проверке подлинности, пароль не присутствует на сервере в открытом виде даже кратковременно, так как клиент шифрует его перед тем как передавать по сети. Предпочтительным методом является SCRAM, так как это стандарт, принятый в Интернете, и он более безопасен, чем собственный протокол проверки MD5 в Postgres Pro.

Защитное преобразование конкретных столбцов

Модуль pgcrypto позволяет хранить в защищённом виде избранные поля. Это полезно, если ценность представляют только некоторые данные. Чтобы прочитать эти поля, клиент передаёт ключ обратного защитного преобразования, сервер обратно преобразовывает данные и выдаёт их клиенту.

Обратно преобразованные данные и ключ обратного защитного преобразования недолгое время находятся на сервере в процессе обратного защитного преобразования и передачи данных. Именно в этот момент данные и ключи могут быть перехвачены тем, кто имеет полный доступ к серверу баз данных, например, системным администратором.

Шифрование раздела данных

Шифрование хранилища данных можно реализовать на уровне файловой системы или на уровне блоков. В Linux можно воспользоваться шифрованными файловыми системами eCryptfs и EncFS, а во FreeBSD есть PEFS. Шифрование всего диска на блочном уровне в Linux можно организовать, используя dm-crypt + LUKS, а во FreeBSD — модули GEOM, geli и gbde. Подобные возможности есть и во многих других операционных системах, включая Windows.

Этот механизм не позволяет читать незашифрованные данные с дисков в случае кражи дисков или всего компьютера. При этом он не защищает данные от чтения, когда эта файловая система смонтирована, так как на смонтированном устройстве операционная система видит все данные в незашифрованном виде. Однако чтобы смонтировать файловую систему, нужно передать операционной системе ключ (иногда он хранится где-то на компьютере, который выполняет монтирование).

Шифрование данных при передаче по сети

SSL-соединения шифруют все данные, передаваемые по сети: пароль, запросы и возвращаемые данные. Файл `pg_hba.conf` позволяет администраторам указать, для каких узлов будут разрешены незашифрованные соединения (`host`), а для каких будет требоваться SSL (`hostssl`). Кроме того, и на стороне клиента можно разрешить подключения к серверам только с SSL.

Защищённые GSSAPI соединения шифруют все данные, передаваемые по сети, включая запросы и возвращаемые данные. (Пароль по сети не передаётся.) В файле `pg_hba.conf` администраторы могут указать, для каких узлов будут разрешены незашифрованные соединения (`host`), а для каких будет требоваться шифрование GSSAPI (`hostgssenc`). В дополнение и клиенты можно настроить так, чтобы они подключались к серверу только с защитой GSSAPI (`gssencmode=require`).

Для шифрования передаваемых данных можно также применять Stunnel или SSH.

Проверка подлинности сервера SSL

И клиент, и сервер могут проверять подлинность друг друга по сертификатам SSL. Это требует дополнительной настройки на каждой стороне, но даёт более надёжную гарантию подлинности, чем обычные пароли. С такой защитой подставной компьютер не сможет представлять из себя сервер с целью получить пароли клиентов. Она также предотвращает атаки с посредником («man in the middle»), когда компьютер между клиентом и сервером представляется сервером и незаметно передаёт все запросы и данные между клиентом и подлинным сервером.

Шифрование на стороне клиента

Если системный администратор сервера, где работает база данных, не является доверенным, клиент должен сам шифровать данные; тогда незашифрованные данные никогда не появятся на этом сервере. В этом случае клиент шифрует данные, прежде чем передавать их серверу, а получив из базы данных результаты, он расшифровывает их для использования.

17.9. Защита соединений TCP/IP с применением SSL

В Postgres Pro встроена поддержка SSL для шифрования трафика между клиентом и сервером, что повышает уровень безопасности системы.

Термины SSL и TLS часто используются взаимозаменяемо для обозначения безопасного зашифрованного соединения по протоколу TLS. Хотя SSL-протоколы, являющиеся предшественниками TLS-протоколов, уже не поддерживаются, термин SSL всё ещё используется. В Postgres Pro термины SSL и TLS взаимозаменяемы.

17.9.1. Базовая настройка

Когда сервер Postgres Pro скомпилирован с SSL, его можно запустить с включённой поддержкой зашифрованных соединений по протоколам TLS, задав в `postgresql.conf` для параметра ssl значение `on`. Запущенный сервер будет принимать как обычные, так и SSL-подключения в одном порту TCP и будет согласовывать использование SSL с каждым клиентом. По умолчанию клиент выбирает режим подключения сам; как настроить сервер, чтобы он требовал использования только SSL для всех или некоторых подключений, вы можете узнать в Разделе 19.1.

Чтобы сервер мог работать в режиме SSL, ему необходимы файлы с сертификатом сервера и закрытым ключом. По умолчанию это должны быть файлы `server.crt` и `server.key`, соответственно, расположенные в каталоге данных, но можно использовать и другие имена и местоположения файлов, задав их в конфигурационных параметрах ssl_cert_file и ssl_key_file.

В Unix-подобных системах к файлу `server.key` должен быть запрещён любой доступ группы и всех остальных; чтобы установить такое ограничение, выполните `chmod 0600 server.key`. Возможен и другой вариант, когда этим файлом владеет root, а группа имеет доступ на чтение (то есть, маска разрешений `0640`). Данный вариант предназначен для систем, в которых файлами сертификатов и ключей управляет сама операционная система. В этом случае пользователь, запускающий сервер Postgres Pro, должен быть членом группы, имеющей доступ к указанным файлам сертификата и ключа.

Если к каталогу данных разрешён доступ группы, файлы сертификатов должны размещаться вне этого каталога для удовлетворения озвученных выше требований безопасности. Вообще говоря, доступ группы разрешается для того, чтобы непривилегированный пользователь мог производить резервное копирование базы данных, и в этом случае средство резервного копирования не сможет прочитать файлы сертификатов, что скорее всего приведёт к ошибке.

Если закрытый ключ защищён паролем, сервер спросит этот пароль и не будет запускаться, пока он не будет введён. По умолчанию использование такого пароля лишает возможности изменять конфигурацию SSL без перезагрузки сервера, однако параметр ssl_passphrase_command_supports_reload при некоторых условиях позволяет делать это. Более того, закрытые ключи, защищённые паролем, не годятся для использования в Windows.

Первым сертификатом в `server.crt` должен быть сертификат сервера, так как он должен соответствовать закрытому ключу сервера. В этот файл также могут быть добавлены сертификаты «промежуточных» центров сертификации. Это избавляет от необходимости хранить все промежуточные сертификаты на клиентах, при условии, что корневой и промежуточные сертификаты были созданы с расширениями `v3_ca`. (При этом в основных ограничениях сертификата устанавливается свойство `CA`, равное `true`.) Это также упрощает управление промежуточными сертификатами с истекающим сроком.

Добавлять корневой сертификат в `server.crt` нет необходимости. Вместо этого клиенты должны иметь этот сертификат в цепочке сертификатов сервера.

17.9.2. Конфигурация OpenSSL

Postgres Pro читает системный файл конфигурации OpenSSL. По умолчанию этот файл называется `openssl.cnf` и находится в каталоге, который сообщает команда `openssl version -d`. Если требуется указать другое расположение файла конфигурации, его можно задать в переменной окружения `OPENSSL_CONF`.

OpenSSL предоставляет широкий выбор шифров и алгоритмов аутентификации разной защищённости. Хотя список шифров может быть задан непосредственно в файле конфигурации OpenSSL, можно задать отдельные шифры именно для сервера баз данных, указав их в параметре ssl_ciphers в `postgresql.conf`.

> **Примечание**
>
> Накладные расходы, связанные с шифрованием, в принципе можно исключить, ограничившись только проверкой подлинности, то есть применяя шифр `NULL-SHA` или `NULL-MD5`. Однако в этом случае посредник сможет пропускать через себя и читать весь трафик между клиентом и сервером. Кроме того, шифрование привносит минимальную дополнительную нагрузку по сравнению с проверкой подлинности. По этим причинам использовать шифры NULL не рекомендуется.

17.9.3. Использование клиентских сертификатов

Чтобы клиенты должны были предоставлять серверу доверенные сертификаты, поместите сертификаты корневых центров сертификации (ЦС), которым вы доверяете, в файл в каталоге данных, укажите в параметре ssl_ca_file в `postgresql.conf` имя этого файла и добавьте вариант аутентификации `clientcert=verify-ca` или `clientcert=verify-full` в соответствующие строки `hostssl` в `pg_hba.conf`. В результате от клиента в процессе установления SSL-подключения будет затребован сертификат. (Как настроить сертификаты на стороне клиента, описывается в Разделе 32.19.)

Для записи `hostssl` с указанием `clientcert=verify-ca` сервер будет проверять, подписан ли сертификат клиента доверенным для него центром сертификации. Для указания `clientcert=verify-full` сервер не только проверяет сертификат по цепочке доверия, но и сверяет имя пользователя или имя, сопоставленное ему, с общим именем (`cn`, Common Name), указанным в представленном сертификате. Заметьте, что цепочка доверия для сертификата при использовании метода аутентификации `cert` проверяется всегда (см. Раздел 19.12).

Промежуточные сертификаты, которые составляют цепочку с существующими корневыми сертификатами, можно также включить в файл ssl_ca_file, если вы не хотите хранить их на стороне клиента (предполагается, что корневой и промежуточный сертификаты были созданы с расширениями `v3_ca`). Если установлен параметр ssl_crl_file или ssl_crl_dir, также проверяются списки отзыва сертификатов (Certificate Revocation List, CRL).

Параметр аутентификации `clientcert` можно использовать с любым методом проверки подлинности, но только в строках `pg_hba.conf` типа `hostssl`. Когда `clientcert` не задан, сервер проверяет клиентский сертификат по своему списку ЦС, только если сертификат предоставлен и этот список определён.

Есть два способа потребовать от пользователей предоставления сертификата при подключении.

Первый вариант заключается в использовании метода аутентификации `cert` в записях `hostssl` в `pg_hba.conf`. При этом сертификат будет использоваться и для проверки подлинности, и для защиты SSL-подключения. За подробностями обратитесь к Разделу 19.12. (Задавать явно какие-либо значения для параметра `clientcert` при использовании метода аутентификации `cert` не требуется.) В этом случае в качестве имени пользователя или сопоставляемого имени используется имя из поля `cn` (Common Name, Общее имя) сертификата.

Второй вариант сочетает использование произвольного метода аутентификации в записях `hostssl` с проверкой клиентских сертификатов, которая выполняется при значении параметра `clientcert`, равном `verify-ca` или `verify-full`. Первое значение включает только проверку достоверности сертификата, а последнее также сверяет имя пользователя или применяемое сопоставление с общим именем `cn` (Common Name), указанным в сертификате.

17.9.4. Файлы, используемые SSL-сервером

В Таблице 17.2 кратко описаны все файлы, имеющие отношение к настройке SSL на сервере. (Здесь приведены стандартные имена файлов. В конкретной системе они могут быть другими.)

Таблица 17.2. Файлы, используемые SSL-сервером

Файл	Содержимое	Назначение
ssl_cert_file (`$PGDATA/server.crt`)	сертификат сервера	отправляется клиенту для идентификации сервера
ssl_key_file (`$PGDATA/server.key`)	закрытый ключ сервера	подтверждает, что сертификат сервера был передан его владельцем; не гарантирует, что его владельцу можно доверять
ssl_ca_file	сертификаты доверенных ЦС	позволяет проверить, что сертификат клиента подписан доверенным центром сертификации
ssl_crl_file	сертификаты, отозванные центрами сертификации	сертификат клиента должен отсутствовать в этом списке

Сервер читает эти файлы при запуске или при перезагрузке конфигурации. В системах Windows они также считываются заново, когда для нового клиентского подключения запускается новый обслуживающий процесс.

Если в этих файлах при запуске сервера обнаружится ошибка, сервер откажется запускаться. Но если ошибка обнаруживается при перезагрузке конфигурации, эти файлы игнорируются и продолжает использоваться старая конфигурация SSL. В системах Windows, если в одном из этих файлов обнаруживается ошибка при запуске обслуживающего процесса, этот процесс не сможет устанавливать SSL-соединения. Во всех таких случаях в журнал событий сервера выводится сообщение об ошибке.

17.9.5. Создание сертификатов

Чтобы создать простой самоподписанный сертификат для сервера, действующий 365 дней, выполните следующую команду OpenSSL, заменив *`dbhost.yourdomain.com`* именем компьютера, где размещён сервер:

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

Затем выполните:

```
chmod og-rwx server.key
```

так как сервер не примет этот файл, если разрешения будут более либеральными, чем показанные. За дополнительными сведениями относительно создания закрытого ключа и сертификата сервера обратитесь к документации OpenSSL.

Хотя самоподписанный сертификат может успешно применяться при тестировании, в производственной среде следует использовать сертификат, подписанный центром сертификации (ЦС) (обычно это корневой ЦС предприятия).

Чтобы создать сертификат сервера, подлинность которого смогут проверять клиенты, сначала создайте запрос на получение сертификата (CSR) и файлы открытого/закрытого ключа:

```
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

Затем подпишите запрос ключом, чтобы создать корневой центр сертификации (с файлом конфигурации OpenSSL, помещённым в Linux в расположение по умолчанию):

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

Наконец, создайте сертификат сервера, подписанный новым корневым центром сертификации:

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key

openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

`server.crt` и `server.key` должны быть сохранены на сервере, а `root.crt` — на клиенте, чтобы клиент мог убедиться в том, что конечный сертификат сервера подписан центром сертификации, которому он доверяет. Файл `root.key` следует хранить в изолированном месте для создания сертификатов в будущем.

Также возможно создать цепочку доверия, включающую промежуточные сертификаты:

```
# корневой сертификат
openssl req -new -nodes -text -out root.csr \
```

```
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt

# промежуточный
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt

# конечный
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
  -CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
  -out server.crt
```

`server.crt` и `intermediate.crt` следует сложить вместе в пакет сертификатов и сохранить на сервере. Также на сервере следует сохранить `server.key`. Файл `root.crt` нужно сохранить на клиенте, чтобы клиент мог убедиться в том, что конечный сертификат сервера был подписан по цепочке сертификатов, связанных с корневым сертификатом, которому он доверяет. Файлы `root.key` и `intermediate.key` следует хранить в изолированном месте для создания сертификатов в будущем.

17.10. Защита соединений TCP/IP с применением GSSAPI

В Postgres Pro имеется собственная поддержка GSSAPI для шифрования трафика между клиентом и сервером, что повышает уровень безопасности. Чтобы её можно было использовать, и на сервере, и на клиенте должна быть установлена реализация GSSAPI (например, MIT Kerberos), и эта поддержка должна быть включена в Postgres Pro при сборке.

17.10.1. Базовая настройка

Сервер Postgres Pro будет принимать как обычные, так и зашифрованные GSSAPI подключения через один TCP-порт и будет согласовывать использование GSSAPI для шифрования (и для аутентификации) с каждым подключающимся клиентом. По умолчанию решение об использовании шифрования принимает клиент (это означает, что злоумышленник может от него отказаться); чтобы использование GSSAPI было обязательным, сервер можно настроить как описано в Разделе 19.1.

Когда GSSAPI используется для шифрования, часто GSSAPI применяют также и для аутентификации, так как нижележащий механизм в любом случае будет проверять подлинность клиента и сервера (этого требует реализация GSSAPI). Но это необязательно — вы можете выбрать любой другой поддерживаемый Postgres Pro метод аутентификации для осуществления дополнительной проверки.

Кроме настройки согласования, для использования шифрования GSSAPI не требуется никаких дополнительных операций, не считая тех, что нужны для включения аутентификации GSSAPI. (Подробнее о том, как это настраивается, рассказывается в Разделе 19.6.)

17.11. Защита соединений TCP/IP с применением туннелей SSH

Для защиты сетевых соединений клиентов с сервером Postgres Pro можно применить SSH. При правильном подходе это позволяет обеспечить должный уровень защиты сетевого трафика, даже для клиентов, не поддерживающих SSL.

Прежде всего убедитесь, что на компьютере с сервером Postgres Pro также работает сервер SSH и вы можете подключиться к нему через `ssh` каким-нибудь пользователем; затем вы сможете установить защищённый туннель с этим удалённым сервером. Защищённый туннель прослушивает локальный порт и перенаправляет весь трафик в порт удалённого компьютера. Трафик, передаваемый в удалённый порт, может поступить на его адрес `localhost` или на другой привязанный адрес, если это требуется. При этом не будет видно, что трафик поступает с вашего компьютера. Следующая команда устанавливает защищённый туннель между клиентским компьютером и удалённой машиной `foo.com`:

```
ssh -L 63333:localhost:5432 joe@foo.com
```

Первое число в аргументе `-L`, 63333 — это локальный номер порта туннеля; это может быть любой незанятый порт. (IANA резервирует порты с 49152 по 65535 для частного использования.) Имя или IP-адрес после него обозначает привязанный адрес с удалённой стороны, к которому вы подключаетесь, в данном случае это `localhost` (и это же значение по умолчанию). Второе число, 5432 — порт с удалённой стороны туннеля, то есть порт вашего сервера баз данных. Чтобы подключиться к этому серверу через созданный тоннель, нужно подключиться к порту 63333 на локальном компьютере:

```
psql -h localhost -p 63333 postgres
```

Для сервера баз данных это будет выглядеть так, как будто вы пользователь `joe` компьютера `foo.com`, подключающийся к адресу `localhost`, и он будет применять ту процедуру проверки подлинности, которая установлена для подключений данного пользователя к этому привязанному адресу. Заметьте, что сервер не будет считать такое соединение защищённым SSL, так как на самом деле трафик между сервером SSH и сервером Postgres Pro не защищён. Это не должно нести какие-то дополнительные риски, так как эти серверы работают на одном компьютере.

Чтобы настроенный таким образом туннель работал, вы должны иметь возможность подключаться к компьютеру через `ssh` под именем `joe@foo.com`, так же, как это происходит при установлении терминального подключения с помощью `ssh`.

Вы также можете настроить перенаправление портов примерно так:

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

Но в этом случае с точки зрения сервера подключение будет приходить на его сетевой адрес `foo.com`, а он по умолчанию не прослушивается (вследствие указания `listen_addresses = 'localhost'`). Обычно требуется другое поведение.

Если вам нужно «перейти» к серверу баз данных через некоторый шлюз, это можно организовать примерно так:

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

Заметьте, что в этом случае трафик между `shell.foo.com` и `db.foo.com` не будет защищён туннелем SSH. SSH предлагает довольно много вариантов конфигурации, что позволяет организовывать защиту сети разными способами. За подробностями обратитесь к документации SSH.

Подсказка

Существуют и другие приложения, которые могут создавать безопасные туннели, применяя по сути тот же подход, что был описан выше.

17.12. Регистрация журнала событий в Windows

Чтобы зарегистрировать библиотеку журнала событий в Windows, выполните следующую команду:

```
regsvr32 каталог_библиотек_pgsql/pgevent.dll
```

При этом в реестре будут созданы необходимые записи для средства просмотра событий, относящиеся к источнику событий по умолчанию с именем `Postgres Pro`.

Чтобы задать другое имя источника событий (см. event_source), укажите ключи `/n` и `/i`:

```
regsvr32 /n /i:имя_источника_событий каталог_библиотек_pgsql/pgevent.dll
```

Чтобы разрегистрировать библиотеку журнала событий в операционной системе, выполните команду:

```
regsvr32 /u [/i:имя_источника_событий] каталог_библиотек_pgsql/pgevent.dll
```

Примечание

Чтобы сервер баз данных записывал сообщения в журнал событий, добавьте `eventlog` в параметр log_destination в `postgresql.conf`.

Глава 18. Конфигурация сервера

На работу системы баз данных оказывают влияние множество параметров конфигурации. В первом разделе этой главы рассказывается, как управлять этими параметрами, а в последующих разделах подробно описывается каждый из них.

18.1. Изменение параметров

18.1.1. Имена и значения параметров

Имена всех параметров являются регистронезависимыми. Каждый параметр принимает значение одного из пяти типов: логический, строка, целое, число с плавающей точкой или перечисление. От типа значения зависит синтаксис установки этого параметра:

- *Логический:* Значения могут задаваться строками `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0` (регистр не имеет значения), либо как достаточно однозначный префикс одной из этих строк.
- *Строка:* Обычно строковое значение заключается в апострофы (при этом внутренние апострофы дублируются). Однако если значение является простым числом или идентификатором, апострофы обычно можно опустить. (Значения, совпадающие с ключевыми словами SQL, всё же требуют заключения в апострофы в некоторых контекстах.)
- *Число (целое или с плавающей точкой):* Значения числовых параметров могут задаваться в обычных форматах, принятых для целых чисел или чисел с плавающей точкой; если параметр целочисленный, дробные значения округляются до ближайшего целого. Кроме того, целочисленные параметры принимают значения в шестнадцатеричном (с префиксом `0x`) и восьмеричном (с префиксом `0`) виде, но дробная часть в таких случаях исключена. Разделители разрядов в значениях использовать нельзя. Заключать в кавычки требуется только значения в шестнадцатеричном виде.
- *Число с единицей измерения:* Некоторые числовые параметры задаются с единицами измерения, так как они описывают количества информации или времени. Единицами могут быть байты, килобайты, блоки (обычно восемь килобайт), миллисекунды, секунды или минуты. При указании только числового значения для такого параметра единицей измерения будет считаться установленная для него единица по умолчанию, которая указывается в `pg_settings.unit`. Для удобства параметры также можно задавать, указывая единицу измерения явно, например, задать `'120 ms'` для значения времени. При этом такое значение будет переведено в основную единицу измерения параметра. Заметьте, что для этого значение должно записываться в виде строки (в апострофах). Имя единицы является регистронезависимым, а между ним и числом допускаются пробельные символы.
  - Допустимые единицы информации: `B` (байты), `kB` (килобайты), `MB` (мегабайты), `GB` (гигабайты) и `TB` (терабайты). Множителем единиц информации считается 1024, не 1000.
  - Допустимые единицы времени: `us` (микросекунды), `ms` (миллисекунды), `s` (секунды), `min` (минуты), `h` (часы) и `d` (дни).

  Если с единицей измерения задаётся дробное значение, оно будет округлено до следующей меньшей единицы, если такая имеется. Например, значение `30.1 GB` будет преобразовано в `30822 MB`, а не в `32319628902 B`. Если параметр имеет целочисленный тип, после преобразования единицы измерения значение окончательно округляется до целого.
- *Перечисление:* Параметры, имеющие тип перечисление, записываются так же, как строковые параметры, но могут иметь только ограниченный набор значений. Список допустимых значений такого параметра задаётся в `pg_settings.enumvals`. В значениях перечислений регистр не учитывается.

18.1.2. Определение параметров в файле конфигурации

Самый основной способ установки этих параметров — определение их значений в файле `postgresql.conf`, который обычно находится в каталоге данных. При инициализации каталога кластера БД в этот каталог помещается копия стандартного файла. Например, он может выглядеть так:

```
# Это комментарий
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Каждый параметр определяется в отдельной строке. Знак равенства в ней между именем и значением является необязательным. Пробельные символы в строке не играют роли (кроме значений, заключённых в апострофы), а пустые строки игнорируются. Знаки решётки (`#`) обозначают продолжение строки как комментарий. Значения параметров, не являющиеся простыми идентификаторами или числами, должны заключаться в апострофы. Чтобы включить в такое значение собственно апостроф, его следует продублировать (предпочтительнее) или предварить обратной косой чертой. Если один и тот же параметр определяется в файле конфигурации неоднократно, действовать будет только последнее определение, остальные игнорируются.

Параметры, установленные таким образом, задают значения по умолчанию для данного кластера. Эти значения будут действовать в активных сеансах, если не будут переопределены. В следующих разделах описывается, как их может переопределить администратор или пользователь.

Основной процесс сервера перечитывает файл конфигурации заново, получая сигнал SIGHUP; послать его проще всего можно, запустив `pg_ctl reload` в командной строке или вызвав SQL-функцию `pg_reload_conf()`. Основной процесс сервера передаёт этот сигнал всем остальным запущенным серверным процессам, так что существующие сеансы тоже получают новые значения (после того, как завершится выполнение текущей команды клиента). Также возможно послать этот сигнал напрямую одному из серверных процессов. Учтите, что некоторые параметры можно установить только при запуске сервера; любые изменения их значений в файле конфигурации не будут учитываться до перезапуска сервера. Более того, при обработке SIGHUP игнорируются неверные значения параметров (но об этом сообщается в журнале).

В дополнение к `postgresql.conf` в каталоге данных Postgres Pro содержится файл `postgresql.auto.conf`, который имеет тот же формат, что и `postgresql.conf`, но предназначен для автоматического изменения, а не для редактирования вручную. Этот файл содержит параметры, задаваемые командой `ALTER SYSTEM`. Он считывается одновременно с `postgresql.conf` и заданные в нём параметры действуют таким же образом. Параметры в `postgresql.auto.conf` переопределяют те, что указаны в `postgresql.conf`.

Вносить изменения в `postgresql.auto.conf` можно и с использованием внешних средств. Однако это не рекомендуется делать на работающем сервере, если параметр конфигурации allow_alter_system имеет значение, отличное от `off`, так эти изменения могут быть потеряны при параллельном выполнении команды `ALTER SYSTEM`. Внешние программы могут просто добавлять новые определения параметров в конец файла или удалять повторяющиеся определения и/или комментарии (как делает `ALTER SYSTEM`).

Системное представление `pg_file_settings` может быть полезным для предварительной проверки изменений в файлах конфигурации или для диагностики проблем, если сигнал SIGHUP не даёт желаемого эффекта.

18.1.3. Управление параметрами через SQL

В Postgres Pro есть три SQL-команды, задающие для параметров значения по умолчанию. Уже упомянутая команда `ALTER SYSTEM` даёт возможность изменять глобальные значения средствами SQL; она функционально равнозначна редактированию `postgresql.conf`. Кроме того, есть ещё две команды, которые позволяют задавать значения по умолчанию на уровне баз данных и ролей:

- Команда `ALTER DATABASE` позволяет переопределить глобальные параметры на уровне базы данных.

- Команда `ALTER ROLE` позволяет переопределить для конкретного пользователя как глобальные, так и локальные для базы данных параметры.

Значения, установленные командами `ALTER DATABASE` и `ALTER ROLE`, применяются только при новом подключении к базе данных. Они переопределяют значения, полученные из файлов конфигурации или командной строки сервера, и применяются по умолчанию в рамках сеанса. Заметьте, что некоторые параметры невозможно изменить после запуска сервера, поэтому их нельзя установить этими командами (или командами, перечисленными ниже).

Когда клиент подключён к базе данных, он может воспользоваться двумя дополнительными командами SQL (и равнозначными функциями), которые предоставляет Postgres Pro для управления параметрами конфигурации:

- Команда `SHOW` позволяет узнать текущее значение любого параметра. Ей соответствует SQL-функция `current_setting(setting_name text)` (см. Подраздел 9.28.1).
- Команда `SET` позволяет изменить текущее значение тех параметров, которые устанавливаются локально в рамках сеанса; на другие сеансы она не влияет. Многие параметры могут быть установлены таким образом любым пользователем, а некоторые — только суперпользователями и пользователями, которым предоставлено право `SET` для этого параметра. Ей соответствует SQL-функция `set_config(setting_name, new_value, is_local)` (см. Подраздел 9.28.1).

Кроме того, просмотреть и изменить значения параметров для текущего сеанса можно в системном представлении `pg_settings`:

- Запрос на чтение представления выдаёт ту же информацию, что и `SHOW ALL`, но более подробно. Этот подход и более гибкий, так как в нём можно указать условия фильтра или связать результат с другими отношениями.
- Выполнение `UPDATE` для этого представления, а именно присваивание значения столбцу `setting`, равносильно выполнению команды `SET`. Например, команде

  ```
  SET configuration_parameter TO DEFAULT;
  ```

  равнозначен запрос:

  ```
  UPDATE pg_settings SET setting = reset_val WHERE name = 'configuration_parameter';
  ```

18.1.4. Управление параметрами в командной строке

Помимо изменения глобальных значений по умолчанию и переопределения их на уровне базы данных или роли, параметры Postgres Pro можно изменить, используя средства командной строки. Управление через командную строку поддерживают и сервер, и клиентская библиотека libpq.

- При запуске сервера, значения параметров можно передать команде `postgres` в аргументе командной строки `-c name=value` или равнозначном варианте `--name=value`. Например:

  ```
  postgres -c log_connections=yes --log-destination='syslog'
  ```

  Параметры, заданные таким образом, переопределяют те, что были установлены в `postgresql.conf` или командой `ALTER SYSTEM`, так что их нельзя изменить глобально без перезапуска сервера.
- При запуске клиентского сеанса, использующего libpq, значения параметров можно указать в переменной окружения `PGOPTIONS`. Заданные таким образом параметры будут определять значения по умолчанию на время сеанса, но никак не влияют на другие сеансы. По историческим причинам формат `PGOPTIONS` похож на тот, что применяется при запуске команды `postgres`; в частности, в нём должен присутствовать флаг `-c` или перед именем должно быть указано `--`. Например:

  ```
  env PGOPTIONS="-c geqo=off --statement-timeout=5min" psql
  ```

  Другие клиенты и библиотеки могут иметь собственные механизмы управления параметрами, через командную строку или как-то иначе, используя которые пользователь сможет менять параметры сеанса, не выполняя непосредственно команды SQL.

18.1.5. Упорядочение содержимого файлов конфигурации

Postgres Pro предоставляет несколько возможностей для разделения сложных файлов `postgresql.conf` на вложенные файлы. Эти возможности особенно полезны при управлении множеством серверов с похожими, но не одинаковыми конфигурациями.

Помимо присваиваний значений параметров, `postgresql.conf` может содержать *директивы включения* файлов, которые будут прочитаны и обработаны, как если бы их содержимое было вставлено в данном месте файла конфигурации. Это позволяет разбивать файл конфигурации на физически отдельные части. Директивы включения записываются просто:

```
include 'имя_файла'
```

Если имя файла задаётся не абсолютным путём, оно рассматривается относительно каталога, в котором находится включающий файл конфигурации. Включения файлов могут быть вложенными.

Кроме того, есть директива `include_if_exists`, которая работает подобно `include`, за исключением случаев, когда включаемый файл не существует или не может быть прочитан. Обычная директива `include` считает это критической ошибкой, но `include_if_exists` просто выводит сообщение и продолжает обрабатывать текущий файл конфигурации.

Файл `postgresql.conf` может также содержать директивы `include_dir`, позволяющие подключать целые каталоги с файлами конфигурации. Они записываются так:

```
include_dir 'каталог'
```

Имена, заданные не абсолютным путём, рассматриваются относительно каталога, содержащего текущий файл конфигурации. В заданном каталоге включению подлежат только файлы с именами, оканчивающимися на `.conf`. При этом файлы с именами, начинающимися с «`.`», тоже игнорируются, для предотвращения ошибок, так как они считаются скрытыми в ряде систем. Набор файлов во включаемом каталоге обрабатывается по порядку имён (определяемому правилами, принятыми в C, т. е. цифры идут перед буквами, а буквы в верхнем регистре — перед буквами в нижнем).

Включение файлов или каталогов позволяет разделить конфигурацию базы данных на логические части, а не вести один большой файл `postgresql.conf`. Например, представьте, что в некоторой компании есть два сервера баз данных, с разным объёмом ОЗУ. Скорее всего при этом их конфигурации будут иметь общие элементы, например, параметры ведения журналов. Но параметры, связанные с памятью, у них будут различаться. Кроме того, другие параметры могут быть специфическими для каждого сервера. Один из вариантов эффективного управления такими конфигурациями — разделить изменения стандартной конфигурации на три файла. Чтобы подключить эти файлы, можно добавить в конец файла `postgresql.conf` следующие директивы:

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

Общие для всех серверов параметры будут помещаться в `shared.conf`. Файл `memory.conf` может иметь два варианта — первый для серверов с 8ГБ ОЗУ, а второй для серверов с 16 ГБ. Наконец, `server.conf` может содержать действительно специфические параметры для каждого отдельного сервера.

Также возможно создать каталог с файлами конфигурации и поместить туда все эти файлы. Например, так можно подключить каталог `conf.d` в конце `postgresql.conf`:

```
include_dir 'conf.d'
```

Затем можно дать файлам в каталоге `conf.d` следующие имена:

```
00shared.conf
01memory.conf
02server.conf
```

Такое именование устанавливает чёткий порядок подключения этих файлов, что важно, так как если параметр определяется несколько раз в разных файлах конфигурации, действовать будет последнее определение. В рамках данного примера, установленное в `conf.d/02server.conf` значение переопределит значение того же параметра, заданное в `conf.d/01memory.conf`.

Вы можете применить этот подход и с описательными именами файлов:

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

При таком упорядочивании каждому варианту файла конфигурации даётся уникальное имя. Это помогает исключить конфликты, если конфигурации разных серверов нужно хранить в одном месте, например, в репозитории системы управления версиями. (Кстати, хранение файлов конфигурации в системе управления версиями — это ещё один эффективный приём, который стоит применять.)

18.2. Расположения файлов

В дополнение к вышеупомянутому `postgresql.conf`, Postgres Pro обрабатывает два редактируемых вручную файла конфигурации, в которых настраивается аутентификация клиентов (их использование рассматривается в Главе 19). По умолчанию все три файла конфигурации размещаются в каталоге данных кластера БД. Параметры, описанные в этом разделе, позволяют разместить их и в любом другом месте. (Это позволяет упростить администрирование, в частности, выполнять резервное копирование этих файлов обычно проще, когда они хранятся отдельно.)

`data_directory` (`string`)

Задаёт каталог, в котором хранятся данные. Этот параметр можно задать только при запуске сервера.

`config_file` (`string`)

Задаёт основной файл конфигурации сервера (его стандартное имя — `postgresql.conf`). Этот параметр можно задать только в командной строке `postgres`.

`hba_file` (`string`)

Задаёт файл конфигурации для аутентификации по сетевым узлам (его стандартное имя — `pg_hba.conf`). Этот параметр можно задать только при запуске сервера.

`ident_file` (`string`)

Задаёт файл конфигурации для сопоставлений имён пользователей (его стандартное имя — `pg_ident.conf`). Этот параметр можно задать только при запуске сервера. См. также Раздел 19.2.

`external_pid_file` (`string`)

Задаёт имя дополнительного файла с идентификатором процесса (PID), который будет создавать сервер для использования программами администрирования. Этот параметр можно задать только при запуске сервера.

При стандартной установке ни один из этих параметров не задаётся явно. Вместо них задаётся только каталог данных, аргументом командной строки `-D` или переменной окружения `PGDATA`, и все необходимые файлы конфигурации загружаются из этого каталога.

Если вы хотите разместить файлы конфигурации не в каталоге данных, то аргумент командной строки `postgres -D` или переменная окружения `PGDATA` должны указывать на каталог, содержащий файлы конфигурации, а в `postgresql.conf` (или в командной строке) должен задаваться параметр `data_directory`, указывающий, где фактически находятся данные. Учтите, что `data_directory` переопределяет путь, задаваемый в `-D` или `PGDATA` как путь каталога данных, но не расположение файлов конфигурации.

При желании вы можете задать имена и пути файлов конфигурации по отдельности, воспользовавшись параметрами `config_file`, `hba_file` и/или `ident_file`. Параметр `config_file` можно задать только в командной строке `postgres`, тогда как остальные можно задать и в основном файле конфигурации. Если явно заданы все три эти параметра плюс `data_directory`, то задавать `-D` или `PGDATA` не нужно.

Во всех этих параметрах относительный путь должен задаваться от каталога, в котором запускается `postgres`.

18.3. Подключения и аутентификация

18.3.1. Параметры подключений

`listen_addresses` (`string`)

Задаёт адреса TCP/IP, по которым сервер будет принимать подключения клиентских приложений. Это значение принимает форму списка, разделённого запятыми, из имён и/или числовых IP-адресов узлов. Особый элемент, `*`, обозначает все имеющиеся IP-интерфейсы. Запись `0.0.0.0` позволяет задействовать все адреса IPv4, а `::` — все адреса IPv6. Если список пуст, сервер не будет привязываться ни к какому IP-интерфейсу, а значит, подключиться к нему можно будет только через Unix-сокеты. При непустом списке сервер запустится, если сможет принимать подключения хотя бы по одному адресу TCP/IP. Предупреждение будет выдано для любого адреса TCP/IP, который невозможно открыть. По умолчанию этот параметр содержит localhost, что допускает подключение к серверу по TCP/IP только через локальный интерфейс «замыкания».

Хотя механизм аутентификации клиентов (см. Главу 19) позволяет гибко управлять доступом пользователей к серверу, параметр `listen_addresses` может ограничить интерфейсы, через которые будут приниматься соединения, что бывает полезно для предотвращения злонамеренных попыток подключения через незащищённые сетевые интерфейсы. Этот параметр можно задать только при запуске сервера.

`port` (`integer`)

TCP-порт, открываемый сервером; по умолчанию, 5432. Заметьте, что этот порт используется для всех IP-адресов, через которые сервер принимает подключения. Этот параметр можно задать только при запуске сервера.

`max_connections` (`integer`)

Определяет максимальное число одновременных подключений к серверу БД. По умолчанию обычно это 100 подключений, но это число может быть меньше, если ядро накладывает свои ограничения (это определяется в процессе initdb). Этот параметр можно задать только при запуске сервера.

Postgres Pro определяет размер некоторых ресурсов в зависимости от значения `max_connections`. Увеличение этого значения приводит к выделению большего объёма этих ресурсов, включая общую память.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

`reserved_connections` (`integer`)

Определяет количество «слотов» подключений, которые PostgreSQL будет резервировать для ролей с правами роли `pg_use_reserved_connections`. Когда число активных одновременных подключений больше superuser_reserved_connections, но меньше суммы `superuser_reserved_connections` и `reserved_connections`, принимаются только подключения суперпользователей и ролей с правами `pg_use_reserved_connections`. Если количество подключений меньше или равно `superuser_reserved_connections`, принимаются только подключения суперпользователей.

По умолчанию подключения не резервируются (параметр имеет нулевое значение). Это значение должно быть меньше значения `max_connections` минус `superuser_reserved_connections`. Задать этот параметр можно только при запуске сервера.

`superuser_reserved_connections` (`integer`)

Определяет количество «слотов» подключений, которые Postgres Pro будет резервировать для суперпользователей. При этом всего одновременно активными могут быть максимум max_connections подключений. Когда число активных одновременных подключений больше или равно `max_connections` минус `superuser_reserved_connections`, принимаются только подключения суперпользователей. Слоты подключения, зарезервированные этим параметром, предназначены для использования в экстренных случаях, когда закончатся слоты, зарезервированные reserved_connections.

По умолчанию резервируются три соединения. Это значение должно быть меньше значения `max_connections` минус `reserved_connections`. Задать этот параметр можно только при запуске сервера.

`unix_socket_directories` (`string`)

Задаёт каталог Unix-сокета, через который сервер будет принимать подключения клиентских приложений. Создать несколько сокетов можно, перечислив в этом значении несколько каталогов через запятую. Пробелы между элементами этого списка игнорируются; если в пути каталога содержатся пробелы, его нужно заключать в двойные кавычки. При пустом значении сервер не будет работать с Unix-сокетами, в этом случае к нему можно будет подключиться только по TCP/IP.

Значение, которое начинается с `@`, указывает, что будет создан сокет домена Unix в абстрактном пространстве имён (в настоящее время поддерживается только в Linux). Такое значение задаёт не «каталог», а префикс, из которого получается фактическое имя сокета таким же образом, как и для пространства имён файловой системы. Хотя префикс имени абстрактного сокета может быть произвольным, поскольку он не задаёт путь в файловой системе, принято использовать значения, подобные путям в файловой системе, такие как `@/tmp`.

Значение по умолчанию обычно `/tmp`, но оно может быть изменено во время сборки. В Windows по умолчанию это значение не задано, то есть сокет домена Unix не создаётся. Этот параметр можно установить только при запуске сервера.

Помимо самого файла сокета, который называется `.s.PGSQL.`*`nnnn`* (где *`nnnn`* — номер порта сервера), в каждом каталоге `unix_socket_directories` создаётся обычный файл `.s.PGSQL.`*`nnnn`*`.lock`. Ни в коем случае не удаляйте эти файлы вручную. Для сокетов в абстрактном пространстве имён файл `.lock` не создаётся.

`unix_socket_group` (`string`)

Задаёт группу-владельца Unix-сокетов. (Пользователем-владельцем сокетов всегда будет пользователь, запускающий сервер.) В сочетании с `unix_socket_permissions` данный параметр можно использовать как дополнительный механизм управления доступом к Unix-сокетам. По умолчанию он содержит пустую строку, то есть группой-владельцем становится основная группа пользователя, запускающего сервер. Задать этот параметр можно только при запуске сервера.

В Windows данный параметр не поддерживается, так что любое его значение ни на что не влияет. Кроме того, сокеты в абстрактном пространстве имён не имеют владельца файла, поэтому для таких сокетов данный параметр также игнорируется.

`unix_socket_permissions` (`integer`)

Задаёт права доступа к Unix-сокетам. Для Unix-сокетов применяется обычный набор разрешений Unix. Значение параметра ожидается в числовом виде, который принимают функции `chmod` и `umask`. (Для применения обычного восьмеричного формата число должно начинаться с `0` (нуля).)

По умолчанию действуют разрешения `0777`, при которых подключаться к сокету могут все. Другие разумные варианты — `0770` (доступ имеет только пользователь и группа, см. также `unix_socket_group`) и `0700` (только пользователь). (Заметьте, что для Unix-сокетов требуется только право на запись, так что добавлять или отзывать права на чтение/выполнение не имеет смысла.)

Этот механизм управления доступом не зависит от описанного в Главе 19.

Этот параметр можно задать только при запуске сервера.

Данный параметр неприменим для некоторых систем, в частности, Solaris (а именно Solaris 10), которые полностью игнорируют разрешения для сокетов. В таких системах примерно тот же эффект можно получить, указав в параметре `unix_socket_directories` каталог, доступ к которому ограничен должным образом.

Сокеты в абстрактном пространстве имён не имеют прав доступа к файлам, поэтому для таких сокетов данный параметр также игнорируется.

`bonjour` (`boolean`)

Включает объявления о существовании сервера посредством Bonjour. По умолчанию выключен. Задать этот параметр можно только при запуске сервера.

`bonjour_name` (`string`)

Задаёт имя службы в среде Bonjour. Если значение этого параметра — пустая строка (`''`) (это значение по умолчанию), в качестве этого имени используется имя компьютера. Этот параметр игнорируется, если сервер был скомпилирован без поддержки Bonjour. Задать этот параметр можно только при запуске сервера.

18.3.2. Параметры TCP

`tcp_keepalives_idle` (`integer`)

Задаёт период отсутствия сетевой активности, по истечении которого операционная система должна отправить клиенту TCP-сигнал сохранения соединения. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. При значении 0 (по умолчанию) действует величина, принятая по умолчанию в операционной системе. В Windows при присваивании нулевого значения этот период устанавливается равным 2 часам, так как Windows не позволяет прочитать системное значение по умолчанию. Этот параметр поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPIDLE` или равнозначный, а также в Windows; в других системах он должен быть равен нулю. В сеансах, подключённых через Unix-сокеты, он игнорируется и всегда считается равным 0.

`tcp_keepalives_interval` (`integer`)

Задаёт интервал, по истечении которого следует повторять TCP-сигнал сохранения соединения, если от клиента не получено подтверждение предыдущего сигнала. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. При значении 0 (по умолчанию) действует величина, принятая по умолчанию в операционной системе. В Windows при нулевом значении этот интервал устанавливается равным 1 секунде, так как Windows не позволяет прочитать системное значение по умолчанию. Этот параметр поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPINTVL` или равнозначный, а также в Windows; в других системах он должен быть равен нулю. В сеансах, подключённых через Unix-сокеты, он игнорируется и всегда считается равным 0.

`tcp_keepalives_count` (`integer`)

Задаёт число TCP-сигналов сохранения соединения, которые могут быть потеряны до того, как соединение сервера с клиентом будет признано прерванным. При значении 0 (по умолчанию) действует величина, принятая по умолчанию в операционной системе. Этот параметр поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPCNT` или равнозначный (Windows не входит в их число); в других системах он должен быть равен нулю. В сеансах, подключённых через Unix-сокеты, он игнорируется и всегда считается равным 0.

`tcp_user_timeout` (integer)

Задаёт интервал, в течение которого переданные данные могут оставаться неподтверждёнными, прежде чем будет принято решение о принудительном закрытии TCP-соединения. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. При значении 0 (по умолчанию) действует величина, принятая по умолчанию в операционной системе. Этот параметр поддерживается только в системах, воспринимающих параметр сокета `TCP_USER_TIMEOUT` (Windows не входит в их число); в других системах он должен быть равен нулю. В сеансах, подключённых через доменные сокеты Unix, он игнорируется и всегда считается равным 0.

`client_connection_check_interval` (integer)

Задаёт интервал между необязательными проверками соединения сервера с клиентом в процессе выполнения запроса. Такая проверка, заключающаяся в опросе состояния сокета, позволяет раньше прерывать длительные запросы, если ядро сообщает, что соединение закрыто.

Эта функциональность зависит от событий ядра, имеющихся только в Linux, macOS, illumos и операционных системах семейства BSD, и в настоящее время недоступна в других системах.

Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию — `0`, то есть проверка соединения отключена. Если соединение не проверяется, сервер обнаружит разрыв соединения только при следующем обращении к сокету, когда будет передавать, ожидать или читать данные.

Чтобы само ядро могло надёжно и в течение известного периода времени обнаруживать потерянные TCP-соединения во всех сценариях, включая сбой сети, также может потребоваться изменить параметры сохранения TCP-соединения в операционной системе или параметры Postgres Pro tcp_keepalives_idle, tcp_keepalives_interval и tcp_keepalives_count.

18.3.3. Аутентификация

`authentication_timeout` (integer)

Максимальное время, за которое должна произойти аутентификация. Если потенциальный клиент не сможет пройти проверку подлинности за это время, сервер закроет соединение. Благодаря этому зависшие при подключении клиенты не будут занимать соединения неограниченно долго. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение этого параметра по умолчанию — одна минута (`1m`). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`password_encryption` (enum)

Когда в CREATE ROLE или ALTER ROLE задаётся пароль, этот параметр определяет, каким алгоритмом его шифровать. Возможные значения данного параметра — `scram-sha-256` (пароль будет шифроваться алгоритмом SCRAM-SHA-256) и `md5` (пароль сохраняется в виде хеша MD5). Значение по умолчанию — `scram-sha-256`.

Учтите, что старые клиенты могут не поддерживать механизм проверки подлинности SCRAM и поэтому не будут работать с паролями, зашифрованными алгоритмом SCRAM-SHA-256. За подробностями обратитесь к Разделу 19.5.

`scram_iterations` (integer)

Количество вычислительных итераций при шифровании пароля с использованием SCRAM-SHA-256. Значение по умолчанию — `4096`. Большее количество итераций обеспечивает дополнительную защиту сохранённых паролей от атак полным перебором, но замедляет аутентификацию. Изменение значения не влияет на существующие пароли, зашифрованные алгоритмом SCRAM-SHA-256, поскольку количество итераций фиксируется во время шифрования. Чтобы использовать изменённое значение, необходимо задать новый пароль.

`krb_server_keyfile` (`string`)

Задаёт расположение файла ключей Kerberos для данного сервера. Значение по умолчанию: `FILE:/usr/local/pgsql/etc/krb5.keytab` (каталог определяется значением параметра `sysconfdir` в процессе сборки; чтобы его узнать, выполните `pg_config --sysconfdir`). Если этот параметр содержит пустую строку, используется значение по умолчанию, зависящее от системы. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. За подробностями обратитесь к Разделу 19.6.

`krb_caseins_users` (`boolean`)

Определяет, должны ли имена пользователей GSSAPI обрабатываться без учёта регистра. По умолчанию значение этого параметра — `off` (регистр учитывается). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`gss_accept_delegation` (`boolean`)

Определяет, должно ли приниматься делегирование учётных данных GSSAPI от клиента. По умолчанию значение этого параметра — `off`, то есть учётные данные клиента *не* принимаются. Изменение этого параметра на `on` заставит сервер принимать учётные данные, переданные ему от клиента. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

18.3.4. SSL

За дополнительной информацией о настройке SSL обратитесь к Разделу 17.9. Параметры конфигурации для управления шифрованием передачи с использованием протоколов TLS называются `ssl` по историческим причинам, несмотря на то, что протокол SSL признан устаревшим. SSL в этом контексте используется взаимозаменяемо с TLS.

`ssl` (`boolean`)

Разрешает подключения SSL. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `off`.

`ssl_ca_file` (`string`)

Задаёт имя файла, содержащего сертификаты центров сертификации (ЦС) для SSL-сервера. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию этот параметр пуст, что означает, что файл сертификатов ЦС не загружается и проверка клиентских сертификатов не производится.

`ssl_cert_file` (`string`)

Задаёт имя файла, содержащего сертификат этого SSL-сервера. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `server.crt`.

`ssl_crl_file` (`string`)

Задаёт имя файла, содержащего список отзыва клиентских сертификатов (CRL, Certificate Revocation List) для SSL. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию этот параметр пуст, что означает, что файл CRL не загружается (если не задан параметр ssl_crl_dir).

`ssl_crl_dir` (`string`)

Задаёт имя каталога, содержащего список отзыва клиентских сертификатов (CRL, Certificate Revocation List) для SSL. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию этот параметр пуст, что означает, что файл CRL не загружается (если не задан параметр ssl_crl_file).

Каталог необходимо подготовить с помощью команды OpenSSL `openssl rehash` или `c_rehash`. За подробностями обратитесь к документации OpenSSL.

Когда применяется этот параметр, списки отзыва сертификатов (Certificate Revocation List, CRL) в указанном каталоге загружаются при необходимости во время подключения. Новые CRL в этом каталоге будут использоваться сразу по мере добавления. Это отличает данный параметр от ssl_crl_file, с которым CRL загружаются из файла во время запуска сервера или при перезагрузке конфигурации. Эти параметры можно использовать вместе.

`ssl_key_file` (`string`)

Задаёт имя файла, содержащего закрытый ключ SSL-сервера. При указании относительного пути он рассматривается от каталога данных. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `server.key`.

`ssl_ciphers` (`string`)

Задаёт список наборов шифров SSL, которые могут применяться для SSL-соединений. Синтаксис этого параметра и список поддерживаемых значений можно найти на странице ciphers руководства по OpenSSL. Этот параметр действует только для подключений TLS версии 1.2 и ниже. Для подключений TLS версии 1.3 возможность выбора шифров в настоящее время отсутствует. Значение по умолчанию — `HIGH:MEDIUM:+3DES:!aNULL`. Обычно оно вполне приемлемо при отсутствии особых требований по безопасности.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Объяснение значения по умолчанию:

`HIGH`

Наборы шифров, в которых используются шифры из группы высокого уровня (`HIGH`), (например: AES, Camellia, 3DES)

`MEDIUM`

Наборы шифров, в которых используются шифры из группы среднего уровня (`MEDIUM`) (например, RC4, SEED)

`+3DES`

Порядок шифров для группы `HIGH` по умолчанию в OpenSSL определён некорректно. В нём 3DES оказывается выше AES128, что неправильно, так как он считается менее безопасным, чем AES128, и работает гораздо медленнее. Включение `+3DES` меняет этот порядок, чтобы данный алгоритм следовал после всех шифров групп `HIGH` и `MEDIUM`.

`!aNULL`

Отключает наборы анонимных шифров, не требующие проверки подлинности. Такие наборы уязвимы для атак с посредником (MITM), поэтому использовать их не следует.

Конкретные наборы шифров и их свойства очень различаются от версии к версии OpenSSL. Чтобы получить фактическую информацию о них для текущей установленной версии OpenSSL, выполните команду `openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'`. Учтите, что этот список фильтруется во время выполнения, в зависимости от типа ключа сервера.

`ssl_prefer_server_ciphers` (boolean)

Определяет, должны ли шифры SSL сервера предпочитаться клиентским. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `on`.

До версии Postgres Pro 9.4 этот параметр отсутствовал и предпочтение отдавалось выбору клиента. Введён этот параметр в основном для обеспечения совместимости с этими версиями. Обычно лучше использовать конфигурацию сервера, так как в конфигурации на стороне клиента более вероятны ошибки.

`ssl_ecdh_curve` (string)

Задаёт имя кривой для использования при обмене ключами ECDH. Эту кривую должны поддерживать все подключающиеся клиенты. Это не обязательно должна быть кривая, с которой был получен ключ сервера. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `prime256v1`.

Наиболее распространённые кривые в OpenSSL — `prime256v1` (NIST P-256), `secp384r1` (NIST P-384), `secp521r1` (NIST P-521). Полный список доступных кривых можно получить командой `openssl ecparam -list_curves`. Однако не все из них пригодны для TLS.

`ssl_min_protocol_version` (enum)

Задаёт минимальную версию протокола SSL/TLS, которая может использоваться. В настоящее время допускаются версии `TLSv1`, `TLSv1.1`, `TLSv1.2`, `TLSv1.3`. Старые версии библиотеки OpenSSL могут не поддерживать все варианты; при выборе неподдерживаемой версии будет выдана ошибка. Версии протокола до TLS 1.0, а именно SSL v.2 и v.3, не будут использоваться в любом случае.

Значение по умолчанию — `TLSv1.2`, что соответствует рекомендациям, актуальным в индустрии на момент написания этой документации.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`ssl_max_protocol_version` (enum)

Задаёт максимальную версию протокола SSL/TLS, которая может использоваться. Допускаются те же версии, что и для ssl_min_protocol_version, а также пустая строка, обозначающая отсутствие ограничения версии. По умолчанию версии не ограничиваются. Устанавливать максимальную возможную версию протокола полезно прежде всего для тестирования или в случае проблем при использовании нового протокола какими-то компонентами.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`ssl_dh_params_file` (string)

Задаёт имя файла с параметрами алгоритма Диффи-Хеллмана, применяемого для так называемого эфемерного семейства DH шифров SSL. По умолчанию значение пустое, то есть используются стандартные параметры DH, заданные при компиляции. Использование нестандартных параметров DH защищает от атаки, рассчитанной на взлом хорошо известных встроенных параметров DH. Создать собственный файл с параметрами DH можно, выполнив команду `openssl dhparam -out dhparams.pem 2048`.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`ssl_passphrase_command` (string)

Задаёт внешнюю команду, которая будет вызываться, когда потребуется пароль для расшифровывания SSL-файла, например закрытого ключа. По умолчанию этот параметр не определён, то есть пароль будет запрашиваться встроенным механизмом.

Эта команда должна вывести пароль на устройство стандартного вывода и завершиться с кодом 0. В строке параметра `%p` заменяется строкой приглашения. (Напишите `%%`, чтобы вывести `%` буквально.) Заметьте, что строка приглашения, вероятно, будет содержать пробелы, так что её нужно будет заключить в кавычки. Если в конце добавлен один перевод строки, он будет отфильтрован при выводе.

Эта команда не обязательно должна запрашивать пароль у пользователя. Она может считать его из файла, извлечь из системной связки ключей или получить другими подобными средствами. Ответственность за выбор достаточно безопасного механизма возлагается на пользователя.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`ssl_passphrase_command_supports_reload` (`boolean`)

Этот параметр определяет, будет ли заданная параметром `ssl_passphrase_command` команда, запрашивающая пароль, также вызываться при перезагрузке конфигурации, если для файла ключа требуется пароль. Когда этот параметр отключён (по умолчанию), команда `ssl_passphrase_command` будет игнорироваться при перезагрузке, и конфигурация SSL не будет обновляться, если требуется пароль. Это значение подходит для команды, требующей для ввода пароля наличия терминала TTY, который может быть недоступен в процессе работы сервера. Включение данного параметра может быть уместно, если, например, пароль считывается из файла.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

18.4. Потребление ресурсов

18.4.1. Память

`shared_buffers` (`integer`)

Задаёт объём памяти, который будет использовать сервер баз данных для буферов в разделяемой памяти. По умолчанию это обычно 128 мегабайт (`128MB`), но может быть и меньше, если конфигурация вашего ядра накладывает дополнительные ограничения (это определяется в процессе initdb). Это значение не должно быть меньше 128 килобайт. Однако для хорошей производительности обычно требуются гораздо большие значения. Если это значение задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Минимальное допустимое значение зависит от величины `BLCKSZ`. Задать этот параметр можно только при запуске сервера.

Если вы используете выделенный сервер с объёмом ОЗУ 1 ГБ и более, разумным начальным значением `shared_buffers` будет 25% от объёма памяти. Существуют варианты нагрузки, при которых эффективны будут и ещё большие значения `shared_buffers`, но так как Postgres Pro использует и кеш операционной системы, выделять для `shared_buffers` более 40% ОЗУ вряд ли будет полезно. При увеличении `shared_buffers` обычно требуется соответственно увеличить `max_wal_size`, чтобы растянуть процесс записи большого объёма новых или изменённых данных на более продолжительное время.

В системах с объёмом ОЗУ меньше 1 ГБ стоит ограничиться меньшим процентом ОЗУ, чтобы оставить достаточно места операционной системе.

`huge_pages` (`enum`)

Определяет, будут ли огромные страницы запрашиваться из основной области общей памяти. Допустимые значения: `try` (по умолчанию), `on` и `off`. Когда параметр `huge_pages` равен `try`, сервер будет пытаться запрашивать огромные страницы, но если это ему не удастся, вернётся к стандартному поведению. Со значением `on`, если получить огромные страницы не удастся, сервер не будет запущен. Со значением `off` большие страницы не будут запрашиваться. Фактическое состояние огромных страниц показывается серверной переменной huge_pages_status.

В настоящее время это поддерживается в Linux и в Windows. В других системах значение `try` просто игнорируется. В Linux эта поддержка обеспечивается, только когда параметр `shared_memory_type` имеет значение `mmap` (по умолчанию).

В результате использования огромных страниц уменьшаются таблицы страниц, и процессор тратит меньше времени на управление памятью, что приводит к увеличению быстродействия. За более подробной информацией об использовании огромных страниц в Linux обратитесь к Подразделу 17.4.5.

Огромные страницы в Windows называются большими страницами. Чтобы использовать их, необходимо дать пользователю Windows, от имени которого работает Postgres Pro, право блокировать страницы. Для назначения пользователю этого права вы можете воспользоваться средством управления групповой политикой Windows (gpedit.msc). Чтобы сервер баз данных запускался в командной строке как отдельный процесс, а не как служба Windows, приглашение командной строки должно запускаться от имени администратора или должен быть отключён механизм UAC (User Access Control, Контроль учётных записей пользователей). Когда UAC включён, в обычном командном приглашении пользователь лишается права блокировать большие страницы в памяти.

Заметьте, что этот параметр влияет только на основную область общей памяти. В операционных системах, таких как Linux, FreeBSD и Illumos огромные страницы (также называемые «суперстраницами» или «большими» страницами) могут также автоматически использоваться при обычном выделении памяти, без явного запроса со стороны Postgres Pro. В Linux это называется «прозрачными огромными страницами»(Transparent Huge Pages, THP). Известно, что это приводит к снижению быстродействия Postgres Pro в некоторых системах Linux у ряда пользователей, поэтому использовать этот механизм в настоящее время не рекомендуется (в отличие от явного использования `huge_pages`).

`huge_page_size` (`integer`)

Задаёт размер огромных страниц, если их использование включено параметром huge_pages. Значение по умолчанию — ноль (`0`). При таком значении будет использоваться размер огромных страниц, заданный в системе по умолчанию. Этот параметр можно установить только при запуске сервера.

На современных 64-битных серверных платформах доступны, в частности, следующие размеры страниц: `2MB` и `1GB` (Intel и AMD), `16MB` и `16GB` (IBM POWER), и `64kB`, `2MB`, `32MB` и `1GB` (ARM). Подробнее использование и администрирование огромных страниц освещается в Подразделе 17.4.5.

Отличные от нуля значения в настоящее время поддерживаются только в Linux.

`temp_buffers` (`integer`)

Задаёт максимальный объём памяти, выделяемой для временных буферов в каждом сеансе. Эти существующие только в рамках сеанса буферы используются исключительно для работы с временными таблицами. Если это значение задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — 8 мегабайт (`8MB`). (Если `BLCKSZ` отличен от 8 КБ, значение по умолчанию корректируется пропорционально.) Этот параметр можно изменить в отдельном сеансе, но только до первого обращения к временным таблицам; после этого изменения его значения не будут влиять на текущий сеанс.

Сеанс выделяет временные буферы по мере необходимости до достижения предела, заданного параметром `temp_buffers`. Если сеанс не задействует временные буферы, то для него хранятся только дескрипторы буферов, которые занимают около 64 байт (в количестве `temp_buffers`). Однако если буфер действительно используется, он будет дополнительно занимать 8192 байта (или `BLCKSZ` байт, в общем случае).

`max_prepared_transactions` (`integer`)

Задаёт максимальное число транзакций, которые могут одновременно находиться в «подготовленном» состоянии (см. PREPARE TRANSACTION). При нулевом значении (по умолчанию) механизм подготовленных транзакций отключается. Задать этот параметр можно только при запуске сервера.

Если использовать транзакции не планируется, этот параметр следует обнулить, чтобы не допустить непреднамеренного создания подготовленных транзакций. Если же подготовленные транзакции применяются, то `max_prepared_transactions`, вероятно, должен быть не меньше, чем max_connections, чтобы подготовить транзакцию можно было в каждом сеансе.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

`work_mem` (`integer`)

Задаёт базовый максимальный объём памяти, который будет использоваться во внутренних операциях при обработке запросов (например, для сортировки или хеш-таблиц), прежде чем будут задействованы временные файлы на диске. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Значение по умолчанию — четыре мегабайта (`4MB`). Заметьте, что в сложных запросах одновременно могут выполняться несколько операций сортировки и хеширования, и при этом примерно этот объём памяти может использоваться в каждой операции, прежде чем данные начнут вытесняться во временные файлы. Кроме того, такие операции могут выполняться одновременно в разных сеансах. Таким образом, общий объём памяти может многократно превосходить значение `work_mem`; это следует учитывать, выбирая подходящее значение. Операции сортировки используются для `ORDER BY`, `DISTINCT` и соединений слиянием. Хеш-таблицы используются при соединениях и агрегировании по хешу, мемоизации узлов, а также обработке подзапросов `IN` с применением хеша.

Операции вычисления хеша обычно более требовательны к памяти, чем равнозначные им операции сортировки. Поэтому ограничение памяти для хеш-таблиц определяется произведением `work_mem` и `hash_mem_multiplier` и может превышать обычный базовый объём `work_mem`.

Примечание

Планировщик Postgres Pro может некорректно оценить размер хеш-таблицы при операциях вычисления хеша. Если одна или несколько порций данных в хеш-таблице настолько большие, что не помещаются в выделенную память, превышая ограничение по памяти, и при этом данные в хеш-таблице невозможно поделить на большее количество порций, планировщик будет пытаться разместить каждую порцию в памяти, даже если её размер превышает упомянутое ограничение.

Таким образом, при операциях вычисления хеша для больших наборов данных потребление памяти может резко возрастать, если данные в полученных хеш-таблицах невозможно разделить на достаточное количество порций.

`hash_mem_multiplier` (`floating point`)

Используется для определения максимального объёма памяти, который может выделяться для операций с хешированием. Итоговый объём определяется произведением `work_mem` и `hash_mem_multiplier`. Значение по умолчанию равно 2.0, то есть для операций с хешированием может использоваться объём, вдвое превышающий базовое значение `work_mem`.

Значение `hash_mem_multiplier` имеет смысл увеличить, когда постоянно наблюдается вытеснение данных на диск при выполнении запросов, а прямолинейное увеличение `work_mem` приводит к дефициту памяти (обычно проявляющемуся в ошибках «нехватка памяти»). В этих случаях действующее по умолчанию значение 2.0 обычно является оптимальным при смешанной нагрузке, а значения 2.0–8.0 могут быть эффективны там, где `work_mem` уже увеличено до 40 Мбайт или более.

`maintenance_work_mem` (`integer`)

Задаёт максимальный объём памяти для операций обслуживания БД, в частности `VACUUM`, `CREATE INDEX` и `ALTER TABLE ADD FOREIGN KEY`. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Значение по умолчанию — 64 мегабайта (`64MB`). Так как в один момент времени в сеансе может выполняться только одна такая операция и обычно они не запускаются параллельно, это значение вполне может быть гораздо больше `work_mem`. Увеличение этого значения может привести к ускорению операций очистки и восстановления БД из копии.

Учтите, что когда выполняется автоочистка, этот объём может быть выделен autovacuum_max_workers раз, поэтому не стоит устанавливать значение по умолчанию слишком большим. Возможно, будет лучше управлять объёмом памяти для автоочистки отдельно, изменяя autovacuum_work_mem.

`autovacuum_work_mem` (`integer`)

Задаёт максимальный объём памяти, который будет использовать каждый рабочий процесс автоочистки. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. При действующем по умолчанию значении `-1` этот объём определяется значением maintenance_work_mem. Этот параметр не влияет на поведение команды `VACUUM`, выполняемой в других контекстах. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`vacuum_buffer_usage_limit` (`integer`)

Указывает размер *стратегии доступа к буферам*, используемой командами `VACUUM` и `ANALYZE`. Значение `0` позволит операции использовать любое количество `shared_buffers`. В противном случае допустимые размеры варьируются от `128 kB` до `16 GB`. Если указанный размер превышает 1/8 размера `shared_buffers`, он автоматически ограничивается этим значением. Значение по умолчанию: `2 MB`. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Этот параметр можно задать в любой момент. Его также можно переопределить для VACUUM и ANALYZE при передаче параметра `BUFFER_USAGE_LIMIT`. С большим значением параметра команды `VACUUM` и `ANALYZE` могут выполняться быстрее, но при слишком большом значении многие полезные страницы могут вытесняться из общих буферов.

`logical_decoding_work_mem` (`integer`)

Задаёт максимальный объём памяти, используемой для логического декодирования, после превышения которого некоторые декодированные изменения будут вымещаться на локальный диск. Тем самым ограничивается объём памяти, используемой подключениями потоковой логической репликации. По умолчанию его значение — 64 мегабайта (`64MB`). Так как каждое подключение репликации использует один буфер заданного размера, а количество таких подключений к одному серверу обычно невелико (оно ограничивается значением `max_wal_senders`), значение данного параметра вполне можно сделать достаточно большим, намного превышающим `work_mem`, чтобы минимизировать объём вымещаемых на диск декодируемых изменений.

`commit_timestamp_buffers` (`integer`)

Задаёт объём памяти, используемый для кеширования содержимого `pg_commit_ts` (см. Таблицу 63.1). Если значение указано без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — `0`, которое означает, что будет использовано `shared_buffers`/512 блоков, но не более 1024 и не менее 16 блоков. Этот параметр можно задать только при запуске сервера.

`multixact_member_buffers` (`integer`)

Задаёт объём общей памяти, используемый для кеширования содержимого `pg_multixact/members` (см. Таблицу 63.1). Если это значение задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — `32`. Задать этот параметр можно только при запуске сервера.

`multixact_offset_buffers` (`integer`)

Задаёт объём общей памяти, используемый для кеширования содержимого `pg_multixact/offsets` (см. Таблицу 63.1). Если это значение задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — `16`. Задать этот параметр можно только при запуске сервера.

`notify_buffers` (`integer`)

Задаёт объём общей памяти, используемый для кеширования содержимого `pg_notify` (см. Таблицу 63.1). Если значение указано без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — `16`. Этот параметр можно задать только при запуске сервера.

`serializable_buffers` (`integer`)

Задаёт объём общей памяти, используемый для кеширования содержимого `pg_serial` (см. Таблицу 63.1). Если это значение задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — `32`. Задать этот параметр можно только при запуске сервера.

`subtransaction_buffers` (`integer`)

Задаёт объём памяти, используемый для кеширования содержимого `pg_subtrans` (см. Таблицу 63.1). Если значение указано без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — `0`, которое означает, что будет использовано `shared_buffers`/512 блоков, но не более 1024 и не менее 16 блоков. Этот параметр можно задать только при запуске сервера.

`transaction_buffers` (`integer`)

Задаёт объём памяти, используемый для кеширования содержимого `pg_xact` (см. Таблицу 63.1). Если значение указано без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — `0`, которое означает, что будет использовано `shared_buffers`/512 блоков, но не более 1024 и не менее 16 блоков. Этот параметр можно задать только при запуске сервера.

`max_stack_depth` (`integer`)

Задаёт максимальную безопасную глубину стека для исполнителя. В идеале это значение должно равняться предельному размеру стека, ограниченному ядром (который устанавливается командой `ulimit -s` или аналогичной), за вычетом запаса примерно в мегабайт. Этот запас необходим, потому что сервер проверяет глубину стека не в каждой процедуре, а только в потенциально рекурсивных процедурах, например, при вычислении выражений. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Значение по умолчанию — два мегабайта (`2MB`) — выбрано с большим запасом, так что риск переполнения стека минимален. Но с другой стороны, его может быть недостаточно для выполнения сложных функций. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Если `max_stack_depth` будет превышать фактический предел ядра, то функция с неограниченной рекурсией сможет вызвать крах отдельного процесса сервера. В системах, где Postgres Pro может определить предел, установленный ядром, он не позволит установить для этого параметра небезопасное значение. Однако эту информацию выдают не все системы, поэтому выбирать это значение следует с осторожностью.

`shared_memory_type` (`enum`)

Выбирает механизм разделяемой памяти, используя который сервер будет работать с основной областью общей памяти, содержащей общие буферы Postgres Pro и другие общие данные. Допустимые варианты: `mmap` (для выделения анонимных блоков разделяемой памяти с помощью `mmap`), `sysv` (для выделения разделяемой памяти System V функцией `shmget`) и `windows` (для выделения разделяемой памяти в Windows). Не все варианты поддерживаются на разных платформах; первый из поддерживаемых данной платформой вариантов становится для неё вариантом по умолчанию. Применять `sysv`, который нигде не выбирается по умолчанию, вообще не рекомендуется, так как для выделения большого объёма памяти (см. Подраздел 17.4.1) обычно требуется нестандартная настройка ядра.

`dynamic_shared_memory_type` (`enum`)

Выбирает механизм динамической разделяемой памяти, который будет использоваться сервером. Допустимые варианты: `posix` (для выделения разделяемой памяти POSIX функцией `shm_open`), `sysv` (для выделения разделяемой памяти System V функцией `shmget`), `windows` (для выделения разделяемой памяти в Windows) и `mmap` (для эмуляции разделяемой памяти через отображение в память файлов, хранящихся в каталоге данных). Не все варианты поддерживаются на разных платформах; по умолчанию обычно выбирается первый из поддерживаемых данной платформой вариантов. Применять `mmap`, который нигде не выбирается по умолчанию, вообще не рекомендуется, так как операционная система может периодически записывать на диск изменённые страницы, что создаст дополнительную нагрузку; однако это может быть полезно для отладки, когда каталог `pg_dynshmem` находится на RAM-диске или когда другие механизмы разделяемой памяти недоступны.

`min_dynamic_shared_memory` (`integer`)

Задаёт объём памяти, который должен быть выделен при запуске сервера для параллельных запросов. Если выделен недостаточный объём памяти или она уже занята одновременно выполняющимися запросами, новые параллельные запросы временно запрашивают дополнительную разделяемую память у операционной системы согласно значению `dynamic_shared_memory_type` (это может приводить к замедлению вследствие накладных расходов на управление памятью). На выделение памяти в соответствии с указанием `min_dynamic_shared_memory` влияет параметр `huge_pages` в тех операционных системах, где он поддерживается; в системе, поддерживающей автоматическое управление большими страницами, их использование вероятнее всего даст положительный эффект. Значение по умолчанию — `0` (память не выделяется). Этот параметр можно установить только при запуске сервера.

18.4.2. Диск

`temp_file_limit` (`integer`)

Задаёт максимальный объём дискового пространства, который сможет использовать один процесс для временных файлов, например, при сортировке и хешировании, или для сохранения удерживаемого курсора. Транзакция, которая попытается превысить этот предел, будет отменена. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Значение `-1` (по умолчанию) означает, что предел отсутствует. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Этот параметр ограничивает общий объём, который могут занимать в момент времени все временные файлы, задействованные в данном процессе Postgres Pro. Следует отметить, что это *не* касается файлов явно создаваемых временных таблиц; ограничивается только объём временных файлов, которые создаются неявно при выполнении запросов.

`max_notify_queue_pages` (`integer`)

Задаёт максимальное количество выделенных страниц для очереди NOTIFY / LISTEN. Значение по умолчанию — 1048576. Для страниц размером 8 КБ позволяет использовать до 8 ГБ дискового пространства.

18.4.3. Использование ресурсов ядра

`max_files_per_process` (`integer`)

Задаёт максимальное число файлов, которые могут быть одновременно открыты каждым серверным подпроцессом. Значение по умолчанию — 1000 файлов. Если ядро реализует безопасное ограничение по процессам, об этом параметре можно не беспокоиться. Но на некоторых платформах (а именно, в большинстве систем BSD) ядро позволяет отдельному процессу открыть больше файлов, чем могут открыть несколько процессов одновременно. Если вы столкнётесь с ошибками «Too many open files» (Слишком много открытых файлов), попробуйте уменьшить это число. Задать этот параметр можно только при запуске сервера.

18.4.4. Задержка очистки по стоимости

Во время выполнения команд VACUUM и ANALYZE система ведёт внутренний счётчик, в котором суммирует оцениваемую стоимость различных выполняемых операций ввода-вывода. Когда накопленная стоимость превышает предел (`vacuum_cost_limit`), процесс, выполняющий эту операцию, засыпает на некоторое время (`vacuum_cost_delay`). Затем счётчик сбрасывается и процесс продолжается.

Данный подход реализован для того, чтобы администраторы могли снизить влияние этих команд на параллельную работу с базой, за счёт уменьшения нагрузки на подсистему ввода-вывода. Очень часто не имеет значения, насколько быстро выполнятся команды обслуживания (например, `VACUUM` и `ANALYZE`), но очень важно, чтобы они как можно меньше влияли на выполнение других операций с базой данных. Администраторы имеют возможность управлять этим, настраивая задержку очистки по стоимости.

По умолчанию этот режим отключён для выполняемых вручную команд `VACUUM`. Чтобы включить его, нужно установить в `vacuum_cost_delay` ненулевое значение.

`vacuum_cost_delay` (`floating point`)

Продолжительность времени, в течение которого будет простаивать процесс, превысивший предел стоимости. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию равно нулю, то есть задержка очистки отсутствует. При положительных значениях интенсивность очистки будет зависеть от стоимости.

При настройке интенсивности очистки для `vacuum_cost_delay` обычно выбираются довольно небольшие значения, вплоть до 1 миллисекунды и меньше. Хотя в `vacuum_cost_delay` можно задавать дробные значения в миллисекундах, такие задержки могут быть неточными на старых платформах. На таких платформах для увеличения интенсивности `VACUUM` по сравнению с уровнем, обеспечиваемым при задержке 1 мс, потребуется настраивать другие параметры стоимости очистки. Тем не менее имеет смысл выбирать настолько малую задержку `vacuum_cost_delay`, насколько может обеспечить платформа; большие задержки не будут полезны.

`vacuum_cost_page_hit` (`integer`)

Примерная стоимость очистки буфера, оказавшегося в общем кеше. Это подразумевает блокировку пула буферов, поиск в хеш-таблице и сканирование содержимого страницы. По умолчанию этот параметр равен одному.

`vacuum_cost_page_miss` (`integer`)

Примерная стоимость очистки буфера, который нужно прочитать с диска. Это подразумевает блокировку пула буферов, поиск в хеш-таблице, чтение требуемого блока с диска и сканирование его содержимого. По умолчанию этот параметр равен 2.

`vacuum_cost_page_dirty` (`integer`)

Примерная стоимость очистки, при которой изменяется блок, не модифицированный ранее. В неё включается дополнительная стоимость ввода-вывода, связанная с записью изменённого блока на диск. По умолчанию этот параметр равен 20.

`vacuum_cost_limit` (`integer`)

Общая стоимость, при накоплении которой процесс очистки будет засыпать на время, указанное в `vacuum_cost_delay`. По умолчанию — 200.

Примечание

Некоторые операции устанавливают критические блокировки и поэтому должны завершаться как можно быстрее. Во время таких операций задержка очистки по стоимости не осуществляется, так что накопленная за это время стоимость может намного превышать установленный предел. Во избежание ненужных длительных задержек в таких случаях, фактическая задержка вычисляется по формуле `vacuum_cost_delay` * `accumulated_balance` / `vacuum_cost_limit` и ограничивается максимумом, равным `vacuum_cost_delay` * 4.

18.4.5. Фоновая запись

В числе специальных процессов сервера есть процесс *фоновой записи*, задача которого — осуществлять запись «грязных» (новых или изменённых) общих буферов на диск. Когда количество чистых общих буферов считается недостаточным, данный процесс записывает грязные буферы в файловую систему и помечает их как чистые. Это снижает вероятность того, что серверные процессы, выполняющие запросы пользователей, не смогут найти чистые буферы и им придётся сбрасывать грязные буферы самостоятельно. Однако процесс фоновой записи увеличивает общую нагрузку на подсистему ввода-вывода, так как он может записывать неоднократно изменяемую страницу несколько раз, тогда как её можно было бы записать всего один раз в контрольной точке. Параметры, рассматриваемые в данном подразделе, позволяют настроить поведение фоновой записи для конкретных нужд.

`bgwriter_delay` (`integer`)

Задаёт задержку между раундами активности процесса фоновой записи. Во время раунда этот процесс осуществляет запись некоторого количества загрязнённых буферов (это настраивается следующими параметрами). Затем он засыпает на время `bgwriter_delay`, и всё повторяется снова. Однако если в пуле не остаётся загрязнённых буферов, он может быть неактивен более длительное время, не зависящее от `bgwriter_delay`. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. По умолчанию этот параметр равен 200 миллисекундам (`200ms`). Обратите внимание, что в некоторых системах разрешение таймера составляет 10 мс, поэтому если задать в `bgwriter_delay` значение, не кратное 10, фактически будет получен тот же результат, что и со следующим за ним кратным 10. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`bgwriter_lru_maxpages` (`integer`)

Задаёт максимальное число буферов, которое сможет записать процесс фоновой записи за раунд активности. При нулевом значении фоновая запись отключается. (Учтите, что на контрольные точки, которые управляются отдельным вспомогательным процессом, это не влияет.) По умолчанию значение этого параметра — 100 буферов. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`bgwriter_lru_multiplier` (`floating point`)

Число загрязнённых буферов, записываемых в очередном раунде, зависит от того, сколько новых буферов требовалось серверным процессам в предыдущих раундах. Средняя недавняя потребность умножается на `bgwriter_lru_multiplier` и предполагается, что именно столько буферов потребуется на следующем раунде. Процесс фоновой записи будет записывать на диск и освобождать буферы, пока число чистых буферов не достигнет целевого значения. (При этом число буферов, записываемых за раунд, ограничивается сверху параметром `bgwriter_lru_maxpages`.) Таким образом, со множителем, равным 1.0, записывается ровно столько буферов, сколько требуется по предположению («точно по плану»). Увеличение этого множителя даёт некоторую страховку от резких скачков потребностей, тогда как уменьшение отражает намерение оставить некоторый объём записи для серверных процессов. По умолчанию он равен 2.0. Этот параметр можно установить только в файле `postgresql.conf` или в командной строке при запуске сервера.

`bgwriter_flush_after` (`integer`)

Когда процессом фоновой записи записывается больше заданного объёма данных, сервер даёт указание ОС произвести запись этих данных в нижележащее хранилище. Это ограничивает объём «грязных» данных в страничном кеше ядра и уменьшает вероятность затормаживания при выполнении `fsync` в конце контрольной точки или когда ОС сбрасывает данные на диск большими порциями в фоне. Часто это значительно уменьшает задержки транзакций, но бывают ситуации (особенно когда объём рабочей нагрузки больше shared_buffers, но меньше страничного кеша ОС), когда производительность может упасть. Этот параметр действует не на всех платформах. Если значение параметра задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Он может принимать значение от `0` (при этом управление отложенной записью отключается) до 2 мегабайт (`2MB`). Значение по умолчанию — `512kB` в Linux и `0` в других ОС. (Если `BLCKSZ` отличен от 8 КБ, значение по умолчанию и максимум корректируются пропорционально.) Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

С маленькими значениями `bgwriter_lru_maxpages` и `bgwriter_lru_multiplier` уменьшается активность ввода-вывода со стороны процесса фоновой записи, но увеличивается вероятность того, что запись придётся производить непосредственно серверным процессам, что замедлит выполнение запросов.

18.4.6. Асинхронное поведение

`backend_flush_after` (`integer`)

Когда одним обслуживающим процессом записывается больше заданного объёма данных, сервер даёт указание ОС произвести запись этих данных в нижележащее хранилище. Это ограничивает объём «грязных» данных в страничном кеше ядра и уменьшает вероятность затормаживания при выполнении `fsync` в конце контрольной точки или когда ОС сбрасывает данные на диск большими порциями в фоне. Часто это значительно сокращает задержки транзакций, но бывают ситуации (особенно когда объём рабочей нагрузки больше shared_buffers, но меньше страничного кеша ОС), когда производительность может упасть. Этот параметр действует не на всех платформах. Если значение параметра задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Он может принимать значение от `0` (при этом управление отложенной записью отключается) до 2 мегабайт (`2MB`). По умолчанию он имеет значение `0`, то есть это поведение отключено. (Если `BLCKSZ` отличен от 8 КБ, максимальное значение корректируется пропорционально.)

`effective_io_concurrency` (`integer`)

Задаёт допустимое число параллельных операций ввода-вывода, которое говорит Postgres Pro о том, сколько операций ввода-вывода могут быть выполнены одновременно. Чем больше это число, тем больше операций ввода-вывода будет пытаться выполнить параллельно Postgres Pro в отдельном сеансе. Допустимые значения лежат в интервале от 1 до 1000, а нулевое значение отключает асинхронные запросы ввода-вывода. В настоящее время этот параметр влияет только на сканирование по битовой карте.

Для магнитных носителей хорошим начальным значением этого параметра будет число отдельных дисков, составляющих массив RAID 0 или RAID 1, в котором размещена база данных. (Для RAID 5 следует исключить один диск (как диск с чётностью).) Однако если база данных часто обрабатывает множество запросов в различных сеансах, и при небольших значениях дисковый массив может быть полностью загружен. Если продолжать увеличивать это значение при полной загрузке дисков, это приведёт только к увеличению нагрузки на процессор. Диски SSD и другие виды хранилища в памяти часто могут обрабатывать множество параллельных запросов, так что оптимальным числом может быть несколько сотен.

Асинхронный ввод-вывод зависит от эффективности функции `posix_fadvise`, которая отсутствует в некоторых операционных системах. В случае её отсутствия попытка задать для этого параметра любое ненулевое значение приведёт к ошибке. В некоторых системах (например, в Solaris), эта функция присутствует, но на самом деле ничего не делает.

Значение по умолчанию равно 1 в системах, где это поддерживается, и 0 в остальных. Это значение можно переопределить для таблиц в определённом табличном пространстве, установив одноимённый параметр табличного пространства (см. ALTER TABLESPACE).

`maintenance_io_concurrency` (integer)

Этот параметр подобен `effective_io_concurrency`, но используется для операций обслуживания, которые выполняются в различных клиентских сеансах.

Значение по умолчанию равно 10 в системах, где это поддерживается, и 0 в остальных. Это значение можно переопределить для таблиц в определённом табличном пространстве, установив одноимённый параметр табличного пространства (см. ALTER TABLESPACE).

`io_combine_limit` (integer)

Задаёт наибольший размер ввода-вывода в операциях, объединяющих ввод-вывод. Значение по умолчанию — 128 КБ.

`max_worker_processes` (integer)

Задаёт максимальное число фоновых процессов, которое может поддерживаться в текущем кластере. Этот параметр можно задать только при запуске сервера. Значение по умолчанию — 8.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

Одновременно с изменением этого значения также может быть полезно изменить max_parallel_workers, max_parallel_maintenance_workers, и max_parallel_workers_per_gather.

`max_parallel_workers_per_gather` (integer)

Задаёт максимальное число рабочих процессов, которые могут запускаться одним узлом `Gather` или `Gather Merge`. Параллельные рабочие процессы берутся из пула процессов, контролируемого параметром max_worker_processes, в количестве, ограничиваемом значением max_parallel_workers. Учтите, что запрошенное количество рабочих процессов может быть недоступно во время выполнения. В этом случае план будет выполняться с меньшим числом процессов, что может быть неэффективно. Значение по умолчанию — 2. Значение 0 отключает параллельное выполнение запросов.

Учтите, что параллельные запросы могут потреблять значительно больше ресурсов, чем не параллельные, так как каждый рабочий процесс является отдельным процессом и оказывает на систему примерно такое же влияние, как дополнительный пользовательский сеанс. Это следует учитывать, выбирая значение этого параметра, а также настраивая другие параметры, управляющие использованием ресурсов, например work_mem. Ограничения ресурсов, такие как `work_mem`, применяются к каждому рабочему процессу отдельно, что означает, что общая нагрузка для всех процессов может оказаться гораздо больше, чем при обычном использовании одного процесса. Например, параллельный запрос, задействующий 4 рабочих процесса, может использовать в 5 раз больше времени процессора, объёма памяти, ввода-вывода и т. д., по сравнению с запросом, не задействующим рабочие процессы вовсе.

За дополнительными сведениями о параллельных запросах обратитесь к Главе 15.

`max_parallel_maintenance_workers` (integer)

Задаёт максимальное число рабочих процессов, которые могут запускаться одной служебной командой. В настоящее время параллельные процессы может использовать `CREATE INDEX` при построении индекса-B-дерева или BRIN и `VACUUM` без указания `FULL`. Параллельные рабочие процессы берутся из пула процессов, контролируемого параметром max_worker_processes, в количестве, ограничиваемом значением max_parallel_workers. Учтите, что запрошенное количество рабочих процессов может быть недоступно во время выполнения. В этом случае служебная операция будет выполняться с меньшим числом процессов, чем ожидалось. Значение по умолчанию — 2. Значение 0 отключает использование параллельных исполнителей служебными командами.

Заметьте, что параллельно выполняемые служебные команды не должны потреблять значительно больше памяти, чем равнозначные непараллельные операции. Это отличает их от параллельных запросов, при выполнении которых ограничения ресурсов действуют на отдельные рабочие процессы. Для параллельных служебных команд ограничение ресурсов `maintenance_work_mem` считается действующим на команду в целом, вне зависимости от числа параллельных рабочих процессов. Тем не менее параллельные служебные команды могут гораздо больше нагружать процессор и каналы ввода-вывода.

`max_parallel_workers` (`integer`)

Задаёт максимальное число рабочих процессов, которое кластер сможет поддерживать для параллельных операций. Значение по умолчанию — 8. При увеличении или уменьшения этого значения также может иметь смысл скорректировать max_parallel_maintenance_workers и max_parallel_workers_per_gather. Заметьте, что значение данного параметра, превышающее max_worker_processes, не будет действовать, так как параллельные рабочие процессы берутся из пула рабочих процессов, ограничиваемого этим параметром.

`parallel_leader_participation` (`boolean`)

Позволяет ведущему процессу выполнять план запроса ниже узлов `Gather` и `Gather Merge`, не ожидая рабочие процессы. По умолчанию этот параметр включён (`on`). Значение `off` снижает вероятность блокировки рабочих процессов в случае, если ведущий процесс будет читать кортежи недостаточно быстро, но тогда ведущему приходится дожидаться запуска рабочих процессов, и только затем выдавать первые кортежи. Степень положительного или отрицательного влияния ведущего зависит от типа плана, числа рабочих процессов и длительности запроса.

18.5. Журнал предзаписи

За дополнительной информацией о настройке этих параметров обратитесь к Разделу 27.5.

18.5.1. Параметры

`wal_level` (`enum`)

Параметр `wal_level` определяет, как много информации записывается в WAL. Со значением `replica` (по умолчанию) в журнал записываются данные, необходимые для поддержки архивирования WAL и репликации, включая запросы только на чтение на ведомом сервере. Вариант `minimal` оставляет только информацию, необходимую для восстановления после сбоя или аварийного отключения. Наконец, `logical` добавляет информацию, требующуюся для поддержки логического декодирования. Каждый последующий уровень включает информацию, записываемую на всех уровнях ниже. Задать этот параметр можно только при запуске сервера.

На уровне `minimal` генерируется минимальный объём WAL. В журнал не записывается информация о производимых до конца транзакции операциях с постоянными отношениями, созданными или перезаписанными в данной транзакции. Это позволяет значительно ускорить такие операции (см. Подраздел 14.4.7). Такая оптимизация включается после следующих команд:

`ALTER ... SET TABLESPACE`
`CLUSTER`
`CREATE TABLE`
`REFRESH MATERIALIZED VIEW` (без `CONCURRENTLY`)
`REINDEX`
`TRUNCATE`

Однако минимальный журнал не будет содержать достаточно информации для восстановления на момент времени, поэтому для включения непрерывной архивации (archive_mode) и потоковой двоичной репликации необходимо использовать уровень не ниже `replica`. Фактически сервер даже не запустится в этом режиме, если значение `max_wal_senders` отлично от нуля. Заметьте, что в случае изменения значения `wal_level` на `minimal` сделанные ранее базовые копии становятся непригодными для восстановления на момент времени и для резервных серверов.

На уровне `logical` в журнал записывается та же информация, что и на уровне `replica`, плюс информация, необходимая для извлечения из журнала наборов логических изменений. Повышение уровня до `logical` приводит к значительному увеличению объёма WAL, особенно если многие таблицы имеют характеристику `REPLICA IDENTITY FULL` и выполняется множество команд `UPDATE` и `DELETE`.

В выпусках до 9.6 для этого параметра допускались значения `archive` и `hot_standby`. Эти значения по-прежнему принимаются, но теперь отображаются в значение `replica`.

`fsync` (boolean)

Если этот параметр установлен, сервер Postgres Pro старается добиться, чтобы изменения были записаны на диск физически, выполняя системные вызовы `fsync()` или другими подобными методами (см. wal_sync_method). Это даёт гарантию, что кластер баз данных сможет вернуться в согласованное состояние после сбоя оборудования или операционной системы.

Хотя отключение `fsync` часто даёт выигрыш в скорости, это может привести к неисправимой порче данных в случае отключения питания или сбоя системы. Поэтому отключать `fsync` рекомендуется, только если вы легко сможете восстановить всю базу из внешнего источника.

В качестве примеров, когда отключение `fsync` неопасно, можно привести начальное наполнение нового кластера данными из копии, обработку массива данных, после которой базу данных можно удалить и создать заново, либо эксплуатацию копии базы данных только для чтения, которая регулярно пересоздаётся и не используется для отработки отказа. Качественное оборудование само по себе не является достаточной причиной для отключения `fsync`.

При смене значения `fsync` с off на on для надёжного восстановления также необходимо сбросить все изменённые буферы из ядра в надёжное хранилище. Это можно сделать, когда сервер остановлен или когда режим `fsync` включён, с помощью команды `initdb --sync-only`, либо выполнить команду `sync`, размонтировать файловую систему или перезагрузить сервер.

Во многих случаях отключение synchronous_commit для некритичных транзакций может дать больший выигрыш в скорости, чем отключение `fsync`, при этом не добавляя риски повреждения данных.

Параметр `fsync` можно задать только в файле `postgresql.conf` или в командной строке при запуске сервера. Если вы отключаете этот параметр, возможно, имеет смысл отключить также и full_page_writes.

`synchronous_commit` (enum)

Определяет, после завершения какого уровня обработки WAL сервер будет сообщать об успешном выполнении операции. Допустимые значения: `remote_apply` (применено удалённо), `on` (вкл., по умолчанию), `remote_write` (записано удалённо), `local` (локально) и `off` (выкл.).

Если значение `synchronous_standby_names` не задано, для данного параметра имеют смысл только значения `on` и `off`; с вариантами `remote_apply`, `remote_write` и `local` будет выбран тот же уровень синхронизации, что и с `on`. Локальное действие всех отличных от `off` режимов заключается в ожидании локального сброса WAL на диск. В режиме `off` ожидание отсутствует, поэтому может образоваться окно от момента, когда клиент узнаёт об успешном завершении, до момента, когда транзакция действительно гарантированно защищена от сбоя. (Максимальный размер окна равен тройному значению wal_writer_delay.) В отличие от fsync, значение `off` этого параметра не угрожает целостности данных: сбой операционной системы или базы данных может привести к потере последних транзакций, считавшихся зафиксированными, но состояние базы данных будет точно таким же, как и в случае штатного прерывания этих транзакций. Поэтому выключение режима `synchronous_commit` может быть полезной альтернативой отключению fsync, когда производительность важнее, чем надёжная гарантия сохранности каждой транзакции. Подробнее это обсуждается в Разделе 27.4.

Если значение synchronous_standby_names не пустое, параметр `synchronous_commit` также определяет, должен ли сервер при фиксировании транзакции ждать, пока соответствующие записи WAL будут обработаны на ведомом сервере (серверах).

Со значением `remote_apply` фиксирование завершается только после получения ответов от текущих синхронных ведомых серверов, говорящих, что они получили запись о фиксировании транзакции, сохранили её в надёжном хранилище, а также применили транзакцию, так что она стала видна для запросов на этих серверах. С таким вариантом задержка при фиксировании оказывается больше, так как необходимо дожидаться воспроизведения WAL. Со значением `on` фиксирование завершается только после получения ответов от текущих синхронных ведомых серверов, подтверждающих, что они получили запись о фиксировании транзакции и передали её в надёжном хранилище. Это гарантирует, что транзакция не будет потеряна, если только база данных не будет повреждена и на ведущем, и на всех синхронных ведомых серверах. Со значением `remote_write` фиксирование завершается после получения ответов от текущих синхронных серверов, говорящих, что они получили запись о фиксировании транзакции и сохранили её в своих ФС. Этот вариант позволяет гарантировать сохранность данных в случае отказа ведомого сервера Postgres Pro, но не в случае сбоя на уровне ОС, так как данные могут ещё не достичь надёжного хранилища на этом сервере. Со значением `local` фиксирование завершается после локального сброса данных, не дожидаясь репликации. Обычно это нежелательный вариант при синхронной репликации, но он представлен для полноты.

Этот параметр можно изменить в любое время; поведение каждой конкретной транзакции определяется значением, действующим в момент её фиксирования. Таким образом, есть возможность и смысл фиксировать некоторые транзакции синхронно, а другие — асинхронно. Например, чтобы зафиксировать одну транзакцию из нескольких команд асинхронно, когда по умолчанию выбран противоположный вариант, выполните в этой транзакции `SET LOCAL synchronous_commit TO OFF`.

Характеристики различных значений `synchronous_commit` сведены в Таблице 18.1.

Таблица 18.1. Режимы synchronous_commit

значение synchronous_commit	**гарантированная локальная фиксация**	**гарантированная фиксация на ведомом после сбоя PG**	**гарантированная фиксация на ведомом после сбоя ОС**	**согласованность запросов на ведомом**
remote_apply	•	•	•	•
on	•	•	•	
remote_write	•	•		
local	•			
off				

`wal_sync_method` (enum)

Метод, применяемый для принудительного сохранения изменений WAL на диске. Если режим `fsync` отключён, данный параметр не действует, так как принудительное сохранение изменений WAL не производится вовсе. Возможные значения этого параметра:

- `open_datasync` (для сохранения файлов WAL открывать их функцией `open()` с параметром `O_DSYNC`)
- `fdatasync` (вызывать `fdatasync()` при каждом фиксировании)
- `fsync` (вызывать `fsync()` при каждом фиксировании)

- `fsync_writethrough` (вызывать `fsync()` при каждом фиксировании, форсируя сквозную запись кеша)
- `open_sync` (для сохранения файлов WAL открывать их функцией `open()` с параметром `O_SYNC`)

Не все эти методы поддерживаются в разных системах. По умолчанию выбирается первый метод из списка, который поддерживается текущей системой, с одним исключением — в Linux и FreeBSD по умолчанию выбирается `fdatasync`. Выбираемый по умолчанию вариант не обязательно будет идеальным — в зависимости от требований к отказоустойчивости или производительности может потребоваться скорректировать выбранное значение или внести другие изменения в конфигурацию вашей системы. Соответствующие аспекты конфигурации рассматриваются в Разделе 27.1. Этот параметр можно задать только в файле `postgresql.conf` или в командной строке при запуске сервера.

`full_page_writes` (boolean)

Когда этот параметр включён, сервер Postgres Pro записывает в WAL всё содержимое каждой страницы при первом изменении этой страницы после контрольной точки. Это необходимо, потому что запись страницы, прерванная при сбое операционной системы, может выполниться частично, и на диске окажется страница, содержащая смесь старых данных с новыми. При этом информации об изменениях на уровне строк, которая обычно сохраняется в WAL, будет недостаточно для получения согласованного содержимого такой страницы при восстановлении после сбоя. Сохранение образа всей страницы гарантирует, что страницу можно восстановить корректно, ценой увеличения объёма данных, которые будут записываться в WAL. (Так как воспроизведение WAL всегда начинается от контрольной точки, достаточно сделать это при первом изменении каждой страницы после контрольной точки. Таким образом, уменьшить затраты на запись полных страниц можно, увеличив интервалы контрольных точек.)

Отключение этого параметра ускоряет обычные операции, но может привести к неисправимому повреждению или незаметной порче данных после сбоя системы. Так как при этом возникают практически те же риски, что и при отключении `fsync`, хотя и в меньшей степени, отключать его следует только при тех же обстоятельствах, которые перечислялись в рекомендациях для вышеописанного параметра.

Отключение этого параметра не влияет на возможность применения архивов WAL для восстановления состояния на момент времени (см. Раздел 24.3).

Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию этот параметр имеет значение `on`.

`wal_log_hints` (boolean)

Когда этот параметр имеет значение `on`, сервер Postgres Pro записывает в WAL всё содержимое каждой страницы при первом изменении этой страницы после контрольной точки, даже при второстепенных изменениях так называемых вспомогательных битов.

Если включён расчёт контрольных сумм данных, изменения вспомогательных битов всегда проходят через WAL и этот параметр игнорируется. С помощью этого параметра можно проверить, насколько больше дополнительной информации записывалось бы в журнал, если бы для базы данных был включён подсчёт контрольных сумм.

Этот параметр можно задать только при запуске сервера. По умолчанию он имеет значение `off`.

`wal_compression` (enum)

Этот параметр включает сжатие WAL указанным методом. Если этот параметр имеет значение `on`, сервер Postgres Pro сжимает образы полных страниц, записываемые в WAL, когда включён режим full_page_writes или при создании базовой копии. Сжатый образ страницы будет развёрнут при воспроизведении WAL. Поддерживаемые методы: `pglz`, `lz4` (если Postgres Pro скомпилирован с параметром `--with-lz4`) и `zstd` (если Postgres Pro скомпилирован с параметром `--with-zstd`). Значение по умолчанию — `lz4`. Чтобы отключить сжатие, задайте этому параметру значение `off`. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Включив сжатие, вы можете без дополнительных рисков повреждения данных уменьшить объём WAL, ценой дополнительной нагрузки на процессор, связанной со сжатием данных при записи в WAL и разворачиванием их при воспроизведении WAL.

`wal_init_zero` (`boolean`)

Если этот параметр включён (`on`), создаваемые файлы WAL заполняются нулями. В ряде файловых систем благодаря этому заранее выделяется пространство, которое потребуется для записи WAL. Однако с файловыми системами, работающими по принципу COW (*Copy-On-Write*, Копирование при записи), это может быть бессмысленно, поэтому данный параметр позволяет отключить в данном случае неэффективное поведение. Со значением `off` в создаваемый файл записывается только последний байт, чтобы файл WAL сразу обрёл желаемый размер.

`wal_recycle` (`boolean`)

Если этот параметр имеет значение `on` (по умолчанию), файлы WAL используются повторно (для этого они переименовываются), что избавляет от необходимости создавать новые файлы. В файловых системах COW может быть быстрее создать новые файлы, поэтому данный параметр позволяет отключить это поведение.

`wal_buffers` (`integer`)

Объём разделяемой памяти, который будет использоваться для буферизации данных WAL, ещё не записанных на диск. Значение по умолчанию, равное -1, задаёт размер, равный 1/32 (около 3%) от shared_buffers, но не меньше чем `64 КБ` и не больше чем размер одного сегмента WAL (обычно `16 МБ`). Это значение можно задать вручную, если выбираемое автоматически слишком мало или велико, но при этом любое положительное число меньше `32 КБ` будет восприниматься как `32 КБ`. Если это значение задаётся без единиц измерения, оно считается заданным в блоках WAL (размер которых равен `XLOG_BLCKSZ` байт, обычно это 8 КБ). Этот параметр можно задать только при запуске сервера.

Содержимое буферов WAL записывается на диск при фиксировании каждой транзакции, так что очень большие значения вряд ли принесут значительную пользу. Однако значение как минимум в несколько мегабайт может увеличить быстродействие при записи на нагруженном сервере, когда сразу множество клиентов фиксируют транзакции. Автонастройка, действующая при значении по умолчанию (-1), в большинстве случаев выбирает разумные значения.

`wal_writer_delay` (`integer`)

Определяет, с какой периодичностью процесс записи WAL будет сбрасывать WAL на диск. После очередного сброса WAL он делает паузу, длительность которой задаётся параметром `wal_writer_delay`, но может быть пробуждён асинхронно фиксируемой транзакцией. Если предыдущая операция сброса имела место в течение заданного параметром `wal_writer_delay` времени и полученный за это время объём WAL не достиг значения `wal_writer_flush_after`, данные WAL только передаются ОС, но не сбрасываются на диск. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию — 200 миллисекунд (`200ms`). Обратите внимание, что в некоторых системах разрешение таймера паузы составляет 10 мс, если задать в `wal_writer_delay` значение, не кратное 10, может быть получен тот же результат, что и со следующим за ним кратным 10. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`wal_writer_flush_after` (`integer`)

Определяет, при каком объёме процесс записи WAL будет сбрасывать WAL на диск. Если предыдущая операция сброса имела место в течение заданного параметром `wal_writer_delay` времени и полученный после неё объём WAL не достиг значения `wal_writer_flush_after`, данные WAL только передаются ОС, но не сбрасываются на диск. Если `wal_writer_flush_after` равен `0`, WAL сбрасывается на диск немедленно. Если это значение задаётся без единиц измерения, оно считается заданным в блоках WAL (размер которых равен `XLOG_BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — 1 мегабайт (`1MB`). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`wal_skip_threshold` (`integer`)

Когда выбран `wal_level` `minimal` и фиксируется транзакция, которая создавала или перезаписывала постоянное отношение, этот параметр определяет, как будут сохраняться новые данные. Если объём данных меньше заданного значения, они будут записываться в журнал WAL; в противном случае затронутые файлы просто синхронизируются с ФС. Изменение этого параметра в зависимости от характеристик вашего хранилища может быть полезным, если при фиксировании такой транзакции наблюдается замедление других транзакций. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Значение по умолчанию — два мегабайта (`2MB`).

`commit_delay` (`integer`)

Параметр `commit_delay` добавляет паузу перед собственно выполнением сохранения WAL. Эта задержка может увеличить быстродействие при фиксировании множества транзакций, позволяя зафиксировать большее число транзакций за одну операцию сохранения WAL, если система нагружена достаточно сильно и за заданное время успевают зафиксироваться другие транзакции. Однако этот параметр также увеличивает задержку максимум до `commit_delay` при каждом сохранении WAL. Эта задержка окажется бесполезной, если никакие другие транзакции не будут зафиксированы за это время, поэтому она добавляется, только если в момент запроса сохранения WAL активны как минимум `commit_siblings` других транзакций. Кроме того, эти задержки не добавляются при выключенном `fsync`. Если это значение задаётся без единиц измерения, оно считается заданным в микросекундах. По умолчанию значение `commit_delay` равно нулю (задержка отсутствует). Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

В PostgreSQL до версии 9.3, параметр `commit_delay` работал по-другому и не так эффективно: он задерживал только фиксирование транзакций, а не все операции сохранения WAL, и заданная пауза выдерживалась полностью, даже если WAL удавалось сохранить быстрее. Начиная с версии 9.3, заданное время ожидает только первый процесс, готовый произвести сохранение, тогда как все последующие процессы ждут только, когда он закончит эту операцию.

`commit_siblings` (`integer`)

Минимальное число одновременно открытых транзакций, при котором будет добавляться задержка `commit_delay`. Чем больше это значение, тем больше вероятность, что минимум одна транзакция окажется готовой к фиксированию за время задержки. По умолчанию это число равно пяти.

18.5.2. Контрольные точки

`checkpoint_timeout` (`integer`)

Максимальное время между автоматическими контрольными точками в WAL. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. Допускаются значения от 30 секунд до одного дня. Значение по умолчанию — пять минут (`5min`). Увеличение этого параметра может привести к увеличению времени, которое потребуется для восстановления после сбоя. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`checkpoint_completion_target` (`floating point`)

Задаёт целевое время для завершения процедуры контрольной точки, как долю общего времени между контрольными точками. Значение по умолчанию — 0.9, что распределяет контрольную точку почти по всему доступному интервалу, обеспечивая достаточно стабильную нагрузку ввода-вывода, а также оставляя некоторое время для издержек на завершение процедуры контрольной точки. Уменьшать значение этого параметра не рекомендуется, так как это ускорит данную процедуру, что нежелательно, поскольку приведёт к ускорению ввода-вывода в процессе контрольной точки, и, следовательно, к меньшему объёму ввода-вывода в промежуток времени между завершением процедуры контрольной точки и следующей запланированной контрольной точкой. Этот параметр можно установить только в файле `postgresql.conf` или в командной строке сервера.

`checkpoint_flush_after` (`integer`)

Когда в процессе контрольной точки записывается больше заданного объёма данных, сервер даёт указание ОС произвести запись этих данных в нижележащее хранилище. Это ограничивает объём «грязных» данных в страничном кеше ядра и уменьшает вероятность затормаживания при выполнении `fsync` в конце контрольной точки или когда ОС сбрасывает данные на диск большими порциями в фоне. Часто это значительно уменьшает задержки транзакций, но бывают ситуации (особенно когда объём рабочей нагрузки больше shared_buffers, но меньше страничного кеша ОС), когда производительность может упасть. Этот параметр действует не на всех платформах. Если значение параметра задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Он может принимать значение от `0` (при этом управление отложенной записью отключается) до 2 мегабайт (`2MB`). Значение по умолчанию — `256kB` в Linux и `0` в других ОС. (Если `BLCKSZ` отличен от 8 КБ, значение по умолчанию и максимум корректируются пропорционально.) Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`checkpoint_warning` (`integer`)

Записывать в журнал сервера сообщение в случае, если контрольные точки, вызванные заполнением файлов сегментов WAL, выполняются быстрее, чем через заданное время (что говорит о том, что нужно увеличить `max_wal_size`). Если это значение задаётся без единиц измерения, оно считается заданным в секундах. Значение по умолчанию равно 30 секундам (`30s`). При нуле это предупреждение отключается. Если `checkpoint_timeout` меньше чем `checkpoint_warning`, предупреждения так же не будут выводиться. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`max_wal_size` (`integer`)

Максимальный размер, до которого может вырастать WAL во время автоматических контрольных точек. Это мягкий предел; размер WAL может превышать `max_wal_size` при особых обстоятельствах, например при большой нагрузке, сбое в `archive_command`/`archive_library` или при большом значении `wal_keep_size`. Если это значение задаётся без единиц измерения, оно считается заданным в мегабайтах. Значение по умолчанию — 1 ГБ. Увеличение этого параметра может привести к увеличению времени, которое потребуется для восстановления после сбоя. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`min_wal_size` (`integer`)

Пока WAL занимает на диске меньше этого объёма, старые файлы WAL в контрольных точках всегда перерабатываются, а не удаляются. Это позволяет зарезервировать достаточно места для WAL, чтобы справиться с резкими скачками использования WAL, например, при выполнении больших пакетных заданий. Если это значение задаётся без единиц измерения, оно считается заданным в мегабайтах. Значение по умолчанию — 80 МБ. Этот параметр можно установить только в `postgresql.conf` или в командной строке сервера.

18.5.3. Архивация

`archive_mode` (`enum`)

Когда параметр `archive_mode` включён, полные сегменты WAL передаются в хранилище архива командой archive_command или функциями archive_library. Помимо значения `off` (выключающего архивацию) есть ещё два: `on` (вкл.) и `always` (всегда). В обычном состоянии эти два режима не различаются, но в режиме `always` архивация WAL активна и во время восстановления архива, и при использовании ведомого сервера. В этом режиме все файлы, восстановленные из архива или полученные при потоковой репликации, будут архивироваться (снова). За подробностями обратитесь к Подразделу 25.2.9.

Параметры `archive_mode` и `archive_command`/`archive_library` разделены, чтобы команду `archive_command` и библиотеку `archive_library` можно было изменять, не отключая режим архивации. Этот параметр можно задать только при запуске сервера. Режим архивации нельзя включить, когда установлен минимальный уровень WAL (`wal_level` имеет значение `minimal`).

`archive_command` (`string`)

Команда локальной оболочки, которая будет выполняться для архивации завершённого сегмента WAL. Любое вхождение `%p` в этой строке заменяется путём архивируемого файла, а вхождение `%f` заменяется только его именем. (Путь задаётся относительно рабочего каталога сервера, то есть каталога данных кластера.) Чтобы вставить в команду символ `%`, его нужно записать как `%%`. Важно, чтобы команда возвращала нулевой код, только если она завершается успешно. За дополнительной информацией обратитесь к Подразделу 24.3.1.

Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Данный параметр используется, только если `archive_mode` был включён при запуске или параметр `archive_library` содержит пустую строку. Если указаны и `archive_command`, и `archive_library`, возникнет ошибка. Если значение `archive_command` — пустая строка (по умолчанию), но `archive_mode` включён (и при этом значение `archive_library` — тоже пустая строка), архивация WAL временно отключается, но сервер продолжает накапливать файлы сегментов WAL в ожидании, что команда будет вскоре определена. Если в качестве `archive_command` задать команду, которая ничего не делает, но сообщает об успешном завершении, например `/bin/true` (или `REM` в Windows), архивация по сути отключается, но при этом нарушается цепочка файлов WAL, необходимых для восстановления архива, поэтому такой вариант следует использовать только в особых случаях.

`archive_library` (`string`)

Библиотека, используемая для архивирования завершённых сегментов файла WAL. Если задана пустая строка (по умолчанию), применяется метод архивирования посредством оболочки и используется archive_command. Если заданы оба параметра `archive_command` и `archive_library`, возникнет ошибка. В противном случае для архивирования используется указанная общая библиотека. При изменении этого параметра postmaster перезапускает процесс архивирования WAL. За дополнительными сведениями обратитесь к разделам Подраздел 24.3.1 и Глава 49.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`archive_timeout` (`integer`)

Команда archive_command или функции archive_library вызываются только для завершённых сегментов WAL. Поэтому, если ваш сервер записывает мало данных WAL (или это наблюдается в некоторые периоды времени), от завершения транзакции до надёжного сохранения её в архивном хранилище может пройти довольно много времени. Для ограничения времени существования неархивированных данных можно установить значение `archive_timeout`, чтобы сервер периодически переключался на новый файл сегмента WAL. Когда этот параметр больше нуля, сервер будет переключаться на новый файл сегмента, если с момента последнего переключения на новый файл прошло заданное время и наблюдалась какая-то активность базы данных, даже если это была просто контрольная точка. (Контрольные точки пропускаются, если в базе отсутствует активность). Заметьте, что архивируемые файлы, закрываемые досрочно из-за принудительного переключения, всё равно будут иметь тот же размер, что и полностью заполненные. Поэтому устанавливать для `archive_timeout` очень маленькое значение неразумно — это ведёт к раздуванию архивного хранилища. Обычно для `archive_timeout` имеет смысл задавать значение около минуты. Если вам нужно, чтобы данные копировались с главного сервера быстрее, вам следует подумать о переходе от архивации к потоковой репликации. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера.

18.5.4. Восстановление

В этом разделе описываются общие параметры, связанные с восстановлением, которые влияют на восстановление после сбоя, потоковую репликацию и репликацию с использованием архива.

`recovery_prefetch` (enum)

Определяет, выполнять ли во время восстановления предварительную выборку блоков, на которые есть ссылки в WAL и которых ещё нет в пуле буферов. Допустимые значения: `off` (выкл.), `on` (вкл.) и `try` (пробовать), выбираемое по умолчанию. Значение `try` включает предвыборку, только если операционная система предоставляет функцию `posix_fadvise`, которая в настоящее время используется для реализации предварительной выборки. Обратите внимание, что некоторые операционные системы предоставляют эту функцию, но она ничего не делает.

Благодаря предвыборке блоков, которые вскоре потребуются, может уменьшиться время ожидания ввода-вывода во время восстановления при определённой нагрузке. См. также параметры wal_decode_buffer_size и maintenance_io_concurrency, ограничивающие активность предварительной выборки.

`wal_decode_buffer_size` (integer)

Ограничение объёма WAL, который сервер будет читать наперёд, чтобы определить блоки для предварительной выборки. Если это значение указано без единиц измерения, оно считается заданным в байтах. По умолчанию 512 КБ.

18.5.5. Восстановление из архива

В этом разделе описываются параметры, действующие только в процессе восстановления. Они должны сбрасываться для любой последующей операции восстановления.

Под «восстановлением» здесь понимается и использование сервера в качестве ведомого, и выполнение целевого восстановления данных. Обычно ведомые серверы используется для обеспечения высокой степени доступности и/или масштабируемости чтения, тогда как целевое восстановление производится в случае потери данных.

Чтобы запустить сервер в режиме ведомого, создайте в каталоге данных файл `standby.signal`. Сервер перейдёт к восстановлению и останется в этом состоянии и по достижении конца заархивированного WAL, чтобы осуществлять восстановление дальше. Для этого он подключится к передающему серверу, используя параметры в `primary_conninfo`, или будет получать новые сегменты WAL с помощью команды `restore_command`. Применительно к данному режиму представляют интерес параметры, описанные в этом разделе и в Подразделе 18.6.3. Параметры, описанные в Подраздел 18.5.6, также будут действовать, хотя для данного режима они вряд ли будут полезными.

Чтобы запустить сервер в режиме целевого восстановления, создайте в каталоге данных `recovery.signal`. В случае одновременного существования файлов `standby.signal` и `recovery.signal` предпочтение отдаётся режиму ведомого. Режим целевого восстановления завершается после полного воспроизведения WAL из архива или при достижении целевой точки (`recovery_target`). В данном режиме используются параметры, описанные в этом разделе и в Подразделе 18.5.6.

`restore_command` (string)

Команда оболочки ОС, которая выполняется для извлечения архивного сегмента из набора файлов WAL. Этот параметр требуется для восстановления из архива, но необязателен для потоковой репликации. Любое вхождение `%f` в строке заменяется именем извлекаемого из архива файла, а `%p` заменяется на путь назначения на сервере. (Путь указывается относительно текущего рабочего каталога, т. е. относительно каталога хранения данных кластера.) Любое вхождение `%r` заменяется на имя файла, в котором содержится последняя действительная точка восстановления. Это самый ранний файл, который требуется хранить для возможности восстановления; зная его имя, размер архива можно уменьшить до минимально необходимого. `%r` обычно используется при организации тёплого резерва (см. Раздел 25.2). Для того чтобы указать символ `%`, продублируйте его (`%%`).

Обратите внимание, что команда должна возвращать ноль на выходе лишь в случае успешного выполнения. Команде *будут* поступать имена файлов, отсутствующих в архиве; в этом случае она должна возвращать ненулевой статус. Примеры:

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  # Windows
```

В случае прерывания команды сигналом (отличным от SIGTERM, который используется для остановки сервера баз данных) или при возникновении ошибки оболочки (например, если команда не найдена), процесс восстановления будет остановлен и сервер не запустится.

`archive_cleanup_command` (`string`)

Этот необязательный параметр указывает команду оболочки ОС, которая будет вызываться при каждой точке перезапуска. Назначение команды `archive_cleanup_command` — предоставить механизм очистки от старых архивных файлов WAL, которые более не нужны на ведомом сервере. Любое вхождение `%r` заменяется на имя файла, содержащего последнюю действительную точку перезапуска. Это самый ранний файл, который необходимо *хранить* для возможности восстановления, а более старые файлы вполне можно удалить. Эта информация может быть использована для усечения архива с целью его минимизации при сохранении возможности последующего восстановления из заданной точки. Модуль pg_archivecleanup часто используется в качестве `archive_cleanup_command` в конфигурациях с одним ведомым сервером, например:

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

Стоит обратить внимание, что в конфигурациях с множеством ведомых серверов, использующих общий архивный каталог для восстановления, необходимо контролировать удаление файлов WAL, так как они могут ещё быть нужны некоторым серверам. Поэтому `archive_cleanup_command` обычно используется при организации тёплого резерва (см. Раздел 25.2). Чтобы указать символ `%` в команде, продублируйте его (`%%`).

В случаях, когда команда возвращает ненулевой статус завершения, в журнал записывается предупреждающее сообщение. Если же команда прерывается сигналом или оболочка ОС выдаёт ошибку (например, команда не найдена), вызывается критическая ошибка.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`recovery_end_command` (`string`)

Этот параметр задаёт команду оболочки, которая будет выполнена единожды в конце процесса восстановления. Назначение параметра `recovery_end_command` — предоставить механизм для очистки после репликации или восстановления. Любое вхождение `%r` заменяется именем файла, содержащим последнюю действительную точку восстановления, например, как в archive_cleanup_command.

В случаях, когда команда возвращает ненулевой статус завершения, в журнал записывается предупреждающее сообщение, но сервер, несмотря на это, продолжает запускаться. Если же команда прерывается сигналом или оболочка ОС выдаёт ошибку (например, команда не найдена), кластер баз данных не запускается.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

18.5.6. Цель восстановления

По умолчанию восстановление производится вплоть до окончания журнала WAL. Чтобы остановить процесс восстановления в более ранней точке, можно использовать один из следующих параметров: `recovery_target`, `recovery_target_lsn`, `recovery_target_name`, `recovery_target_time` или `recovery_target_xid`. Если в конфигурационном файле устанавливаются сразу несколько этих параметров, выдаётся ошибка. Задать эти параметры можно только при запуске сервера.

`recovery_target= 'immediate'`

Данный параметр указывает, что процесс восстановления должен завершиться, как только будет достигнуто целостное состояние, т. е. как можно раньше. При восстановлении из оперативной резервной копии, это будет точкой, в которой завершился процесс резервного копирования.

Технически это строковый параметр, но значение `'immediate'` — единственно допустимое в данный момент.

`recovery_target_name` (`string`)

Этот параметр указывает именованную точку восстановления (созданную с помощью `pg_create_restore_point()`), до которой будет производиться восстановление.

`recovery_target_time` (`timestamp`)

Данный параметр указывает точку времени, вплоть до которой будет производиться восстановление. Окончательно точка останова определяется в зависимости от значения recovery_target_inclusive.

Значение этого параметра задаётся в том же формате, что принимается типом данных `timestamp with time zone`, за исключением того, что в нём нельзя использовать сокращённое название часового пояса (если только переменная timezone_abbreviations не была установлена в файле конфигурации выше). Поэтому рекомендуется задавать числовое смещение от UTC или записывать название часового пояса полностью, например `Europe/Helsinki` (но не `EEST`).

`recovery_target_xid` (`string`)

Параметр указывает идентификатор транзакции, вплоть до которой необходимо произвести процедуру восстановления. Имейте в виду, что числовое значение идентификатора отражает последовательность именно старта транзакций, а фиксироваться они могут в ином порядке. Восстановлению будут подлежать все транзакции, что были зафиксированы до указанной (и, возможно, включая её). Точность точки останова также зависит от recovery_target_inclusive.

`recovery_target_lsn` (`pg_lsn`)

Данный параметр указывает LSN позиции в журнале предзаписи, до которой должно выполняться восстановление. Точная позиция остановки зависит также от параметра recovery_target_inclusive. Этот параметр принимает значение системного типа данных `pg_lsn`.

Следующие параметры уточняют целевую точку восстановления и определяют, что будет происходить при её достижении:

`recovery_target_inclusive` (`boolean`)

Указывает на необходимость остановки сразу после (`on`) либо до (`off`) достижения целевой точки. Применяется одновременно с recovery_target_lsn, recovery_target_time или recovery_target_xid. Этот параметр определяет, нужно ли восстанавливать транзакции, у которых позиция в WAL (LSN), время фиксации либо идентификатор в точности совпадает с заданным соответствующим значением. По умолчанию выбирается вариант `on`.

`recovery_target_timeline` (`string`)

Указывает линию времени для восстановления. Значение может задаваться числовым идентификатором линии времени или ключевым словом. С ключевым словом `current` восстанавливается та линия времени, которая была активной при создании базовой резервной копии. С ключевым словом `latest` восстанавливаться будет последняя линия времени, найденная в архиве, что полезно для ведомого сервера. По умолчанию подразумевается `latest`.

Чтобы указать идентификатор линии времени в шестнадцатеричном формате (например, если он извлечён из имени файла WAL или файла истории), добавьте к нему префикс `0x`. Например, если имя файла WAL — `00000011000000A10000004F`, то идентификатор линии времени — `0x11` (или десятичное число 17).

Задавать этот параметр обычно требуется только в сложных ситуациях с повторами восстановления, когда необходимо вернуться к состоянию, которое само было достигнуто после восстановления на момент времени. Это обсуждается в Подразделе 24.3.6.

`recovery_target_action` (`enum`)

Указывает, какое действие должен предпринять сервер после достижения цели восстановления. Вариант по умолчанию — `pause`, что означает приостановку восстановления. Второй вариант, `promote`, означает, что процесс восстановления завершится, и сервер начнёт принимать подключения. Наконец, с вариантом `shutdown` сервер остановится, как только цель восстановления будет достигнута.

Вариант `pause` позволяет выполнить запросы к базе данных и убедиться в том, что достигнутая цель оказалась желаемой точкой восстановления. Для снятия с паузы нужно вызвать `pg_wal_replay_resume()` (см. Таблицу 9.97), что в итоге приведёт к завершению восстановления. Если же окажется, что мы ещё не достигли желаемой точки восстановления, нужно остановить сервер, установить более позднюю цель и перезапустить сервер для продолжения восстановления.

Вариант `shutdown` полезен для получения готового экземпляра сервера в желаемой точке. При этом данный экземпляр сможет воспроизводить дополнительные записи WAL (а при перезапуске ему придётся воспроизводить записи WAL после последней контрольной точки).

Заметьте, что так как `recovery.signal` не переименовывается, когда в `recovery_target_action` выбран вариант `shutdown`, при последующем запуске будет происходить немедленная остановка, пока вы не измените конфигурацию или не удалите файл `recovery.signal` вручную.

Этот параметр не действует, если цель восстановления не установлена. Если не включён режим hot_standby, значение `pause` действует так же, как и `shutdown`. Если цель восстановления достигается в процессе повышения, `pause` действует как `promote`.

В любом случае, если задана цель восстановления, но восстановление архива завершается до её завершения, сервер завершит работу с критической ошибкой.

18.5.7. Создание сводок WAL

Эти параметры управляют созданием сводок WAL — функциональностью, которую необходимо включить для выполнения инкрементального резервного копирования.

`summarize_wal` (`boolean`)

Включает процесс создания сводок WAL. Обратите внимание, что создание сводок WAL можно включить как на ведущем, так и на резервном сервере. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию `off`.

Сервер не может быть запущен с `summarize_wal=on`, если `wal_level` имеет значение `minimal`. Если после запуска сервера задать `summarize_wal=on`, когда `wal_level=minimal`, процесс создания сводок запустится, но не станет создавать файлы сводок для WAL, сгенерированных с `wal_level=minimal`.

`wal_summary_keep_time` (`integer`)

Задаёт время, по истечении которого процесс создания сводок WAL автоматически удаляет старые сводки. Чтобы определять, какие файлы подлежат удалению, используются их временные метки. Обычно значение следует установить с запасом на время, превышающее интервал между полным резервным копированием и последующим инкрементальным резервным копированием. Сводки WAL должны быть доступны для всего диапазона изменений между предыдущей копией и новой, иначе создание инкрементальной копии завершится ошибкой. Если значение этого параметра равно нулю, сводки WAL не будут автоматически удаляться, но можно вручную удалять файлы, которые не требуются для будущих инкрементальных резервных копий. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Если значение задано без единиц измерения, оно считается заданным в минутах. Значение по умолчанию — 10 дней. Если `summarize_wal = off`, существующие сводки WAL не будут удаляться независимо от значения этого параметра, поскольку процесс создания сводок WAL не запустится.

18.6. Репликация

Эти параметры управляют поведением встроенных механизмов *потоковой репликации* (см. Подраздел 25.2.5) и *логической репликации* (см. Главу 28).

В *потоковой репликации* один сервер является ведущим, а другие — ведомыми. Ведущий сервер всегда передаёт, а ведомые всегда принимают данные репликации, но когда настроена каскадная репликация (см. Подраздел 25.2.7), ведомые серверы могут быть и передающими. Следующие параметры в основном относятся к передающим и ведомым серверам, хотя некоторые параметры имеют смысл только для ведущего. Все эти параметры могут быть разными в рамках одного кластера, если это требуется.

Для *логической репликации публикующие серверы* (серверы, выполняющие `CREATE PUBLICATION`) передают данные репликации *подписчикам* (серверам, выполняющим `CREATE SUBSCRIPTION`). Серверы также могут быть публикующими и подписчиками одновременно. Обратите внимание, что в следующих разделах публикующие серверы называются «передающими серверами» или «передатчиками». За дополнительной информацией о параметрах конфигурации логической репликации обратитесь к Разделу 28.11.

18.6.1. Передающие серверы

Эти параметры можно задать на любом сервере, который передаёт данные репликации одному или нескольким ведомым. Ведущий сервер всегда является передающим, так что на нём они должны задаваться всегда. Роль и значение этих параметров не меняются после того, как ведомый сервер становится ведущим.

`max_wal_senders` (`integer`)

Задаёт максимально допустимое число одновременных подключений ведомых серверов или клиентов потокового копирования (т. е. максимальное количество одновременно работающих процессов передачи WAL). Значение по умолчанию — `10`. При значении `0` репликация отключается. В случае неожиданного отключения клиента потоковой передачи слот подключения может оставаться занятым до достижения тайм-аута, так что этот параметр должен быть немного больше максимально допустимого числа клиентов, чтобы отключившиеся клиенты могли переподключиться немедленно. Задать этот параметр можно только при запуске сервера. Чтобы к данному серверу могли подключаться ведомые, уровень `wal_level` должен быть `replica` или выше.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

`max_replication_slots` (`integer`)

Задаёт максимальное число слотов репликации (см. Подраздел 25.2.6), которое сможет поддерживать сервер. Значение по умолчанию — 10. Этот параметр можно задать только при запуске сервера. Если заданное значение данного параметра будет меньше, чем число уже существующих слотов репликации, сервер не запустится. Чтобы слоты репликации можно было использовать, нужно также установить в `wal_level` уровень `replica` или выше.

Обратите внимание, что этот параметр также применяется на стороне подписчика, но с другим значением.

`wal_keep_size` (`integer`)

Задаёт минимальный объём прошлых файлов WAL, который будет сохраняться в каталоге `pg_wal`, чтобы ведомый сервер мог выбрать их при потоковой репликации. Если ведомый сервер, подключённый к передающему, отстаёт больше чем на `wal_keep_size` мегабайт, передающий может удалить сегменты WAL, всё ещё необходимые ведомому, и в этом случае соединение репликации прервётся. В результате этого затем также будут прерваны зависимые соединения. (Однако ведомый сервер сможет восстановиться, выбрав этот сегмент из архива, если осуществляется архивация WAL.)

Этот параметр задаёт только минимальный объём сегментов, который будет сохраняться в каталоге `pg_wal`; для архивации WAL или для восстановления с момента контрольной точки может потребоваться сохранить больше сегментов. Если `wal_keep_size` равен нулю (это значение по умолчанию), система не сохраняет никакие дополнительные сегменты для ведомых серверов, поэтому число старых сегментов WAL, доступных для ведомых, зависит от положения предыдущей контрольной точки и состояния архивации WAL. Если это значение задаётся без единиц измерения, оно считается заданным в мегабайтах. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`max_slot_wal_keep_size` (`integer`)

Задаёт максимальный размер файлов WAL, который может оставаться в каталоге `pg_wal` для слотов репликации после выполнения контрольной точки. Со значением `max_slot_wal_keep_size`, равным -1 (по умолчанию), для слотов репликации может сохраняться неограниченный объём файлов WAL. При неотрицательном значении, если позиция restart_lsn для слота репликации отстаёт от текущего LSN более чем на заданное количество мегабайт, использующий этот слот ведомый сервер может лишиться возможности продолжить репликацию вследствие удаления нужных ему файлов WAL. Доступность WAL для слотов репликации показывается в представлении pg_replication_slots. Если это значение указано без единиц измерения, оно считается заданным в мегабайтах. Данный параметр можно задать только в файле `postgresql.conf` или в командной строке сервера.

`wal_sender_timeout` (`integer`)

Задаёт период времени, по истечении которого прерываются неактивные соединения репликации. Это помогает передающему серверу обнаружить сбой ведомого или разрывы сети. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию — 60 секунд. При значении, равном нулю, тайм-аут отключается.

Если узлы кластера распределены географически, его можно гибко настроить, используя разные значения в разных расположениях. Маленькие значения полезны для более быстрого обнаружения потери ведомого, подключённого по скоростному каналу, а большие — для более надёжного определения состояния ведомого, расположенного в удалённой сети, соединение с которой характеризуется большими задержками.

`track_commit_timestamp` (`boolean`)

Включает запись времени фиксации транзакций. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию этот параметр имеет значение `off`.

`synchronized_standby_slots` (`string`)

Разделённый запятыми список имён слотов резервных серверов потоковой репликации, которых будут ожидать логические передающие WAL процессы. Эти процессы будут отправлять декодированные изменения модулям только после того, как указанные слоты репликации подтвердят получение WAL. Это гарантирует, что слоты отработки отказа логической репликации не будут использовать изменения до тех пор, пока эти изменения не будут получены и сброшены на соответствующие физические резервные серверы. Если подключение логической репликации предназначено для переключения на физический резервный сервер после повышения статуса резервного сервера, в списке должен быть указан слот физической репликации для резервного сервера. Обратите внимание, что логическая репликация не будет продолжена, если слоты, указанные в `synchronized_standby_slots`, не существуют или аннулированы. Кроме того, функции управления репликацией `pg_replication_slot_advance`, `pg_logical_slot_get_changes` и `pg_logical_slot_peek_changes` при использовании со слотами логической отработки отказов будут блокироваться до тех пор, пока все физические слоты, указанные в `synchronized_standby_slots`, не подтвердят получение WAL.

Резервные серверы, соответствующие слотам физической репликации в `synchronized_standby_slots`, должны установить `sync_replication_slots = true`, чтобы получать изменения слотов отработки отказа логической репликации от ведущего сервера.

18.6.2. Главный сервер

Эти параметры можно задать на главном/ведущем сервере, который должен передавать данные репликации одному или нескольким ведомым. Заметьте, что помимо этих параметров на ведущем сервере должен быть правильно установлен wal_level, а также может быть включена архивация WAL (см. Подраздел 18.5.3). Значения этих параметров на ведомых серверах не важны, хотя их можно подготовить заранее, на случай, если ведомый сервер придётся сделать ведущим.

`synchronous_standby_names` (`string`)

Определяет список ведомых серверов, которые могут поддерживать *синхронную репликацию*, как описано в Подразделе 25.2.8. Активных синхронных ведомых серверов может быть один или несколько; транзакции, ожидающие фиксации, будут завершаться только после того, как эти ведомые подтвердят получение их данных. Синхронными ведомыми будут те, имена которых указаны в этом списке и которые подключены к ведущему и принимают поток данных в реальном времени (что показывает признак `streaming` в представлении `pg_stat_replication`). Указание нескольких имён ведомых серверов позволяет обеспечить очень высокую степень доступности и защиту от потери данных.

Именем ведомого сервера в этом контексте считается значение `application_name` ведомого сервера, задаваемое в свойствах подключения. При организации физической репликации оно должно задаваться в строке `primary_conninfo`; по умолчанию это значение параметра cluster_name, если он задан, или `walreceiver` в противном случае. Для логической репликации его можно задать в строке подключения для подписки (по умолчанию это имя подписки). Как задать его для других потребителей потоков репликации, вы можете узнать в их документации.

Этот параметр принимает список ведомых серверов в одной из следующих форм:

```
[FIRST] число_синхронных ( имя_ведомого [, ...] )
ANY число_синхронных ( имя_ведомого [, ...] )
имя_ведомого [, ...]
```

здесь *`число_синхронных`* — число синхронных ведомых серверов, от которых необходимо дожидаться ответов для завершения транзакций, а *`имя_ведомого`* — имя ведомого сервера. Слова `FIRST` и `ANY` задают метод выбора синхронных ведомых из перечисленных серверов.

Ключевое слово `FIRST`, в сочетании с *`числом_синхронных`*, выбирает синхронную репликацию на основе приоритетов, когда транзакции фиксируются только после того, как их записи в WAL реплицируются на *`число_синхронных`* ведомых серверов, выбираемых согласно приоритетам. Например, со значением `FIRST 3 (s1, s2, s3, s4)` для фиксации транзакции необходимо дождаться ответа от трёх наиболее приоритетных из серверов `s1`, `s2`, `s3` и `s4`. Ведомые серверы, имена которых идут в этом списке первыми, будут иметь больший приоритет и будут считаться синхронными. Серверы, следующие в списке за ними, будут считаться потенциальными синхронными. Если один из текущих синхронных серверов по какой-то причине отключается, он немедленно будет заменён следующим сервером с наибольшим приоритетом. Ключевое слово `FIRST` может быть опущено.

Ключевое слово `ANY`, в сочетании с *`числом_синхронных`*, выбирает синхронную репликацию на основе кворума, когда транзакции фиксируются только после того, как их записи в WAL реплицируются на *как минимум* *`число_синхронных`* перечисленных серверов. Например, со значением `ANY 3 (s1, s2, s3, s4)` для фиксации транзакции необходимо дождаться ответа от как минимум трёх из серверов `s1`, `s2`, `s3` и `s4`.

Ключевые слова `FIRST` и `ANY` воспринимаются без учёта регистра. Если такое же имя оказывается у одного из ведомых серверов, его *`имя_ведомого`* нужно заключить в двойные кавычки.

Третья форма использовалась в Postgres Pro до версии 9.6 и по-прежнему поддерживается. По сути она равнозначна первой с `FIRST` и *`числом_синхронным`*, равным 1. Например, `FIRST 1 (s1, s2)` и `s1, s2` действуют одинаково: в качестве синхронного ведомого выбирается либо `s1`, либо `s2`.

Специальному элементу `*` соответствует имя любого ведомого.

Уникальность имён ведомых серверов не контролируется. В случае дублирования имён более приоритетным будет один из серверов с подходящим именем, хотя какой именно, не определено.

Примечание

Каждое *`имя_ведомого`* должно задаваться в виде допустимого идентификатора SQL, кроме `*`. При необходимости его можно заключать в кавычки. Но заметьте, что идентификаторы *`имя_ведомого`* сравниваются с именами приложений без учёта регистра, независимо от того, заключены ли они в кавычки или нет.

Если имена синхронных ведомых серверов не определены, синхронная репликация не включается и фиксируемые транзакции не будут ждать репликации. Это поведение по умолчанию. Даже когда синхронная репликация включена, для отдельных транзакций можно отключить ожидание репликации, задав для параметра synchronous_commit значение `local` или `off`.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

18.6.3. Ведомые серверы

Эти параметры управляют поведением ведомого сервера, который будет получать данные репликации. На ведущем сервере они не играют никакой роли.

`primary_conninfo` (`string`)

Указывает строку подключения резервного сервера к передающему. Формат строки описан в Подразделе 32.1.1. Вместо опущенных параметров подключения используются соответствующие переменные окружения (см. Раздел 32.15). Если же и переменные не установлены, используются значения по умолчанию.

В строке подключения должно задаваться имя (или адрес) передающего сервера, а также номер порта, если он отличается от подразумеваемого по умолчанию резервным. Также в ней указывается имя пользователя, соответствующее роли с необходимыми правами на передающем сервере (см. Подраздел 25.2.5.1). Если сервер осуществляет аутентификацию по паролю, дополнительно потребуется задать пароль. Его можно указать как в строке `primary_conninfo`, так и отдельно, в файле `~/.pgpass` на резервном сервере (для базы данных `replication`).

Для синхронизации слотов репликации (см. Подраздел 47.2.3) также необходимо указать допустимое имя `dbname` в строке `primary_conninfo`. Оно используется только для синхронизации слотов и игнорируется при потоковой передаче.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Если значение данного параметра меняется во время работы процесса-приёмника WAL, этому процессу посылается сигнал для отключения, и ожидается, что он перезапустится с новым значением (если только определена непустая строка `primary_conninfo`). Этот параметр оказывает влияние только при работе сервера в режиме ведомого.

`primary_slot_name` (`string`)

Дополнительно задаёт заранее созданный слот, который будет использоваться при подключении к передающему серверу по протоколу потоковой репликации для управления освобождением ресурсов вышестоящего узла (см. Подраздел 25.2.6). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Если значение данного параметра меняется во время работы процесса-приёмника WAL, этому процессу посылается сигнал для отключения, и ожидается, что он перезапустится с новым значением. Этот параметр не действует, если строка `primary_conninfo` не определена или сервер работает не в режиме ведомого.

`hot_standby` (`boolean`)

Определяет, можно ли будет подключаться к серверу и выполнять запросы в процессе восстановления, как описано в Разделе 25.4. Значение по умолчанию — `on` (подключения разрешаются). Задать этот параметр можно только при запуске сервера. Данный параметр играет роль только в режиме ведомого сервера или при восстановлении архива.

`max_standby_archive_delay` (`integer`)

В режиме горячего резерва этот параметр определяет, как долго должен ждать ведомый сервер, прежде чем отменять запросы, конфликтующие с очередными изменениями в WAL, как описано в Подразделе 25.4.2. Задержка `max_standby_archive_delay` применяется при обработке данных WAL, считываемых из архива (не текущих данных). Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию равно 30 секундам. При значении, равном -1, ведомый может ждать завершения конфликтующих запросов неограниченное время. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Заметьте, что параметр `max_standby_archive_delay` определяет не максимальное время, которое отводится для выполнения каждого запроса, а максимальное общее время, за которое должны быть применены изменения из одного сегмента WAL. Таким образом, если один запрос привёл к значительной задержке при обработке сегмента WAL, остальным конфликтующим запросам будет отведено гораздо меньше времени.

`max_standby_streaming_delay` (`integer`)

В режиме горячего резерва этот параметр определяет, как долго должен ждать ведомый сервер, прежде чем отменять запросы, конфликтующие с очередными изменениями в WAL, как описано в Подразделе 25.4.2. Задержка `max_standby_streaming_delay` применяется при обработке данных WAL, поступающих при потоковой репликации. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию равно 30 секундам. При значении, равном -1, ведомый может ждать завершения конфликтующих запросов неограниченное время. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Заметьте, что параметр `max_standby_streaming_delay` определяет не максимальное время, которое отводится для выполнения каждого запроса, а максимальное общее время, за которое должны быть применены изменения из WAL после получения от главного сервера. Таким образом, если один запрос привёл к значительной задержке, остальным конфликтующим запросам будет отводиться гораздо меньше времени, пока резервный сервер не догонит главный.

`wal_receiver_create_temp_slot` (`boolean`)

Определяет, должен ли процесс-приёмник WAL создавать временный слот репликации на удалённом сервере в случаях, когда постоянный слот репликации не настроен (не задан в primary_slot_name). По умолчанию этот параметр отключён. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Если значение данного параметра меняется во время работы процесса-приёмника WAL, этому процессу посылается сигнал для отключения, и ожидается, что он перезапустится с новым значением.

`wal_receiver_status_interval` (`integer`)

Определяет минимальную частоту, с которой процесс, принимающий WAL на ведомом сервере, будет сообщать о состоянии репликации ведущему или вышестоящему ведомому, где это состояние можно наблюдать в представлении `pg_stat_replication`. В этом сообщении передаются следующие позиции в журнале предзаписи: позиция изменений записанных, изменений, сохранённых на диске, и изменений применённых. Значение параметра определяет максимальный интервал между сообщениями. Сообщения о состоянии передаются при каждом продвижении позиций записанных или сохранённых на диске изменений, или с интервалом, заданным этим параметром, если он имеет ненулевое значение. В некоторых случаях сообщения о состоянии отправляются без учёта данного параметра; например, когда завершается обработка существующего WAL или когда переменная `synchronous_commit` имеет значение `remote_apply`. Таким образом, последняя переданная позиция применённых изменений может немного отставать от фактической в текущий момент. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. Значение по умолчанию равно 10 секундам. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`hot_standby_feedback` (`boolean`)

Определяет, будет ли сервер горячего резерва сообщать ведущему или вышестоящему ведомому о запросах, которые он выполняет в данный момент. Это позволяет исключить необходимость отмены запросов, вызванную очисткой записей, но при некоторых типах нагрузки это может приводить к раздуванию базы данных на ведущем сервере. Эти сообщения о запросах будут отправляться не чаще, чем раз в интервал, задаваемый параметром `wal_receiver_status_interval`. Значение данного параметра по умолчанию — `off`. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Если используется каскадная репликация, сообщения о запросах передаются выше, пока в итоге не достигнут ведущего сервера. На промежуточных серверах эта информация больше никак не задействуется.

`wal_receiver_timeout` (`integer`)

Задаёт период времени, по истечении которого прерываются неактивные соединения репликации. Это помогает принимающему ведомому серверу обнаружить сбой ведущего или разрыв сети. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию — 60 секунд. При значении, равном нулю, тайм-аут отключается. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`wal_retrieve_retry_interval` (`integer`)

Определяет, сколько ведомый сервер должен ждать поступления данных WAL из любых источников (потоковая репликация, локальный `pg_wal` или архив WAL), прежде чем повторять попытку получения WAL. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию — 5 секунд. Задать этот параметр можно только в `postgresql.conf` или в командной строке сервера.

Этот параметр полезен в конфигурациях, когда для узла в схеме восстановления нужно регулировать время ожидания новых данных WAL. Например, при восстановлении архива можно ускорить реакцию на появление нового файла WAL, уменьшив значение этого параметра. В системе с низкой активностью WAL увеличение этого параметра приведёт к сокращению числа запросов, необходимых для отслеживания архивов WAL, что может быть полезно в облачных окружениях, где учитывается число обращений к инфраструктуре.

В логической репликации этот параметр также ограничивает частоту перезапуска неудачного рабочего процесса применения изменений репликации.

`recovery_min_apply_delay` (`integer`)

По умолчанию ведомый сервер восстанавливает записи WAL передающего настолько быстро, насколько это возможно. Иногда полезно иметь возможность задать задержку при копировании данных, например, для устранения ошибок, связанных с потерей данных. Этот параметр позволяет отложить восстановление на заданное время. Например, если установить значение `5min`, ведомый сервер будет воспроизводить фиксацию транзакции не раньше, чем через 5 минут (судя по его системным часам) после времени фиксации, сообщённого ведущим. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию равно нулю, то есть задержка не добавляется.

Возможна ситуация, когда задержка репликации между серверами превышает значение этого параметра. В этом случае дополнительная задержка не добавляется. Заметьте, что задержка вычисляется как разница между меткой времени, записанной в WAL на ведущем сервере, и текущим временем на ведомом. Запаздывание передачи, связанное с задержками в сети или каскадной репликацией, может существенно сократить реальное время ожидания. Если время на главном и ведомом сервере не синхронизировано, это может приводить к применению записей ранее ожидаемого, однако это не очень важно, потому что полезные значения этого параметра намного больше, чем обычно бывает разница во времени между двумя серверами.

Задержка применяется лишь для записей WAL, представляющих фиксацию транзакций. Остальные записи проигрываются незамедлительно, так как их эффект не будет заметен до применения соответствующей записи о фиксации транзакции, благодаря правилам видимости MVCC.

Задержка добавляется, как только восстанавливаемая база данных достигает согласованного состояния, и исключается, когда ведущий сервер переключается в режим основного. После переключения ведущий сервер завершает восстановление незамедлительно.

Записи WAL должны храниться на резервном сервере, пока они не будут применены. Таким образом, чем больше задержка, тем больший объём файлов WAL необходимо накапливать, и тем больше дискового пространства потребуется для каталога `pg_wal` на ведомом сервере.

Данный параметр предназначен для применения в конфигурациях с потоковой репликацией; однако если он задан, он будет учитываться во всех случаях, кроме восстановления после сбоя. Задержка, устанавливаемая этим параметром, влияет и на работу механизма `hot_standby_feedback`, что может привести к раздуванию базы на главном сервере; использовать данный параметр при включении этого механизма следует с осторожностью.

Предупреждение

Этот параметр влияет на синхронную репликацию, когда `synchronous_commit` имеет значение `remote_apply`; каждый `COMMIT` будет ждать подтверждения применения.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`sync_replication_slots` (`boolean`)

Позволяет физическому резервному серверу синхронизировать слоты логической репликации для отработки отказа с ведущим сервером, чтобы подписчики логической репликации могли возобновить репликацию с нового ведущего сервера после отработки отказа.

По умолчанию отключён. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

18.6.4. Подписчики

Эти параметры управляют поведением подписчика логической репликации. На публикующем сервере они не играют роли. За подробностями обратитесь к Разделу 28.11

`max_replication_slots` (`integer`)

Указывает, сколько источников репликации (см. Главу 48) можно отслеживать одновременно, по сути ограничивая количество подписок на логическую репликацию, которые могут быть созданы на сервере. Если установленное значение будет меньше, чем текущее количество отслеживаемых источников репликации (показываемое в pg_replication_origin_status), сервер не запустится. Значение параметра `max_replication_slots` должно быть не менее числа подписок, добавляемого подписчику, плюс некоторый резерв для синхронизации таблиц.

Обратите внимание, что этот параметр также применяется на стороне передающего сервера, но с другим значением.

`max_logical_replication_workers` (`integer`)

Задаёт максимально возможное число рабочих процессов логической репликации. В это число входят как ведущие и параллельные рабочие процессы, применяющие изменения, так и процессы, синхронизирующие таблицы.

Рабочие процессы логической репликации берутся из пула, контролируемого параметром `max_worker_processes`.

Значение по умолчанию — 4. Этот параметр можно задать только при запуске сервера.

`max_sync_workers_per_subscription` (`integer`)

Максимальное число рабочих процессов, выполняющих синхронизацию, для одной подписки. Этот параметр управляет степенью распараллеливания копирования начальных данных в процессе инициализации подписки или при добавлении новых таблиц.

В настоящее время одну таблицу может обрабатывать только один рабочий процесс синхронизации.

Рабочие процессы синхронизации берутся из пула, контролируемого параметром `max_logical_replication_workers`.

Значение по умолчанию — 2. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`max_parallel_apply_workers_per_subscription` (`integer`)

Максимальное число параллельных рабочих процессов, применяющих изменения, для одной подписки. Этот параметр управляет степенью распараллеливания для потоковой передачи незавершённых транзакций с параметром подписки `streaming = parallel`.

Рабочие процессы применения изменений берутся из пула, контролируемого параметром `max_logical_replication_workers`.

Значение по умолчанию — 2. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

18.7. Планирование запросов

18.7.1. Конфигурация методов планировщика

Эти параметры конфигурации дают возможность грубо влиять на планы, выбираемые оптимизатором запросов. Если автоматически выбранный оптимизатором план конкретного запроса оказался неоптимальным, в качестве *временного* решения можно воспользоваться одним из этих параметров и вынудить планировщик выбрать другой план. Улучшить качество планов, выбираемых планировщиком, можно и более подходящими способами, в частности, скорректировать константы стоимости (см. Подраздел 18.7.2), выполнить `ANALYZE` вручную, увеличить значение параметра конфигурации default_statistics_target и увеличить объём статистики, собираемой для отдельных столбцов, воспользовавшись командой `ALTER TABLE SET STATISTICS`.

`enable_any_to_lateral_transformation` (`boolean`)

> Включает или отключает преобразование подзапросов `ANY` в соединения с ключевым словом `LATERAL`. По умолчанию имеет значение `on` (вкл.).

`enable_async_append` (`boolean`)

> Включает или отключает использование планировщиком планов с асинхронным добавлением данных. По умолчанию имеет значение `on` (вкл.).

`enable_bitmapscan` (`boolean`)

> Включает или отключает использование планов сканирования по битовой карте. По умолчанию имеет значение `on` (вкл.).

`enable_extra_transformations` (`boolean`)

> Включает или отключает дополнительные преобразования дерева запросов. В некоторых случаях дерево запросов сворачивается за счёт «подтягивания» подзапросов. Значение по умолчанию — `on`.

`enable_gathermerge` (`boolean`)

> Включает или отключает использование планов соединения посредством сбора. По умолчанию имеет значение `on` (вкл.).

`enable_group_by_reordering` (`boolean`)

> Определяет, будет ли планировщик запросов создавать план, в котором ключи `GROUP BY` отсортированы в порядке ключей дочернего узла, например в порядке сканирования индекса. Если этот параметр отключён, планировщик запросов создаст план, в котором ключи `GROUP BY` отсортированы в соответствии с предложениями `ORDER BY` при его наличии. Когда этот параметр включён, планировщик пытается создать более эффективный план. Значение по умолчанию — `on`.

`enable_hashagg` (`boolean`)

> Включает или отключает использование планов агрегирования по хешу. По умолчанию имеет значение `on` (вкл.).

`enable_alternative_sorting_cost_model` (`boolean`)

> Включает или отключает использование планировщиком альтернативной модели подсчёта стоимости сортировки кортежей. Если этот параметр отключён, планировщик будет использовать стандартную модель. По умолчанию: `on` (вкл.).

`enable_hashjoin` (`boolean`)

> Включает или отключает использование планов соединения по хешу. По умолчанию имеет значение `on` (вкл.).

`enable_incremental_sort` (`boolean`)

> Включает или отключает использование планировщиком инкрементальной сортировки. По умолчанию имеет значение `on` (вкл.).

`enable_indexscan` (`boolean`)

> Включает или отключает использование планов сканирования индекса и сканирования только индекса. По умолчанию имеет значение `on` (вкл.). См. также enable_indexonlyscan.

`enable_indexonlyscan` (`boolean`)

Включает или отключает использование планов сканирования только индекса (см. Раздел 11.9). По умолчанию имеет значение `on` (вкл.). Чтобы использовать планы сканирования только индекса, также должен быть включён параметр enable_indexscan.

`enable_material` (`boolean`)

Включает или отключает использование материализации при планировании запросов. Полностью исключить материализацию невозможно, но при выключении этого параметра планировщик не будет вставлять узлы материализации, за исключением случаев, где они требуются для правильности. По умолчанию этот параметр имеет значение `on` (вкл.).

`enable_memoize` (`boolean`)

Включает или отключает использование планировщиком планов с мемоизацией для кеширования результатов параметризованного сканирования внутри соединений вложенным циклом. Этот тип плана позволяет пропускать сканирование нижележащих планов, когда результаты, соответствующие текущим параметрам, уже находятся в кеше. Менее востребованные результаты могут быть удалены из кеша, когда требуется больше места для новых записей. По умолчанию имеет значение `on` (вкл.).

`enable_mergejoin` (`boolean`)

Включает или отключает использование планов соединения слиянием. По умолчанию имеет значение `on` (вкл.).

`enable_nestloop` (`boolean`)

Включает или отключает использование планировщиком планов соединения с вложенными циклами. Полностью исключить вложенные циклы невозможно, но при выключении этого параметра планировщик не будет использовать данный метод, если можно применить другие. По умолчанию этот параметр имеет значение `on`.

`enable_parallel_append` (`boolean`)

Включает или отключает использование планировщиком планов с распараллеливанием добавления данных. По умолчанию имеет значение `on` (вкл.).

`enable_parallel_hash` (`boolean`)

Включает или отключает использование планировщиком планов соединения по хешу с распараллеливанием хеширования. Не действует, если планы соединения по хешу отключены. По умолчанию имеет значение `on` (вкл.).

`enable_parallel_temptables` (`boolean`)

Включает или отключает механизм параллельного выполнения запросов для запросов, использующих хотя бы одну временную таблицу. Включение этого механизма может как уменьшить время выполнения запросов за счёт параллельной обработки данных, так и увеличить его за счёт накладных расходов на организацию параллельной обработки. В связи с этим значение по умолчанию - `off` (выкл). Обратите внимание, что в настоящее время функциональность параллельного выполнения запросов для запросов с временным таблицами является экспериментальной, и её пока не следует использовать в производственной среде.

`enable_partition_pruning` (`boolean`)

Включает или отключает в планировщике возможность отсекать секции секционированных таблиц из планов запроса. Также влияет на возможность планировщика генерировать планы запросов, позволяющие исполнителю пропускать (игнорировать) секции при выполнении запросов. По умолчанию имеет значение `on` (вкл.). За подробностями обратитесь к Подразделу 5.12.4.

`enable_partitionwise_join` (`boolean`)

Включает или отключает использование планировщиком соединения с учётом секционирования, что позволяет выполнять соединение секционированных таблиц путём соединения соответствующих секций. Соединение с учётом секционирования в настоящее время может применяться, только когда условия соединения включают все ключи секционирования; при этом ключи должны быть одного типа данных и дочерние секции должны соответствовать один-к-одному. Если этот параметр включён, количество узлов, использование памяти которых ограничено параметром `work_mem`, указанным в окончательном плане, может линейно увеличиваться в соответствии с количеством сканируемых секций. Это может привести к значительному увеличению общего потребления памяти во время выполнения запроса. Для планирования запросов также становится необходимо гораздо больше процессорного времени и памяти. По умолчанию этот параметр выключен (`off`).

`enable_partitionwise_aggregate` (`boolean`)

Включает или отключает использование планировщиком группировки или агрегирования с учётом секционирования, что позволяет выполнять группировку или агрегирование в секционированных таблицах по отдельности для каждой секции. Если предложение `GROUP BY` не включает ключи секционирования, на уровне секций может быть выполнено только частичное агрегирование, а затем требуется итоговая обработка. Если этот параметр включён, количество узлов, использование памяти которых ограничено параметром `work_mem`, указанным в окончательном плане, может линейно увеличиваться в соответствии с количеством сканируемых секций. Это может привести к значительному увеличению общего потребления памяти во время выполнения запроса. Для планирования запросов также становится необходимо гораздо больше процессорного времени и памяти. По умолчанию этот параметр выключен (`off`).

`enable_presorted_aggregate` (`boolean`)

Определяет, будет ли планировщик запросов создавать план, в котором строки предварительно отсортированы в порядке, необходимом для агрегатных функций запроса с предложениями `ORDER BY` / `DISTINCT`. Если этот параметр отключён, планировщик запросов будет создавать план, который всегда будет требовать от исполнителя выполнять сортировку перед вызовом каждой агрегатной функции с предложениями `ORDER BY` или `DISTINCT`. Когда этот параметр включён, планировщик пытается создать более эффективный план, предоставляющий входные данные для агрегатных функций, которые предварительно отсортированы в том порядке, который они требуют для агрегирования. Значение по умолчанию — `on`.

`enable_compound_index_stats` (`boolean`)

Включает/отключает использование статистики составных индексов для оценки избирательности.

По умолчанию: `on` (вкл.)

`enable_self_join_removal` (`boolean`)

Включает или отключает оптимизацию планировщика запросов, при которой анализируется дерево запросов и замкнутые соединения заменяются семантически эквивалентными одиночными сканированиями. Учитываются только обычные таблицы. Удаление замкнутых соединений, образованных условием по уникальному столбцу, может значительно ускорить выполнение запросов, не влияя на результаты. Значение по умолчанию — `on`.

`enable_seqscan` (`boolean`)

Включает или отключает использование планировщиком планов последовательного сканирования. Полностью исключить последовательное сканирование невозможно, но при выключении этого параметра планировщик не будет использовать данный метод, если можно применить другие. По умолчанию этот параметр имеет значение `on`.

`enable_sort` (`boolean`)

Включает или отключает использование планировщиком шагов с явной сортировкой. Полностью исключить явную сортировку невозможно, но при выключении этого параметра планировщик не будет использовать данный метод, если можно применить другие. По умолчанию этот параметр имеет значение `on`.

`enable_tidscan` (`boolean`)

Включает или отключает использование планов сканирования TID. По умолчанию имеет значение `on` (вкл.).

`self_join_search_limit` (`integer`)

Задаёт максимальный размер списка ссылок на одну и ту же таблицу в запросе. Такой список создаётся для поиска замкнутых соединений с целью их удаления. Чтобы выявить замкнутое соединение, нужно проанализировать взаимосвязь каждого элемента данного списка с каждым из остальных. Ограничение размера списка предусмотрено для уменьшения быстро растущей сложности этого процесса. Размер по умолчанию — `32`.

18.7.2. Константы стоимости для планировщика

Переменные *стоимости*, описанные в данном разделе, задаются по произвольной шкале. Значение имеют только их отношения, поэтому умножение или деление всех переменных на один коэффициент никак не повлияет на выбор планировщика. По умолчанию эти переменные определяются относительно стоимости чтения последовательной страницы: то есть, переменную `seq_page_cost` удобно задать равной `1.0`, а все другие переменные стоимости определить относительно неё. Но при желании можно использовать и другую шкалу, например, выразить в миллисекундах фактическое время выполнения запросов на конкретной машине.

> **Примечание**
>
> К сожалению, какого-либо чётко определённого способа определения идеальных значений стоимости не существует. Лучше всего выбирать их как средние показатели при выполнении целого ряда разнообразных запросов, которые будет обрабатывать конкретная СУБД. Это значит, что менять их по результатам всего нескольких экспериментов очень рискованно.

`seq_page_cost` (`floating point`)

Задаёт приблизительную стоимость чтения одной страницы с диска, которое выполняется в серии последовательных чтений. Значение по умолчанию равно 1.0. Это значение можно переопределить для таблиц и индексов в определённом табличном пространстве, установив одноимённый параметр табличного пространства (см. ALTER TABLESPACE).

`random_page_cost` (`floating point`)

Задаёт приблизительную стоимость чтения одной произвольной страницы с диска. Значение по умолчанию равно 4.0. Это значение можно переопределить для таблиц и индексов в определённом табличном пространстве, установив одноимённый параметр табличного пространства (см. ALTER TABLESPACE).

При уменьшении этого значения по отношению к `seq_page_cost` система начинает предпочитать сканирование по индексу; при увеличении такое сканирование становится более дорогостоящим. Оба эти значения также можно увеличить или уменьшить одновременно, чтобы изменить стоимость операций ввода-вывода по отношению к стоимости процессорных операций, которая определяется следующими параметрами.

Произвольный доступ к механическому дисковому хранилищу обычно гораздо дороже последовательного доступа, более чем в четыре раза. Однако по умолчанию выбран небольшой коэффициент (4.0), в предположении, что большой объём данных при произвольном доступе, например, при чтении индекса, окажется в кеше. Таким образом, можно считать, что значение по умолчанию моделирует ситуацию, когда произвольный доступ в 40 раз медленнее последовательного, но 90% операций произвольного чтения удовлетворяются из кеша.

Если вы считаете, что для вашей рабочей нагрузки процент попаданий не достигает 90%, вы можете увеличить параметр random_page_cost, чтобы он больше соответствовал реальной стоимости произвольного чтения. И напротив, если ваши данные могут полностью поместиться в кеше, например, когда размер базы меньше общего объёма памяти сервера, может иметь смысл уменьшить random_page_cost. С хранилищем, у которого стоимость произвольного чтения не намного выше последовательного, как например, у твердотельных накопителей, так же лучше выбрать меньшее значение random_page_cost, например `1.1`.

Подсказка

Хотя система позволяет сделать `random_page_cost` меньше, чем `seq_page_cost`, это лишено физического смысла. Однако сделать их равными имеет смысл, если база данных полностью кешируется в ОЗУ, так как в этом случае с обращением к страницам в произвольном порядке не связаны никакие дополнительные издержки. Кроме того, для сильно загруженной базы данных оба этих параметра следует понизить по отношению к стоимости процессорных операций, так как стоимость выборки страницы, уже находящейся в ОЗУ, оказывается намного меньше, чем обычно.

`write_page_cost` (`floating point`)

Задаёт приблизительную стоимость сброса страниц временных таблиц на диск. Значение по умолчанию — 5.0. Обратите внимание, что этот параметр работает только в том случае, если включён enable_parallel_temptables.

`cpu_tuple_cost` (`floating point`)

Задаёт приблизительную стоимость обработки каждой строки при выполнении запроса. Значение по умолчанию — 0.01.

`cpu_index_tuple_cost` (`floating point`)

Задаёт приблизительную стоимость обработки каждой записи индекса при сканировании индекса. Значение по умолчанию — 0.005.

`cpu_operator_cost` (`floating point`)

Задаёт приблизительную стоимость обработки оператора или функции при выполнении запроса. Значение по умолчанию — 0.0025.

`parallel_setup_cost` (`floating point`)

Задаёт приблизительную стоимость запуска параллельных рабочих процессов. Значение по умолчанию — 1000.

`parallel_tuple_cost` (`floating point`)

Задаёт приблизительную стоимость передачи одного кортежа от параллельного рабочего процесса другому процессу. Значение по умолчанию — 0.1.

`min_parallel_table_scan_size` (`integer`)

Задаёт минимальный объём данных таблицы, подлежащий сканированию, при котором может применяться параллельное сканирование. Для параллельного последовательного сканирования объём сканируемых данных всегда равняется размеру таблицы, но когда используются индексы, этот объём обычно меньше. Если это значение задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — 8 мегабайт (`8MB`).

`min_parallel_index_scan_size` (`integer`)

Задаёт минимальный объём данных индекса, подлежащий сканированию, при котором может применяться параллельное сканирование. Заметьте, что при параллельном сканировании по индексу обычно не затрагивается весь индекс; здесь учитывается число страниц, которое по мнению планировщика будет затронуто при сканировании. Этот параметр также учитывается, когда нужно определить, может ли некоторый индекс обрабатываться при параллельной очистке. См. VACUUM. Если это значение задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — 512 килобайт (`512kB`).

`effective_cache_size` (`integer`)

Определяет представление планировщика об эффективном размере дискового кеша, доступном для одного запроса. Это представление влияет на оценку стоимости использования индекса; чем выше это значение, тем больше вероятность, что будет применяться сканирование по индексу, чем ниже, тем более вероятно, что будет выбрано последовательное сканирование. При установке этого параметра следует учитывать и объём разделяемых буферов Postgres Pro, и процент дискового кеша ядра, который будут занимать файлы данных Postgres Pro, хотя некоторые данные могут оказаться и там, и там. Кроме того, следует принять во внимание ожидаемое число параллельных запросов к разным таблицам, так как общий размер будет разделяться между ними. Этот параметр не влияет на размер разделяемой памяти, выделяемой Postgres Pro, и не задаёт размер резервируемого в ядре дискового кеша; он используется только в качестве ориентировочной оценки. При этом система не учитывает, что данные могут оставаться в дисковом кеше от запроса к запросу. Если это значение задаётся без единиц измерения, оно считается заданным в блоках (размер которых равен `BLCKSZ` байт, обычно это 8 КБ). Значение по умолчанию — 4 гигабайта (`4GB`). Если `BLCKSZ` отличен от 8 КБ, значение по умолчанию корректируется пропорционально.

`jit_above_cost` (`floating point`)

Устанавливает предел стоимости запроса, при превышении которого включается JIT-компиляция, если она поддерживается (см. Главу 29). Применение JIT занимает время при планировании, но может ускорить выполнение запроса в целом. Значение `-1` отключает JIT-компиляцию. Значение по умолчанию — `100000`.

`jit_inline_above_cost` (`floating point`)

Устанавливает предел стоимости, при превышении которого будет допускаться встраивание функций и операторов в процессе JIT-компиляции. Встраивание занимает время при планировании, но в целом может ускорить выполнение. Присваивать этому параметру значение, меньшее чем `jit_above_cost`, не имеет смысла. Значение `-1` отключает встраивание. Значение по умолчанию — `500000`.

`jit_optimize_above_cost` (`floating point`)

Устанавливает предел стоимости, при превышении которого в JIT-компилированных программах может применяться дорогостоящая оптимизация. Такая оптимизация увеличивает время планирования, но в целом может ускорить выполнение. Присваивать этому параметру значение, меньшее чем `jit_above_cost`, не имеет смысла, а при значениях, превышающих `jit_inline_above_cost`, положительный эффект маловероятен. Значение `-1` отключает дорогостоящие оптимизации. Значение по умолчанию — `500000`.

`generic_plan_fuzz_factor` (`double`)

Задаёт коэффициент расчёта стоимости плана для планировщика, повышающий вероятность того, что общий или специализированный план будет выбираться чаще. По умолчанию выставлено значение `1` — это означает, что предпочтение при планировании отдаётся общему плану, а не специализированному. Чем выше значение, тем выше вероятность того, что автоматически будет выбран специализированный план. Если для параметра plan_cache_mode выставлено значение `force_generic_plan`, то планировщик будет всегда отдавать предпочтение общему плану вне зависимости от значения этого конфигурационного параметра.

18.7.3. Генетический оптимизатор запросов

Генетический оптимизатор запросов (GEnetic Query Optimizer, GEQO) осуществляет планирование запросов, применяя эвристический поиск. Это позволяет сократить время планирования для сложных запросов (в которых соединяются множество отношений), ценой того, что иногда полученные планы уступают по качеству планам, выбираемым при полном переборе. За дополнительными сведениями обратитесь к Главе 58.

`geqo` (`boolean`)

Включает или отключает генетическую оптимизацию запросов. По умолчанию она включена. В производственной среде её лучше не отключать; более гибко управлять GEQO можно с помощью переменной `geqo_threshold`.

`geqo_threshold` (`integer`)

Задаёт минимальное число элементов во `FROM`, при котором для планирования запроса будет привлечён генетический оптимизатор. (Заметьте, что конструкция `FULL OUTER JOIN` считается одним элементом списка `FROM`.) Значение по умолчанию — 12. Для более простых запросов часто лучше использовать обычный планировщик, производящий полный перебор, но для запросов со множеством таблиц полный перебор займёт слишком много времени, чаще гораздо больше, чем будет потеряно из-за выбора не самого эффективного плана. Таким образом, ограничение по размеру запроса даёт удобную возможность управлять GEQO.

`geqo_effort` (`integer`)

Управляет выбором между сокращением временем планирования и повышением качества плана запроса в GEQO. Это значение должна задаваться целым числом от 1 до 10. Значение по умолчанию равно пяти. Чем больше значение этого параметра, тем больше времени будет потрачено на планирование запроса, но и тем больше вероятность, что будет выбран эффективный план.

Параметр `geqo_effort` сам по себе ничего не делает, он используется только для вычисления значений по умолчанию для других переменных, влияющих на поведение GEQO (они описаны ниже). При желании эти переменные можно просто установить вручную.

`geqo_pool_size` (`integer`)

Задаёт размер пула для алгоритма GEQO, то есть число особей в генетической популяции. Это число должно быть не меньше двух, но полезные значения обычно лежат в интервале от 100 до 1000. Если оно равно нулю (это значение по умолчанию), то подходящее число выбирается, исходя из значения `geqo_effort` и числа таблиц в запросе.

`geqo_generations` (`integer`)

Задаёт число поколений для GEQO, то есть число итераций этого алгоритма. Оно должно быть не меньше единицы, но полезные значения находятся в том же диапазоне, что и размер пула. Если оно равно нулю (это значение по умолчанию), то подходящее число выбирается, исходя из `geqo_pool_size`.

`geqo_selection_bias` (`floating point`)

Задаёт интенсивность селекции для GEQO, то есть селективное давление в популяции. Допустимые значения лежат в диапазоне от 1.50 до 2.00 (это значение по умолчанию).

`geqo_seed` (`floating point`)

Задаёт начальное значение для генератора случайных чисел, который применяется в GEQO для выбора случайных путей в пространстве поиска порядка соединений. Может иметь значение от нуля (по умолчанию) до одного. При изменении этого значения меняется набор анализируемых путей, в результате чего может быть найден как более, так и менее оптимальный путь.

18.7.4. Другие параметры планировщика

`default_statistics_target` (`integer`)

Устанавливает значение ориентира статистики по умолчанию, распространяющееся на столбцы, для которых командой `ALTER TABLE SET STATISTICS` не заданы отдельные ограничения. Чем больше установленное значение, тем больше времени требуется для выполнения `ANALYZE`, но тем выше может быть качество оценок планировщика. Значение этого параметра по умолчанию — 100. За дополнительными сведениями об использовании статистики планировщиком запросов Postgres Pro обратитесь к Разделу 14.2.

`constraint_exclusion` (enum)

Управляет использованием планировщиком ограничений таблицы для оптимизации запросов. Допустимые значения `constraint_exclusion`: `on` (задействовать ограничения всех таблиц), `off` (никогда не задействовать ограничения) и `partition` (задействовать ограничения только для дочерних таблиц и подзапросов `UNION ALL`). Значение по умолчанию — `partition`. Оно часто помогает увеличить производительность, когда применяются традиционные деревья наследования.

Когда данный параметр разрешает это для таблицы, планировщик сравнивает условия запроса с ограничениями `CHECK` данной таблицы и не сканирует её, если они оказываются несовместимыми. Например:

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

Если включено исключение по ограничению, команда `SELECT` не будет сканировать таблицу `child1000`, в результате чего запрос выполнится быстрее.

В настоящее время исключение по ограничению разрешено по умолчанию только в условиях, возникающих при реализации секционирования через деревья наследования. Включение этой возможности для всех таблиц влечёт дополнительные издержки на планирование, довольно заметные для простых запросов, но может не дать никакого выигрыша. Если вы не применяете секционирование через деревья наследования, имеет смысл её полностью отключить. (Заметьте, что похожая функциональность для секционированных таблиц управляется отдельным параметром, enable_partition_pruning.)

За дополнительными сведениями о применении исключений по ограничению для секционирования таблиц обратитесь к Подразделу 5.12.5.

`cursor_tuple_fraction` (`floating point`)

Задаёт для планировщика оценку процента строк, которые будут получены через курсор. Значение по умолчанию — 0.1 (10%). При меньших значениях планировщик будет склонен использовать для курсоров планы с «быстрым стартом», позволяющие получать первые несколько строк очень быстро, хотя для выборки всех строк может уйти больше времени. При больших значениях планировщик стремится оптимизировать общее время запроса. При максимальном значении, равном 1.0, работа с курсорами планируется так же, как и обычные запросы — минимизируется только общее время, а не время получения первых строк.

`from_collapse_limit` (integer)

Задаёт максимальное число элементов в списке `FROM`, до которого планировщик будет объединять вложенные запросы с внешним запросом. При меньших значениях сокращается время планирования, но план запроса может стать менее эффективным. По умолчанию это значение равно восьми. За дополнительными сведениями обратитесь к Разделу 14.3.

Если это значение сделать равным geqo_threshold или больше, при таком объединении запросов может включиться планировщик GEQO и в результате будет получен неоптимальный план. См. Подраздел 18.7.3.

`jit` (boolean)

Определяет, может ли Postgres Pro использовать компиляцию JIT, если она поддерживается (см. Главу 29). По умолчанию он включён (`on`).

`join_collapse_limit` (`integer`)

Задаёт максимальное количество элементов в списке `FROM`, до достижения которого планировщик будет сносить в него явные конструкции `JOIN` (за исключением `FULL JOIN`). При меньших значениях сокращается время планирования, но план запроса может стать менее эффективным.

По умолчанию эта переменная имеет то же значение, что и `from_collapse_limit`, и это приемлемо в большинстве случаев. При значении, равном 1, предложения `JOIN` переставляться не будут, так что явно заданный в запросе порядок соединений определит фактический порядок, в котором будут соединяться отношения. Так как планировщик не всегда выбирает оптимальный порядок соединений, опытные пользователи могут временно задать для этой переменной значение 1, а затем явно определить желаемый порядок. За дополнительными сведениями обратитесь к Разделу 14.3.

Если это значение сделать равным geqo_threshold или больше, при таком объединении запросов может включиться планировщик GEQO и в результате будет получен неоптимальный план. См. Подраздел 18.7.3.

`plan_cache_mode` (`enum`)

Подготовленные операторы (они могут быть подготовлены явно либо неявно, как например в PL/pgSQL) могут выполняться с использованием специализированных или общих планов. Специализированные планы строятся заново для каждого выполнения с конкретным набором значений параметров, тогда как общий план не зависит от значений параметров и может использоваться многократно. Таким образом, общий план позволяет сэкономить время планирования, но он может быть неэффективным, если идеальные планы в большой степени определяются значениями параметров. Выбор между этими вариантами обычно производится автоматически, но его можно переопределить, воспользовавшись параметром `plan_cache_mode`. Он может принимать значение `auto` (по умолчанию), `force_custom_plan` (принудительно использовать специализированные планы) и `force_generic_plan` (принудительно использовать общие планы). Значение этого параметра учитывается при выполнении плана, а не при построении. За дополнительными сведениями обратитесь к PREPARE.

`recursive_worktable_factor` (`floating point`)

Задаёт оценку планировщиком среднего размера рабочей таблицы рекурсивного запроса как множитель ожидаемого размера начальной нерекурсивной части запроса. Это помогает планировщику выбрать наиболее подходящий метод для соединения этой рабочей таблицы с другими таблицами запроса. Значение по умолчанию — `10.0`. Меньшее значение, например `1.0`, может оказаться эффективнее, когда рекурсия слабо «разветвляется» от одного шага к другому, как, например, в запросах кратчайшего пути. При аналитической обработке графов оптимальные значения могут быть больше, чем значение по умолчанию.

`enable_appendorpath` (`boolean`)

Позволяет применять план `Append` для предложений `OR`. Этот параметр добавляет ещё одну стратегию для оптимизатора — применение плана `Append` для выражений, содержащих предложения `OR`. Этот план будет полезен для использования приложениями с автоматически генерируемыми запросами.

18.8. Регистрация ошибок и протоколирование работы сервера

18.8.1. Куда протоколировать

`log_destination` (`string`)

Postgres Pro поддерживает несколько методов протоколирования сообщений сервера: stderr, csvlog, jsonlog и syslog. В Windows также поддерживается eventlog. В качестве значения `log_destination` указывается один или несколько методов протоколирования, разделённых запятыми. По умолчанию используется stderr. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

Если в `log_destination` включено значение csvlog, то протоколирование ведётся в формате CSV (разделённые запятыми значения). Это удобно для программной обработки журнала. Подробнее об этом в Подразделе 18.8.4. Для вывода в формате CSV должен быть включён logging_collector.

Если в `log_destination` включено значение jsonlog, то протоколирование ведётся в формате JSON. Это удобно для программной обработки журнала. Подробнее об этом в Подразделе 18.8.5. Для вывода в формате JSON должен быть включён logging_collector.

Если присутствует указание stderr, csvlog или jsonlog, создаётся файл `current_logfiles`, в который записывается расположение файлов журнала, в настоящее время используемых сборщиком сообщений для соответствующих назначений. Это позволяет легко определить, какие файлы журнала используются в данный момент экземпляром сервера. Например, он может иметь такое содержание:

```
stderr log/postgresql.log
csvlog log/postgresql.csv
jsonlog log/postgresql.json
```

`current_logfiles` переписывается, когда при прокрутке создаётся новый файл журнала или когда изменяется значение `log_destination`. Он удаляется, когда в `log_destination` не задаётся ни stderr, ни csvlog, ни jsonlog, а также когда сборщик сообщений отключён.

> ## Примечание
>
> В большинстве систем Unix потребуется изменить конфигурацию системного демона syslog для использования варианта syslog в `log_destination`. Для указания типа протоколируемой программы (facility), Postgres Pro может использовать значения с `LOCAL0` по `LOCAL7` (см. syslog_facility). Однако на большинстве платформ конфигурация syslog по умолчанию не учитывает сообщения подобного типа. Чтобы это работало, потребуется добавить в конфигурацию демона syslog что-то подобное:
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> Для использования `eventlog` в `log_destination` на Windows, необходимо зарегистрировать источник событий и его библиотеку в операционной системе. Тогда Windows Event Viewer сможет отображать сообщения журнала событий. Подробнее в Разделе 17.12.

`logging_collector` (`boolean`)

Параметр включает сборщик сообщений (*logging collector*). Это фоновый процесс, который собирает отправленные в stderr сообщения и перенаправляет их в журнальные файлы. Такой подход зачастую более полезен чем запись в syslog, поскольку некоторые сообщения в syslog могут не попасть. (Типичный пример с сообщениями об ошибках динамического связывания, другой пример — ошибки в скриптах типа `archive_command`.) Для установки параметра требуется перезапуск сервера.

> ## Примечание
>
> Можно обойтись без сборщика сообщений и просто писать в stderr. Сообщения будут записываться в место, куда направлен поток stderr. Такой способ подойдёт только для небольших объёмов протоколирования, потому что не предоставляет удобных средств для организации ротации журнальных файлов. Кроме того, на некоторых платформах отказ от использования сборщика сообщений может привести к потере или искажению сооб-

щений, так как несколько процессов, одновременно пишущих в один журнальный файл, могут перезаписывать информацию друг друга.

Примечание

Сборщик спроектирован так, чтобы сообщения никогда не терялись. А это значит, что при очень высокой нагрузке, серверные процессы могут быть заблокированы при попытке отправить сообщения во время сбоя фонового процесса сборщика. В противоположность этому, syslog предпочитает удалять сообщения, при невозможности их записать. Поэтому часть сообщений может быть потеряна, но система не будет блокироваться.

`log_directory` (`string`)

При включённом `logging_collector`, определяет каталог, в котором создаются журнальные файлы. Можно задавать как абсолютный путь, так и относительный от каталога данных кластера. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера. Значение по умолчанию — `log`.

`log_filename` (`string`)

При включённом `logging_collector` задаёт имена журнальных файлов. Значение трактуется как строка формата в функции `strftime`, поэтому в ней можно использовать спецификаторы `%` для включения в имена файлов информации о дате и времени. (При наличии зависящих от часового пояса спецификаторов `%` будет использован пояс, заданный в log_timezone.) Поддерживаемые спецификаторы `%` похожи на те, что перечислены в описании *strftime* спецификации Open Group. Обратите внимание, что системная функция `strftime` напрямую не используется. Поэтому нестандартные, специфичные для платформы особенности не будут работать. Значение по умолчанию `postgresql-%Y-%m-%d_%H%M%S.log`.

Если для задания имени файлов не используются спецификаторы `%`, то для избежания переполнения диска, следует использовать утилиты для ротации журнальных файлов. В версиях до 8.4, при отсутствии спецификаторов `%`, PostgreSQL автоматически добавлял время в формате Epoch к имени файла. Сейчас в этом больше нет необходимости.

Если в `log_destination` включён вывод в формате CSV, то к имени журнального файла будет добавлено расширение `.csv`. (Если `log_filename` заканчивается на `.log`, то это расширение заменится на `.csv`.)

Если в `log_destination` включён вывод в формате JSON, то к имени журнального файла будет добавлено расширение `.json`. (Если `log_filename` заканчивается на `.log`, то это расширение заменится на `.json`.)

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`log_file_mode` (`integer`)

В системах Unix задаёт права доступа к журнальным файлам, при включённом `logging_collector`. (В Windows этот параметр игнорируется.) Значение параметра должно быть числовым, в формате команд `chmod` и `umask`. (Для восьмеричного формата, требуется задать лидирующий `0` (ноль).)

Права доступа по умолчанию `0600`, т. е. только владелец сервера может читать и писать в журнальные файлы. Также, может быть полезным значение `0640`, разрешающее чтение файлов членам группы. Однако чтобы установить такое значение, нужно каталог для хранения журнальных файлов (log_directory) вынести за пределы каталога данных кластера. В любом случае нежелательно открывать для всех доступ на чтение журнальных файлов, так как они могут содержать конфиденциальные данные.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`log_rotation_age` (`integer`)

При включённом `logging_collector` этот параметр определяет максимальное время жизни отдельного журнального файла, по истечении которого создаётся новый файл. Если это значение задаётся без единиц измерения, оно считается заданным в минутах. Значение по умолчанию — 24 часа. При нулевом значении смена файлов по времени не производится. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`log_rotation_size` (`integer`)

При включённом `logging_collector` этот параметр определяет максимальный размер отдельного журнального файла. При достижении этого размера создаётся новый файл. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Значение по умолчанию — 10 мегабайт. При нулевом значении смена файлов по размеру не производится. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`log_truncate_on_rotation` (`boolean`)

Если параметр `logging_collector` включён, Postgres Pro будет перезаписывать существующие журнальные файлы, а не дописывать в них. Однако перезапись при переключении на новый файл возможна только в результате ротации по времени, но не при старте сервера или ротации по размеру файла. При выключенном параметре всегда продолжается запись в существующий файл. Например, включение этого параметра в комбинации с `log_filename` равным `postgresql-%H.log`, приведёт к генерации 24-х часовых журнальных файлов, которые циклически перезаписываются. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

Пример: для хранения журнальных файлов в течение 7 дней, по одному файлу на каждый день с именами вида `server_log.Mon`, `server_log.Tue` и т. д., а также с автоматической перезаписью файлов прошлой недели, нужно установить `log_filename` в `server_log.%a`, `log_truncate_on_rotation` в `on` и `log_rotation_age` в `1440`.

Пример: для хранения журнальных файлов в течение 24 часов, по одному файлу на час, с дополнительной возможностью переключения файла при превышения 1ГБ, установите `log_filename` в `server_log.%H%M`, `log_truncate_on_rotation` в `on`, `log_rotation_age` в `60` и `log_rotation_size` в `1000000`. Добавление `%M` в `log_filename` позволит при переключении по размеру указать другое имя файла в пределах одного часа.

`syslog_facility` (`enum`)

При включённом протоколировании в syslog, этот параметр определяет значение «facility». Допустимые значения `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7`. По умолчанию используется `LOCAL0`. Подробнее в документации на системный демон syslog. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

`syslog_ident` (`string`)

При включённом протоколировании в syslog, этот параметр задаёт имя программы, которое будет использоваться в syslog для идентификации сообщений относящихся к Postgres Pro. По умолчанию используется `postgres`. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`syslog_sequence_numbers` (`boolean`)

Когда сообщения выводятся в syslog и этот параметр включён (по умолчанию), все сообщения будут предваряться последовательно увеличивающимися номерами (например, `[2]`). Это позволяет обойти подавление повторов «--- последнее сообщение повторилось N раз ---», которое по умолчанию осуществляется во многих реализациях syslog. В более современных реализациях syslog подавление повторных сообщений можно настроить (например, в rsyslog есть директива `$RepeatedMsgReduction`), так что это может излишне. Если же вы действительно хотите, чтобы повторные сообщения подавлялись, вы можете отключить этот параметр.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`syslog_split_messages` (`boolean`)

Когда активен вывод сообщений в syslog, этот параметр определяет, как будут доставляться сообщения. Если он включён (по умолчанию), сообщения разделяются по строкам, а длинные строки разбиваются на строки не длиннее 1024 байт, что составляет типичное ограничение размера для традиционных реализаций syslog. Когда он отключён, сообщения сервера Postgres Pro передаются службе syslog как есть, и она должна сама корректно воспринять потенциально длинные сообщения.

Если syslog в итоге выводит сообщения в текстовый файл, результат будет тем же и лучше оставить этот параметр включённым, так как многие реализации syslog не способны обрабатывать большие сообщения или их нужно специально настраивать для этого. Но если syslog направляет сообщения в некоторую другую среду, может потребоваться или будет удобнее сохранять логическую целостность сообщений.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`event_source` (`string`)

При включённом протоколировании в event log, этот параметр задаёт имя программы, которое будет использоваться в журнале событий для идентификации сообщений относящихся к Postgres Pro. По умолчанию используется `Postgres Pro`. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера.

18.8.2. Когда протоколировать

`log_min_messages` (`enum`)

Управляет минимальным уровнем сообщений, записываемых в журнал сервера. Допустимые значения `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` и `PANIC`. Каждый из перечисленных уровней включает все идущие после него. Чем дальше в этом списке уровень сообщения, тем меньше сообщений будет записано в журнал сервера. По умолчанию используется `WARNING`. Обратите внимание, позиция `LOG` здесь отличается от принятой в client_min_messages. Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

`log_min_error_statement` (`enum`)

Управляет тем, какие SQL-операторы, завершившиеся ошибкой, записываются в журнал сервера. SQL-оператор будет записан в журнал, если он завершится ошибкой с указанным уровнем важности или выше. Допустимые значения: `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` и `PANIC`. По умолчанию используется `ERROR`. Это означает, что в журнал сервера будут записаны все операторы, завершившиеся сообщением с уровнем важности `ERROR`, `LOG`, `FATAL` и `PANIC`. Чтобы фактически отключить запись операторов с ошибками, установите для этого параметра значение `PANIC`. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`log_min_duration_statement` (`integer`)

Записывает в журнал продолжительность выполнения всех команд, время работы которых не меньше указанного. Например, при значении `250ms` в журнал сервера будут записаны все команды, выполняющиеся 250 миллисекунд и дольше. С помощью этого параметра можно выявить неоптимизированные запросы в приложениях. Если значение этого параметра задаётся без единиц измерения, оно считается заданным в миллисекундах. При нулевом значении записывается продолжительность выполнения всех команд. Со значением -1 (по умолчанию) запись полностью отключается. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Этот параметр переопределяет log_min_duration_sample, то есть запросы с длительностью, превышающей заданное значение, всегда фиксируются в журнале, вне зависимости от параметров извлечения выборки.

Для клиентов, использующих расширенный протокол запросов, будет записываться продолжительность фаз: разбор, связывание и выполнение.

Примечание

При использовании совместно с log_statement, текст SQL-операторов будет записываться только один раз (от использования `log_statement`) и не будет задублирован в сообщении о длительности выполнения. Если не используется вывод в syslog, то рекомендуется в log_line_prefix включить идентификатор процесса или сеанса. Это позволит связать текст запроса с записью о продолжительности выполнения, которая появится позже.

`log_min_duration_sample` (`integer`)

Позволяет сделать выборку по продолжительности команд, которые выполнялись не менее чем определённое время. При этом в журнал будут вноситься такие же записи, как и при включённом параметре log_min_duration_statement, но не для всех команд, а только для их подмножества, ограничиваемого параметром log_statement_sample_rate. Например, при значении `100ms` предварительно для выборки будут отобраны все SQL-операторы, выполняющиеся 100 миллисекунд и дольше. Этот параметр может быть полезен, когда количество запросов слишком велико, чтобы записывать в журнал их все. Если значение этого параметра задаётся без единиц измерения, оно считается заданным в миллисекундах. При нулевом значении для выборки отбираются команды с любой продолжительностью. Со значением -1 (по умолчанию) формирование выборки по продолжительности полностью отключается. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Этот параметр имеет меньший приоритет, чем `log_min_duration_statement`, то есть команды с длительностью, превышающей `log_min_duration_statement`, будут регистрироваться в журнале всегда, вне зависимости от того, какой будет выборка.

Другие замечания, относящиеся к `log_min_duration_statement`, применимы так же и к данному параметру.

`log_statement_sample_rate` (`floating point`)

Определяет, какая доля команд с длительностью, достигшей log_min_duration_sample, будет регистрироваться в журнале. Выборка формируется вероятностным образом, например, со значением `0.5` можно считать, что шанс попадания каждой отдельной команды в выборку равен один к двум. Значение по умолчанию — `1.0`, то есть выбираются и регистрируются все команды. Со значением 0 запись команд выборки, в зависимости от их длительности, отключается, так же как и при `log_min_duration_sample`, равном `-1`. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`log_transaction_sample_rate` (`floating point`)

Задаёт долю транзакций, команды из которых будут записываться в журнал дополнительно (помимо команд, записываемых по другим причинам). Этот параметр действует на все транзакции, независимо от длительности команд. Выборка осуществляется вероятностным образом, например, со значением `0.1` можно считать, что каждая транзакция может попасть в журнал с шансом один к десяти. Параметр `log_transaction_sample_rate` может быть полезен для анализа выборки транзакций. Значение по умолчанию — `0`, то есть команды из дополнительно выбираемых транзакций не записываются. При значении `1` записываются все команды из всех транзакций. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Примечание

С этим параметром, как и со всеми остальными, управляющими журналированием команд, могут быть связаны значительные издержки.

`log_startup_progress_interval` (`integer`)

Задаёт интервал времени, по истечении которого процесс запуска будет выдавать сообщение о длительной операции, которая всё ещё выполняется; с таким же интервалом будут выдаваться и последующие сообщения об этой операции. Значение по умолчанию — 10 секунд. Значение `0` отключает эту функцию. Если это значение указано без единиц измерения, оно считается заданным в миллисекундах. Этот интервал отсчитывается для каждой операции в отдельности. Задать этот параметр можно только в файле `postgresql.conf` или в командной строке сервера.

Например, при значении по умолчанию, равном 10 сек., если синхронизация каталога данных продолжается 25 секунд, а после этого сброс нежурналируемых отношений выполняется 8 секунд, в журнал будет выдано сообщение об операции синхронизации через 10 секунд после её начала, а затем ещё одно сообщение об этой операции, когда её продолжительность достигнет 20 секунд, а сообщений об операции сброса нежурналируемых отношений в журнале не будет.

В Таблице 18.2 поясняются уровни важности сообщений в Postgres Pro. Также в этой таблице показано, как эти уровни транслируются в системные при использовании syslog или eventlog в Windows.

Таблица 18.2. Уровни важности сообщений

Уровень	Использование	syslog	eventlog
`DEBUG1 .. DEBUG5`	Более детальная информация для разработчиков. Чем больше номер, тем детальнее.	`DEBUG`	`INFORMATION`
`INFO`	Неявно запрошенная пользователем информация, например вывод команды `VACUUM VERBOSE`.	`INFO`	`INFORMATION`
`NOTICE`	Информация, которая может быть полезной пользователям. Например, уведомления об усечении длинных идентификаторов.	`NOTICE`	`INFORMATION`
`WARNING`	Предупреждения о возможных проблемах. Например, `COMMIT` вне транзакционного блока.	`NOTICE`	`WARNING`
`ERROR`	Сообщает об ошибке, из-за которой прервана текущая команда.	`WARNING`	`ERROR`
`LOG`	Информация, полезная для администраторов. Например, выполнение контрольных точек.	`INFO`	`INFORMATION`
`FATAL`	Сообщает об ошибке, из-за которой прерван текущий сеанс.	`ERR`	`ERROR`
`PANIC`	Сообщает об ошибке, из-за которой прерваны все сеансы.	`CRIT`	`ERROR`

18.8.3. Что протоколировать

> **Примечание**
>
> Выбирая, что будет записываться в протокол, важно учитывать возможные риски безопасности; подробнее об этом говорится в Разделе 23.3.

`application_name` (`string`)

: `application_name` — это любая строка длиной не более `NAMEDATALEN` символов (64 символа при стандартной сборке). Обычно устанавливается приложением при подключении к серверу. Значение отображается в представлении `pg_stat_activity` и добавляется в журнал сервера, при использовании формата CSV. Для прочих форматов, `application_name` можно добавить в журнал через параметр log_line_prefix. Значение `application_name` может содержать только печатные ASCII символы. Остальные символы заменяются шестнадцатеричными спецсимволами в стиле C.

`debug_print_parse` (`boolean`)
`debug_print_rewritten` (`boolean`)
`debug_print_plan` (`boolean`)

: Эти параметры включают вывод различной отладочной информации. А именно: вывод дерева запроса, дерево запроса после применения правил или плана выполнения запроса, соответственно. Все эти сообщения имеют уровень `LOG`. Поэтому, по умолчанию, они записываются в журнал сервера, но не отправляются клиенту. Отправку клиенту можно настроить через client_min_messages и/или log_min_messages. По умолчанию параметры выключены.

`debug_pretty_print` (`boolean`)

: Включает выравнивание сообщений, выводимых `debug_print_parse`, `debug_print_rewritten` или `debug_print_plan`. В результате сообщения легче читать, но они значительно длиннее, чем в формате «compact», который используется при выключенном значении. По умолчанию включён.

`log_autovacuum_min_duration` (`integer`)

: Задаёт время выполнения действия автоочистки, при превышении которого информация об этом действии записывается в журнал. При нулевом значении в журнале фиксируются все действия автоочистки. Значение `-1` отключает журналирование действий автоочистки. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Например, если задать значение `250ms`, в журнале будут фиксироваться все операции автоматической очистки и анализа, выполняемые дольше 250 мс. Кроме того, когда этот параметр имеет любое значение, отличное от `-1`, в журнал будет записываться сообщение в случае пропуска действия автоочистки из-за конфликтующей блокировки или параллельного удаления отношения. Значение по умолчанию — `10min`. Во включённом состоянии этот параметр помогает отслеживать активность автоочистки. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако его можно переопределить для отдельных таблиц, изменив их параметры хранения.

`log_checkpoints` (`boolean`)

: Включает протоколирование выполнения контрольных точек и точек перезапуска сервера. При этом записывается некоторая статистическая информация. Например, число записанных буферов и время, затраченное на их запись. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера. По умолчанию включён.

`log_connections` (`boolean`)

: Включает протоколирование всех попыток подключения к серверу, в том числе успешного завершения как аутентификации (если она требуется), так и авторизации клиентов. Изменить его можно только в начале сеанса и сделать это могут только суперпользователи и пользователи с соответствующим правом `SET`. Значение по умолчанию — `off`.

> ### Примечание
>
> Некоторые программы, например psql, предпринимают две попытки подключения (первая для определения, нужен ли пароль). Поэтому дублирование сообщения «connection received» не обязательно говорит о наличии проблемы.

`log_disconnections` (boolean)

Включает протоколирование завершения сеанса. В журнал выводится примерно та же информация, что и с `log_connections`, плюс длительность сеанса. Изменить этот параметр можно только в начале сеанса и сделать это могут только суперпользователи и пользователи с соответствующим правом `SET`. Значение по умолчанию — `off`.

`log_duration` (boolean)

Записывает продолжительность каждой завершённой команды. По умолчанию выключен. Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

Для клиентов, использующих расширенный протокол запросов, будет записываться продолжительность фаз: разбор, связывание и выполнение.

> ### Примечание
>
> Включение параметра `log_duration` не равнозначно установке нулевого значения для log_min_duration_statement. Разница в том, что при превышении значения `log_min_duration_statement` в журнал записывается текст запроса, а при включении данного параметра — нет. Таким образом, при `log_duration` = `on` и положительном значении `log_min_duration_statement` в журнал записывается длительность для всех команд, а текст запроса — только для команд с длительностью, превышающей предел. Такое поведение может оказаться полезным при сборе статистики в условиях большой нагрузки.

`log_error_verbosity` (enum)

Управляет количеством детальной информации, записываемой в журнал сервера для каждого сообщения. Допустимые значения: `TERSE`, `DEFAULT` и `VERBOSE`. Каждое последующее значение добавляет больше полей в выводимое сообщение. Для `TERSE` из сообщения об ошибке исключаются поля `DETAIL`, `HINT`, `QUERY` и `CONTEXT`. Для `VERBOSE` в сообщение включается код ошибки `SQLSTATE` (см. Приложение A), а также имя файла с исходным кодом, имя функции и номер строки сгенерировавшей ошибку. Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

`log_hostname` (boolean)

По умолчанию, сообщения журнала содержат лишь IP-адрес подключившегося клиента. При включении этого параметра, дополнительно будет фиксироваться и имя сервера. Обратите внимание, что в зависимости от применяемого способа разрешения имён, это может отрицательно сказаться на производительности. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`log_line_prefix` (string)

Строка, в стиле функции `printf`, которая выводится в начале каждой строки журнала сообщений. С символов `%` начинаются управляющие последовательности, которые заменяются статусной информацией, описанной ниже. Неизвестные управляющие последовательности игнорируются. Все остальные символы напрямую копируются в выводимую строку. Некоторые управляющие последовательности используются только для пользовательских процессов и будут игнорироваться фоновыми процессами, например основным процессом сервера. Статусная информация может быть выровнена по ширине влево или вправо указанием числа после % и перед кодом последовательности. Отрицательное число дополняет значение пробелами справа до заданной ширины, а положительное число — слева. Выравнивание может быть полезно для улучшения читаемости.

Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. По умолчанию этот параметр имеет значение `'%m [%p] '`, с которым в журнал выводится время и идентификатор процесса (PID).

Спецсимвол	**Назначение**	**Только для пользовательского процесса**
`%a`	Имя приложения (application_name)	да
`%u`	Имя пользователя	да
`%d`	Имя базы данных	да
`%r`	Имя удалённого узла или IP-адрес, а также номер порта	да
`%h`	Имя удалённого узла или IP-адрес	да
`%b`	Тип обслуживающего процесса	нет
`%p`	Идентификатор процесса	нет
`%P`	Идентификатор ведущего процесса группы, если текущий процесс является исполнителем параллельного запроса	нет
`%t`	Штамп времени, без миллисекунд	нет
`%m`	Штамп времени, с миллисекундами	нет
`%n`	Штамп времени, с миллисекундами (в виде времени Unix)	нет
`%i`	Тег команды: тип текущей команды в сеансе	да
`%e`	Код ошибки SQLSTATE	нет
`%c`	Идентификатор сеанса. Подробности ниже	нет
`%l`	Номер строки журнала для каждого сеанса или процесса. Начинается с 1	нет
`%s`	Штамп времени начала процесса	нет
`%v`	Виртуальный идентификатор транзакции (procNumber/localXID); см. Раздел 64.1	нет
`%x`	Идентификатор транзакции (0 если не присвоен); см. Раздел 64.1	нет

Спецсимвол	**Назначение**	**Только для пользовательского процесса**
`%q`	Ничего не выводит. Непользовательские процессы останавливаются в этой точке. Игнорируется пользовательскими процессами	нет
`%Q`	Идентификатор текущего запроса. Вычисление идентификаторов запроса по умолчанию отключено, поэтому в этом поле будет выводиться значение 0, если не включён параметр compute_query_id или не настроен дополнительный модуль, вычисляющий идентификаторы запросов.	да
`%%`	Выводит `%`	нет

Тип обслуживающего процесса соответствует содержимому столбца `backend_type` в представлении `pg_stat_activity`, но в журнале могут фигурировать и другие типы, которые не показываются в этом представлении.

Спецпоследовательность `%c` заменяется на практически уникальный идентификатор сеанса, содержащий из 4 шестнадцатеричных чисел (без ведущих нулей), разделённых точками. Эти числа представляют время запуска процесса и его PID, так что `%c` можно использовать как более компактную альтернативу совокупности этих двух элементов. Чтобы получить такой идентификатор сеанса, например из `pg_stat_activity`, воспользуйтесь этим запросом:

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

Подсказка

Последним символом в `log_line_prefix` лучше оставлять пробел, чтобы отделить от остальной строки. Можно использовать и символы пунктуации.

Подсказка

Syslog также формирует штамп времени и идентификатор процесса, поэтому вероятно нет смысла использовать соответствующие управляющие последовательности при использовании syslog.

Подсказка

Спецпоследовательность `%q` полезна для включения информации, которая существует только в контексте сеанса (обслуживающего процесса), например, имя пользователя или базы данных. Например:

```
log_line_prefix = '%m [%p] %q%u@%d/%a '
```

Примечание

Спецпоследовательность `%Q` всегда выдаёт нулевой идентификатор для строк, выводимых посредством log_statement, потому что соответствующий вывод `log_statement` формируется до вычисления этого идентификатора, в том числе и для некорректных операторов, для которых идентификатор вычислить нельзя.

`log_lock_waits` (`boolean`)

Определяет, нужно ли фиксировать в журнале события, когда сеанс ожидает получения блокировки дольше, чем указано в deadlock_timeout. Это позволяет выяснить, не связана ли низкая производительность с ожиданием блокировок. По умолчанию отключён. Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

`log_recovery_conflict_waits` (`boolean`)

Определяет, нужно ли фиксировать в журнале события, когда сеанс ожидает получения блокировки дольше, чем указано в `deadlock_timeout` для конфликтов восстановления. Это позволяет выяснить, препятствуют ли конфликты восстановления применению WAL.

Значение по умолчанию — `off` (выкл.). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`log_parameter_max_length` (`integer`)

Положительное значение устанавливает количество байт, до которого будут усекаться значения привязанных SQL-параметров, выводимые в сообщениях вместе с SQL-операторами (это не относится к регистрации ошибок). Ноль отключает вывод значений привязанных параметров в таких сообщениях. Значение `-1` (по умолчанию) позволяет получить в журнале привязанные параметры в полном объёме. Если значение этого параметра задаётся без единиц измерения, оно считается заданным в байтах. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Этот параметр влияет только на сообщения, выводимые в журнал по критериям, которые устанавливают параметры log_statement, log_duration и связанные с ними. С ненулевыми значениями этого параметра сопряжены некоторые издержки, особенно если привязанные значения передаются в двоичной форме, так как их нужно будет переводить в текстовый вид.

`log_parameter_max_length_on_error` (`integer`)

Положительное значение устанавливает количество байт, до которого будут усекаться значения привязанных SQL-параметров, выводимые вместе с SQL-операторами при регистрации ошибок. Нулевое значение отключает вывод значений привязанных операторов в сообщениях об ошибках. Значение `-1` (по умолчанию) позволяет получить в журнале привязанные параметры в полном объёме. Если значение этого параметра задаётся без единиц измерения, оно считается заданным в байтах.

С ненулевыми значениями этого параметра сопряжены некоторые издержки, так как Postgres Pro должен будет сформировать в памяти текстовые представления всех значений параметров перед выполнением всех операторов, в том числе, выполненных в итоге без ошибок. Издержки дополнительно возрастают, когда привязанные параметры передаются не в текстовой, а в двоичной форме, так как их недостаточно просто скопировать в строку, их нужно ещё преобразовать.

`log_statement` (`enum`)

Управляет тем, какие SQL-команды записывать в журнал. Допустимые значения: `none` (отключено), `ddl`, `mod` и `all` (все команды). `ddl` записывает все команды определения данных, такие как `CREATE`, `ALTER`, `DROP`. `mod` записывает все команды `ddl`, а также команды изменяющие данные, такие как `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` и `COPY FROM`. `PREPARE`, `EXECUTE` и `EXPLAIN ANALYZE` также записываются, если вызваны для команды соответствующего типа. Если клиент использует расширенный протокол запросов, то запись происходит на фазе выполнения и содержит значения всех связанных переменных (если есть символы одиночных кавычек, то они дублируются).

По умолчанию `none`. Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

Примечание

Команды с синтаксическими ошибками не записываются, даже если `log_statement = all`, так как сообщение формируется только после выполнения предварительного разбора, определяющего тип команды. При расширенном протоколе запросов, похожим образом не будут записываться команды, неуспешно завершившиеся до фазы выполнения (например, при разборе или построении плана запроса). Для включения в журнал таких команд установите `log_min_error_statement` в `ERROR` (или ниже).

Записываемые в журнал команды могут раскрывать конфиденциальные данные и даже содержать незашифрованные пароли.

`log_replication_commands` (`boolean`)

Включает запись в журнал сервера всех команд репликации и всех случаев получения/освобождения слота репликации процессом `walsender`. Подробнее о командах репликации можно узнать в Разделе 53.4. Значение по умолчанию — `off`. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`log_temp_files` (`integer`)

Управляет регистрацией в журнале имён и размеров временных файлов. Временные файлы могут использоваться для сортировки, хеширования и временного хранения результатов запросов. Когда этот параметр включён, при удалении временного файла информация о нём может записываться в журнал с указанием размера файла в байтах. При нулевом значении записывается информация обо всех файлах, а при положительном — о файлах, размер которых не меньше заданной величины. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Значение по умолчанию равно -1, то есть запись такой информации отключена. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`log_timezone` (`string`)

Устанавливает часовой пояс для штампов времени при записи в журнал сервера. В отличие от TimeZone, это значение одинаково для всех баз данных кластера, поэтому для всех сеансов используются согласованные значения штампов времени. Встроенное значение по умолчанию `GMT`, но оно переопределяется в `postgresql.conf`: initdb записывает в него значение, соответствующее системной среде. Подробнее об этом в Подразделе 8.5.3. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

18.8.4. Использование вывода журнала в формате CSV

Добавление `csvlog` в `log_destination` делает удобным загрузку журнальных файлов в таблицу базы данных. Строки журнала представляют собой значения, разделённые запятыми (формат CSV), со следующими полями: штамп времени с миллисекундами; имя пользователя; имя базы данных; идентификатор процесса; клиентский узел:номер порта; идентификатор сеанса; номер строки каждого сеанса; тег команды; время начала сеанса; виртуальный идентификатор транзакции; идентификатор транзакции; уровень важности ошибки; код ошибки SQLSTATE; сообщение об ошибке; подробности к сообщению об ошибке; подсказка к сообщению об ошибке; внутренний запрос, приведший к ошибке (если есть); номер символа внутреннего запроса, где произошла ошибка; контекст ошибки; запрос пользователя, приведший к ошибке (если есть и включён `log_min_error_statement`); номер символа в запросе пользователя, где произошла ошибка; расположение ошибки в исходном коде Postgres Pro (если `log_error_verbosity` установлен в `verbose`), имя приложения, тип обслуживающего процесса, идентификатор ведущего процесса группы и идентификатор запроса. Вот пример определения таблицы, для хранения журналов в формате CSV:

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text,
  leader_pid integer,
  query_id bigint,
  PRIMARY KEY (session_id, session_line_num)
);
```

Для загрузки журнального файла в такую таблицу можно использовать команду `COPY FROM`:

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

Также можно обратиться к этому файлу как к сторонней таблице, используя поставляемый модуль file_fdw.

Для упрощения загрузки журналов в CSV формате используйте следующее:

1. Установите для `log_filename` и `log_rotation_age` значения, гарантирующие согласованную и предсказуемую схему именования журнальных файлов. Зная, какие имена будут у журнальных файлов, можно определить, когда конкретный файл заполнен и готов к загрузке.

2. Установите `log_rotation_size` в 0, чтобы запретить ротацию файлов по достижении определённого размера, так как это делает непредсказуемой схему именования журнальных файлов.

3. Установите `log_truncate_on_rotation` в `on`, чтобы новые сообщения не смешивались со старыми при переключении на существующий файл.

4. Определение таблицы содержит первичный ключ. Это полезно для предотвращения случайной повторной загрузки данных. Команда `COPY` фиксирует изменения один раз, поэтому любая ошибка приведёт к отмене всей загрузки. Если сначала загрузить неполный журнальный файл, то его повторная загрузка (по заполнении) приведёт к нарушению первичного ключа и, следо-

вательно, к ошибке загрузки. Поэтому необходимо дожидаться окончания записи в журнальный файл перед началом загрузки. Похожим образом предотвращается случайная загрузка частично сформированной строки сообщения, что также приведёт к сбою в команде `COPY`.

18.8.5. Использование вывода журнала в формате JSON

Добавление `jsonlog` в список `log_destination` позволяет легко импортировать файлы журналов в различные программы. Когда выбран этот метод, строки журнала выводятся в формате JSON.

В этом формате строковые поля, содержащие NULL, не выводятся. Описанный ниже набор полей не окончательный, в будущем могут появиться новые поля. При обработке вывода `jsonlog` в пользовательских приложениях поля, неизвестные для этих приложений, должны игнорироваться.

Каждая строка журнала сериализуется в виде объекта JSON с набором ключей и связанных с ними значений, перечисленных в Таблица 18.3.

Таблица 18.3. Ключи и значения записей журнала в формате JSON

Имя ключа	Тип	Описание
`timestamp`	строка	Штамп времени, с миллисекундами
`user`	строка	Имя пользователя
`dbname`	строка	Имя базы данных
`pid`	число	Идентификатор процесса
`remote_host`	строка	Компьютер клиента
`remote_port`	число	Порт клиента
`session_id`	строка	Идентификатор сеанса
`line_num`	число	Номер строки внутри сеанса
`ps`	строка	Текущая строка для ps
`session_start`	строка	Время начала сеанса
`vxid`	строка	Идентификатор виртуальной транзакции
`txid`	строка	Идентификатор обычной транзакции
`error_severity`	строка	Уровень ошибки
`state_code`	строка	Код SQLSTATE
`message`	строка	Сообщение об ошибке
`detail`	строка	Подробное сообщение об ошибке
`hint`	строка	Подсказка, связанная с ошибкой
`internal_query`	строка	Внутренний запрос, вызвавший ошибку
`internal_position`	число	Положение курсора в тексте внутреннего запроса
`context`	строка	Контекст ошибки
`statement`	строка	Строка запроса, предоставленная клиентом
`cursor_position`	число	Положение курсора в строке запроса

Имя ключа	Тип	Описание
`func_name`	строка	Имя функции, в которой произошла ошибка
`file_name`	строка	Имя файла, в котором произошла ошибка
`file_line_num`	число	Номер строки в файле, где произошла ошибка
`application_name`	строка	Имя клиентского приложения
`backend_type`	строка	Тип обслуживающего процесса
`leader_pid`	число	Идентификатор ведущего процесса группы параллельных исполнителей
`query_id`	число	Идентификатор запроса

18.8.6. Заголовок процесса

Эти параметры влияют на то, как формируются заголовки серверных процессов. Заголовок процесса обычно можно наблюдать в программах типа ps, а в Windows — в Process Explorer. За подробностями обратитесь к Разделу 26.1.

`cluster_name` (`string`)

Задаёт имя, которое идентифицирует данный кластер (экземпляр сервера) для различных целей. Имя кластера выводится в заголовке процесса для всех серверных процессов в данном кластере. Более того, это имя будет именем приложения по умолчанию, используемым при подключении ведомого сервера (см. synchronous_standby_names).

Это имя может быть любой строкой не длиннее `NAMEDATALEN` символов (64 символа в стандартной сборке). В строке `cluster_name` могут использоваться только печатаемые ASCII-символы. Остальные символы заменяются шестнадцатеричными спецсимволами в стиле C. Если этот параметр содержит пустую строку `''` (это значение по умолчанию), никакое имя не выводится. Этот параметр можно задать только при запуске сервера.

`update_process_title` (`boolean`)

Включает изменение заголовка процесса при получении сервером каждой очередной команды SQL. На большинстве платформ он включён (имеет значение `on`), но в Windows по умолчанию выключен (`off`) ввиду больших издержек на изменение этого заголовка. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

18.9. Статистика времени выполнения

18.9.1. Накопительная статистика по запросам и индексам

Эти параметры управляют системой накопительной статистики на уровне сервера. Когда сбор статистики включён, собираемые данные можно просмотреть в семействе системных представлений `pg_stat` и `pg_statio`. За дополнительными сведениями обратитесь к Главе 26.

`track_activities` (`boolean`)

Включает сбор сведений о текущих командах, выполняющихся во всех сеансах (в частности, отслеживается идентификатор и время запуска команды). По умолчанию этот параметр включён. Заметьте, что даже когда сбор ведётся, собранная информация доступна не для всех пользователей, а только для суперпользователей и пользователей, включённых в роль `pg_read_all_stats`. Также пользователям доступна информация о командах в их сеансах (включая сеансы ролей, в которые эти пользователи включены). Поэтому это не должно повлечь риски, связанные с безопасностью. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`track_activity_query_size` (`integer`)

Задаёт объём памяти, резервируемой для хранения текста выполняемой в данной момент команды в каждом активном сеансе, для поля `pg_stat_activity.query`. Если это значение задаётся без единиц измерения, оно считается заданным в байтах. Значение по умолчанию — 1024 байта. Задать этот параметр можно только при запуске сервера.

`track_counts` (`boolean`)

Включает сбор статистики активности в базе данных. Этот параметр по умолчанию включён, так как собранная информация требуется демону автоочистки. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`track_io_timing` (`boolean`)

Включает замер времени операций ввода-вывода. Этот параметр по умолчанию отключён, так как для данного замера требуется постоянно запрашивать текущее время у операционной системы, что может значительно замедлить работу на некоторых платформах. Для оценивания издержек замера времени на вашей платформе можно воспользоваться утилитой pg_test_timing. Статистику ввода-вывода можно получить через представление `pg_stat_database`, `pg_stat_io`, в выводе EXPLAIN (когда используется параметр `BUFFERS`), в выводе VACUUM (когда используется параметр `VERBOSE`), от процесса автоочистки, выполняющего операции очистки и сбора статистики, когда установлен параметр log_autovacuum_min_duration, и через представление pg_stat_statements. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`track_wal_io_timing` (`boolean`)

Включает замер времени операций ввода-вывода WAL. Этот параметр по умолчанию отключён, так как для данного замера требуется постоянно запрашивать текущее время у операционной системы, что может значительно замедлить работу на некоторых платформах. Для оценивания издержек замера времени на вашей платформе можно воспользоваться утилитой pg_test_timing. Статистику ввода-вывода можно получить через представление `pg_stat_wal`. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`track_functions` (`enum`)

Включает подсчёт вызовов функций и времени их выполнения. Значение `pl` включает отслеживание только функций на процедурном языке, а `all` — также функций на языках SQL и C. Значение по умолчанию — `none`, то есть сбор статистики по функциям отключён. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Примечание

Функции на языке SQL, достаточно простые для «внедрения» в вызывающий запрос, отслеживаться не будут вне зависимости от этого параметра.

`stats_fetch_consistency` (`enum`)

Определяет поведение при многократном обращении к накопительной статистике в рамках транзакции. Когда задано значение `none`, счётчики при каждом обращении извлекаются из общей памяти. Когда задано значение `cache`, при первом обращении кешируется статистика по конкретному объекту; сбрасывается она при завершении транзакции или при вызове `pg_stat_clear_snapshot()`. Когда задано значение `snapshot`, при первом обращении кешируется вся статистика, имеющаяся в текущей базе данных; сбрасывается она также при завершении транзакции или при вызове `pg_stat_clear_snapshot()`. При изменении этого параметра в транзакции сбрасывается снимок статистики. Значение по умолчанию — `cache`.

> **Примечание**
>
> Вариант `none` больше подходит для систем мониторинга. Он наиболее эффективен, если происходит только одно обращение к значениям. Вариант `cache` гарантирует, что при повторном обращении будут получены те же значения; это нужно для запросов, содержащих, например, замкнутые соединения. Вариант `snapshot` может быть полезен при интерактивном просмотре статистики, но с ним связаны дополнительные издержки, особенно если в базе много объектов.

18.9.2. Мониторинг статистики

`compute_query_id` (enum)

Включает вычисление идентификатора запроса в ядре. Идентификаторы запроса могут отображаться в представлении `pg_stat_activity` в выводе `EXPLAIN` или записываться в журнал, при заданным соответствующим образом параметре log_line_prefix. Идентификаторы запросов также должны вычисляться для расширения pg_stat_statements. Обратите внимание, что в качестве альтернативы можно использовать внешний модуль, если метод вычисления идентификатора запроса в ядре является неприемлемым. Это вычисление в ядре нужно полностью отключить. Допустимые значения: `off` (всегда отключено); `on` (всегда включено); `auto`, которое позволяет таким модулям, как pg_stat_statements, автоматически включить данное вычисление; `regress`, которое действует так же, как `auto`, но идентификатор запроса не показывается в выводе команды `EXPLAIN`, что облегчает автоматическое регрессионное тестирование. По умолчанию значение этого параметра — `auto`.

> **Примечание**
>
> Чтобы для запроса гарантированно вычислялся и отображался только один идентификатор, вычисляющие его расширения должны выдавать ошибку, если идентификатор запроса уже был вычислен.

`log_statement_stats` (boolean)
`log_parser_stats` (boolean)
`log_planner_stats` (boolean)
`log_executor_stats` (boolean)

Эти параметры включают вывод статистики по производительности соответствующего модуля в протокол работы сервера. Это грубый инструмент профилирования, похожий на функцию `getrusage()` в операционной системе. Параметр `log_statement_stats` включает вывод общей статистики по операторам, тогда как другие управляют статистикой по модулям (разбор, планирование, выполнение). Включить `log_statement_stats` одновременно с параметрами, управляющими модулями, нельзя. По умолчанию все эти параметры отключены. Изменить эти параметры могут только суперпользователи и пользователи с соответствующим правом `SET`.

18.10. Автоматическая очистка

Эти параметры управляют поведением механизма *автоочистки*. За дополнительными сведениями обратитесь к Подразделу 23.1.6. Заметьте, что многие из этих параметров могут быть переопределены на уровне таблиц; см. Параметры хранения.

`autovacuum` (boolean)

Управляет состоянием демона, запускающего автоочистку. По умолчанию он включён, но чтобы автоочистка работала, нужно также включить track_counts. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако автоочистку можно отключить для отдельных таблиц, изменив их параметры хранения.

Заметьте, что даже если этот параметр отключён, система будет запускать процессы автоочистки, когда это необходимо для предотвращения зацикливания идентификаторов транзакций. За дополнительными сведениями обратитесь к Подразделу 23.1.5.

`autovacuum_max_workers` (`integer`)

Задаёт максимальное число процессов автоочистки (не считая процесс, запускающий автоочистку), которые могут выполняться одновременно. По умолчанию это число равно трём. Задать этот параметр можно только при запуске сервера.

`autovacuum_naptime` (`integer`)

Задаёт минимальную задержку между двумя запусками автоочистки для отдельной базы данных. Демон автоочистки проверяет базу данных через заданный интервал времени и выдаёт команды `VACUUM` и `ANALYZE`, когда это требуется для таблиц этой базы. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. По умолчанию задержка равна одной минуте (`1min`). Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера.

`autovacuum_vacuum_threshold` (`integer`)

Задаёт минимальное число изменённых или удалённых кортежей, при котором будет выполняться `VACUUM` для отдельно взятой таблицы. Значение по умолчанию — 50 кортежей. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

`autovacuum_vacuum_insert_threshold` (`integer`)

Задаёт число добавленных кортежей, при достижении которого будет выполняться `VACUUM` для отдельно взятой таблицы. Значение по умолчанию — 1000 кортежей. При значении -1 процедура автоочистки не будет производить операции `VACUUM` с таблицами в зависимости от числа добавленных строк. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

`autovacuum_analyze_threshold` (`integer`)

Задаёт минимальное число добавленных, изменённых или удалённых кортежей, при котором будет выполняться `ANALYZE` для отдельно взятой таблицы. Значение по умолчанию — 50 кортежей. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

`autovacuum_vacuum_scale_factor` (`floating point`)

Задаёт процент от размера таблицы, который будет добавляться к `autovacuum_vacuum_threshold` при выборе порога срабатывания команды `VACUUM`. Значение по умолчанию — 0.2 (20% от размера таблицы). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

`autovacuum_vacuum_insert_scale_factor` (`floating point`)

Задаёт процент от размера таблицы, который будет добавляться к `autovacuum_vacuum_insert_threshold` при выборе порога срабатывания команды `VACUUM`. Значение по умолчанию — 0.2 (20% от размера таблицы). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

`autovacuum_analyze_scale_factor` (`floating point`)

Задаёт процент от размера таблицы, который будет добавляться к `autovacuum_analyze_threshold` при выборе порога срабатывания команды `ANALYZE`. Значение по умолчанию — 0.1 (10% от размера таблицы). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако данное значение можно переопределить для избранных таблиц, изменив их параметры хранения.

`autovacuum_freeze_max_age` (`integer`)

Задаёт максимальный возраст (в транзакциях) для поля `pg_class.relfrozenxid` некоторой таблицы, при достижении которого будет запущена операция `VACUUM` для предотвращения зацикливания идентификаторов транзакций в этой таблице. Заметьте, что система запустит процессы автоочистки для предотвращения зацикливания, даже если для всех других целей автоочистка отключена.

При очистке могут также удаляться старые файлы из подкаталога `pg_xact`, поэтому значение по умолчанию сравнительно мало — 200 миллионов транзакций. Задать этот параметр можно только при запуске сервера, но для отдельных таблиц его можно определить по-другому, изменив их параметры хранения. За дополнительными сведениями обратитесь к Подразделу 23.1.5.

`autovacuum_multixact_freeze_max_age` (`integer`)

Задаёт максимальный возраст (в мультитранзакциях) для поля `pg_class.relminmxid` таблицы, при достижении которого будет запущена операция `VACUUM` для предотвращения зацикливания идентификаторов мультитранзакций в этой таблице. Заметьте, что система запустит процессы автоочистки для предотвращения зацикливания, даже если для всех других целей автоочистка отключена.

При очистке мультитранзакций могут также удаляться старые файлы из подкаталогов `pg_multixact/members` и `pg_multixact/offsets`, поэтому значение по умолчанию сравнительно мало — 400 миллионов мультитранзакций. Этот параметр можно задать только при запуске сервера, но для отдельных таблиц его можно определить по-другому, изменив их параметры хранения. За дополнительными сведениями обратитесь к Подразделу 23.1.5.1.

`autovacuum_vacuum_cost_delay` (`floating point`)

Задаёт задержку при превышении предела стоимости, которая будет применяться при автоматических операциях `VACUUM`. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. При значении -1 применяется обычная задержка vacuum_cost_delay. Значение по умолчанию — 2 миллисекунды. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако его можно переопределить для отдельных таблиц, изменив их параметры хранения.

`autovacuum_vacuum_cost_limit` (`integer`)

Задаёт предел стоимости, который будет учитываться при автоматических операциях `VACUUM`. При значении -1 (по умолчанию) применяется обычное значение vacuum_cost_limit. Заметьте, что это значение распределяется пропорционально среди всех работающих процессов автоочистки, если их больше одного, так что сумма ограничений всех процессов никогда не превосходит данный предел. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера. Однако его можно переопределить для отдельных таблиц, изменив их параметры хранения.

18.11. Параметры клиентских сеансов по умолчанию

18.11.1. Поведение команд

`client_min_messages` (`enum`)

Управляет минимальным уровнем сообщений, посылаемых клиенту. Допустимые значения `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `LOG`, `NOTICE`, `WARNING` и `ERROR`. Каждый из перечисленных уровней включает все идущие после него. Чем дальше в этом списке уровень сообщения, тем меньше сообщений будет посылаться клиенту. По умолчанию используется `NOTICE`. Обратите внимание, позиция `LOG` здесь отличается от принятой в log_min_messages.

Сообщения уровня `INFO` передаются клиенту всегда.

`search_path` (`string`)

Эта переменная определяет порядок, в котором будут просматриваться схемы при поиске объекта (таблицы, типа данных, функции и т. д.), к которому обращаются просто по имени, без указания схемы. Если объекты с одинаковым именем находятся в нескольких схемах, использоваться будет тот, что встретится первым при просмотре пути поиска. К объекту, который не относится к схемам, перечисленным в пути поиска, можно обратиться только по полному имени (с точкой), с указанием содержащей его схемы.

Значением `search_path` должен быть список имён схем через запятую. Если для имени, указанного в этом списке, не находится существующая схема, либо пользователь не имеет права `USAGE` для схемы с этим именем, такое имя просто игнорируется.

Если список содержит специальный элемент `$user`, вместо него подставляется схема с именем, возвращаемым функцией `CURRENT_USER`, если такая схема существует и пользователь имеет право `USAGE` для неё. (В противном случае элемент `$user` игнорируется.)

Схема системных каталогов, `pg_catalog`, просматривается всегда, независимо от того, указана она в пути или нет. Если она указана в пути, она просматривается в заданном порядке. Если же `pg_catalog` отсутствует в пути, эта схема будет просматриваться *перед* остальными элементами пути.

Аналогично всегда просматривается схема временных таблиц текущего сеанса, `pg_temp_nnn`, если она существует. Её можно включить в путь поиска, указав её псевдоним `pg_temp`. Если она отсутствует в пути, она будет просматриваться первой (даже перед `pg_catalog`). Временная схема просматривается только при поиске отношений (таблиц, представлений, последовательностей и т. д.) и типов данных, но никогда при поиске функций и операторов.

Когда объекты создаются без указания определённой целевой схемы, они помещаются в первую пригодную схему, указанную в `search_path`. Если путь поиска схем пуст, выдаётся ошибка.

По умолчанию этот параметр имеет значение `"$user", public`. При таком значении поддерживается совместное использование базы данных (когда пользователи не имеют личных схем, все используют схему `public`), использование личных схем, а также комбинация обоих вариантов. Другие подходы можно реализовать, изменяя значение пути по умолчанию, либо глобально, либо индивидуально для каждого пользователя.

Более подробно обработка схем описана в Разделе 5.10. В частности, стандартная конфигурация схем подходит только для баз данных с одним пользователем или с взаимно доверяющими пользователями.

Текущее действующее значение пути поиска можно получить, воспользовавшись SQL-функцией `current_schemas` (см. Раздел 9.27). Это значение может отличаться от значения `search_path`, так как `current_schemas` показывает, как были преобразованы элементы, фигурирующие в `search_path`.

`row_security` (`boolean`)

Эта переменная определяет, должна ли выдаваться ошибка при применении политик защиты строк. Со значением `on` политики применяются в обычном режиме. Со значением `off` запросы, ограничиваемые минимум одной политикой, будут выдавать ошибку. Значение по умолчанию — `on`. Значение `off` рекомендуется, когда ограничение видимости строк чревато некорректными результатами; например, pg_dump устанавливает это значение. Эта переменная не влияет на роли, которые обходят все политики защиты строк, а именно, на суперпользователей и роли с атрибутом `BYPASSRLS`.

Подробнее о политиках защиты строк можно узнать в описании CREATE POLICY.

`default_table_access_method` (`string`)

Этот параметр задаёт табличный метод доступа по умолчанию, который будет использоваться при создании таблиц или материализованных представлений, если в команде `CREATE` не будет явно указан метод доступа, или при выполнении команды `SELECT ... INTO`, в которой явно задать метод доступа нельзя. Значение по умолчанию — `heap`.

`default_tablespace` (`string`)

Эта переменная устанавливает табличное пространство по умолчанию, в котором будут создаваться объекты (таблицы и индексы), когда в команде `CREATE` табличное пространство не указывается явно.

Её значением может быть либо имя табличного пространства, либо пустая строка, подразумевающая использование табличного пространства по умолчанию в текущей базе данных. Если табличное пространство с заданным именем не существует, Postgres Pro будет автоматически использовать табличное пространство по умолчанию. Если используется не пространство по умолчанию, пользователь должен иметь право `CREATE` для него, иначе он не сможет создавать объекты.

Эта переменная не используется для временных таблиц; для них задействуется temp_tablespaces.

Эта переменная также не используется при создании баз данных. По умолчанию, новая база данных наследует выбор табличного пространства от базы-шаблона, из которой она копируется.

Если на момент создания секционированной таблицы значение этого параметра отлично от пустого, оно задаёт табличное пространство для секционированной таблицы, в котором по умолчанию будут располагаться её секции, создаваемые в дальнейшем, даже если позже значение `default_tablespace` будет изменено.

За дополнительными сведениями о табличных пространствах обратитесь к Разделу 21.6.

`default_toast_compression` (`enum`)

Эта переменная устанавливает метод сжатия TOAST по умолчанию для значений сжимаемых столбцов. (Его можно переопределить для отдельных столбцов, установив параметр столбца `COMPRESSION` в `CREATE TABLE` или `ALTER TABLE`.) Поддерживаются методы сжатия `pglz` и `lz4` (если Postgres Pro скомпилирован с параметром `--with-lz4`). По умолчанию используется `lz4`.

`temp_tablespaces` (`string`)

Эта переменная задаёт табличные пространства, в которых будут создаваться временные объекты (временные таблицы и индексы временных таблиц), когда в команде `CREATE` табличное пространство не указывается явно. В этих табличных пространствах также создаются временные файлы для внутреннего использования, например, для сортировки больших наборов данных.

Её значение содержит список имён табличных пространств. Когда этот список содержит больше одного имени, Postgres Pro выбирает из этого списка случайный элемент при создании каждого временного объекта; однако при создании последующих объектов внутри транзакции табличные пространства перебираются последовательно. Если в этом списке оказывается пустая строка, Postgres Pro будет автоматически использовать вместо этого элемента табличное пространство по умолчанию для текущей базы данных.

Когда `temp_tablespaces` задаётся интерактивно, указание несуществующего табличного пространства считается ошибкой, как и указание табличного пространства, для которого пользователь не имеет права `CREATE`. Однако при использовании значения, заданного ранее, несуществующие табличные пространства и пространства, для которых у пользователя нет права `CREATE`, просто игнорируются. В частности, это касается значения, заданного в `postgresql.conf`.

По умолчанию значение этой переменной — пустая строка. С таким значением все временные объекты создаются в табличном пространстве по умолчанию, установленном для текущей базы данных.

См. также default_tablespace.

`check_function_bodies` (boolean)

Этот параметр обычно включён. Выключение этого параметра (присваивание ему значения `off`) отключает проверку строки с телом подпрограммы, передаваемой командам CREATE FUNCTION и CREATE PROCEDURE. Отключение проверки позволяет избежать побочных эффектов процесса проверки, в частности исключить ложные срабатывания из-за таких проблем, как ссылки вперёд. Этому параметру нужно присваивать значение `off` перед загрузкой функций от лица других пользователей; pg_dump делает это автоматически.

`default_transaction_isolation` (enum)

Для каждой транзакции в SQL устанавливается уровень изоляции: «read uncommitted», «read committed», «repeatable read» или «serializable». Этот параметр задаёт уровень изоляции, который будет устанавливаться по умолчанию для новых транзакций. Значение этого параметра по умолчанию — «read committed».

Дополнительную информацию вы можете найти в Главе 13 и SET TRANSACTION.

`default_transaction_read_only` (boolean)

SQL-транзакции в режиме «только чтение» не могут модифицировать не временные таблицы. Этот параметр определяет, будут ли новые транзакции по умолчанию иметь характеристику «только чтение». Значение этого параметра по умолчанию — `off` (допускается чтение и запись).

За дополнительной информацией обратитесь к SET TRANSACTION.

`default_transaction_deferrable` (boolean)

Транзакция, работающая на уровне изоляции `serializable`, в режиме «только чтение» может быть задержана, прежде чем будет разрешено её выполнение. Однако когда она начинает выполняться, для обеспечения сериализуемости не требуется никаких дополнительных усилий, так что коду сериализации ни при каких условиях не придётся прерывать её из-за параллельных изменений, поэтому это вполне подходит для длительных транзакций в режиме «только чтение».

Этот параметр определяет, будет ли каждая новая транзакция по умолчанию откладываемой. В настоящее время его действие не распространяется на транзакции, для которых устанавливается режим «чтение/запись» или уровень изоляции ниже `serializable`. Значение по умолчанию — `off` (выкл.).

За дополнительной информацией обратитесь к SET TRANSACTION.

`transaction_isolation` (enum)

Данный параметр отражает уровень изоляции текущей транзакции. В начале каждой транзакции ему присваивается текущее значение default_transaction_isolation. Последующая попытка изменить значение этого параметра равнозначна команде SET TRANSACTION.

`transaction_read_only` (boolean)

Данный параметр отражает характеристику «только чтение» для текущей транзакции. В начале каждой транзакции ему присваивается текущее значение default_transaction_read_only. Последующая попытка изменить значение этого параметра равнозначна команде SET TRANSACTION.

`transaction_deferrable` (boolean)

Данный параметр отражает характеристику «откладываемости» для текущей транзакции. В начале каждой транзакции ему присваивается текущее значение default_transaction_deferrable. Последующая попытка изменить значение этого параметра равнозначна команде SET TRANSACTION.

`session_replication_role` (enum)

Управляет срабатыванием правил и триггеров, связанных с репликацией, в текущем сеансе. Она может принимать следующие значения: `origin` (значение по умолчанию), `replica` и `local`. Изменение этой переменной приводит к сбросу всех ранее кешированных планов запросов. Только суперпользователи и пользователи с соответствующим правом `SET` могут изменять её значение.

Предполагается, что системы логической репликации будут устанавливать для него значение `replica`, применяя реплицированные изменения. В результате триггеры и правила (с конфигурацией по умолчанию) не будут срабатывать в репликах. Подробнее об этом говорится в описании предложений `ENABLE TRIGGER` и `ENABLE RULE` команды `ALTER TABLE`.

Внутри Postgres Pro значения `origin` и `local` воспринимаются как равнозначные. Сторонние системы репликации могут различать их для своих внутренних целей, например, отмечать значением `local` сеанс, изменения в котором не должны реплицироваться.

Так как внешние ключи реализованы посредством триггеров, присваивание этому параметру значения `replica` влечёт отключение всех проверок внешних ключей, что может привести к нарушению согласованности данных при некорректном использовании.

`statement_timeout` (integer)

Задаёт максимальную длительность выполнения оператора, при превышении которой оператор прерывается. Если `log_min_error_statement` имеет значение `ERROR` или ниже, оператор, прерванный по тайм-ауту, будет также записан в журнал. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. При значении, равном нулю (по умолчанию), этот контроль длительности отключается.

Длительность выполнения отсчитывается с момента получения команды сервером до завершения её выполнения. Если в одном сообщении простого запроса поступает несколько SQL-операторов, ограничение длительности действует на каждый независимо. (До 13 версии Postgres Pro обычно считалось, что этот тайм-аут должен относиться ко всей строке запроса.) В расширенном протоколе запросов таймер запускается с момента получения любого связанного с запросом сообщения (Parse, Bind, Execute, Describe), а останавливается на стадии обработки сообщений Execute или Sync.

Устанавливать значение `statement_timeout` в `postgresql.conf` не рекомендуется, так как это повлияет на все сеансы.

`transaction_timeout` (integer)

Завершать любые сеансы, длящиеся дольше заданного времени в транзакции. Ограничение применяется как к явным транзакциям (начинающимся с `BEGIN`), так и к неявно запущенным транзакциям, состоящим из одного оператора. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. При значении, равном нулю (по умолчанию), этот контроль длительности отключается.

Если `transaction_timeout` меньше или равен `idle_in_transaction_session_timeout` или `statement_timeout`, игнорируется более длительный тайм-аут.

Не рекомендуется устанавливать значение `transaction_timeout` в `postgresql.conf`, так как это повлияет на все сеансы.

Примечание

Этот тайм-аут не влияет на подготовленные транзакции.

`lock_timeout` (`integer`)

Задаёт максимальную длительность ожидания любым оператором получения блокировки таблицы, индекса, строки или другого объекта базы данных. Если ожидание не закончилось за указанное время, оператор прерывается. Это ограничение действует на каждую попытку получения блокировки по отдельности и применяется как к явным запросам блокировки (например, `LOCK TABLE` или `SELECT FOR UPDATE` без `NOWAIT`), так и к неявным. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. При значении, равном нулю (по умолчанию), этот контроль длительности отключается.

В отличие от `statement_timeout`, этот тайм-аут может произойти только при ожидании блокировки. Заметьте, что при ненулевом `statement_timeout` бессмысленно задавать в `lock_timeout` такое же или большее значение, так как тайм-аут оператора всегда будет происходить раньше. Если `log_min_error_statement` имеет значение `ERROR` или ниже, оператор, прерванный по тайм-ауту, будет записан в журнал.

Устанавливать значение `lock_timeout` в `postgresql.conf` не рекомендуется, так как это повлияет на все сеансы.

`idle_in_transaction_session_timeout` (`integer`)

Завершать любые сеансы, простаивающие (то есть ожидающие запросов от клиентов) дольше заданного времени в открытой транзакции. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. При значении, равном нулю (по умолчанию), этот контроль длительности отключается.

Этот параметр полезен для предотвращения ситуаций, когда простаивающие сеансы удерживают блокировки чрезмерно долго. И даже если сеанс не держит мешающие другим блокировки, открытые транзакции препятствуют очистке недавно удалённых кортежей, которые могут быть видимыми только для данной транзакции. Поэтому длительный простой транзакций может приводить и к раздуванию таблиц. За подробностями обратитесь к Разделу 23.1.

`idle_session_timeout` (`integer`)

Завершать любые сеансы, простаивающие (то есть ожидающие запросов от клиентов) дольше заданного времени вне открытой транзакции. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. При значении, равном нулю (по умолчанию), этот контроль длительности отключается.

В отличие от ситуации с открытой транзакцией, простаивающий вне транзакции сеанс не оказывает значительного влияния на работу сервера, поэтому на практике чаще требуется включать тайм-аут `idle_in_transaction_session_timeout`.

Для подключений, устанавливаемых через промежуточный слой, например через пул соединений, включать данный тайм-аут следует с осторожностью, так как этот слой может не реагировать корректно на неожиданное закрытие соединений. Может иметь смысл установить этот тайм-аут только для интерактивных сеансов, возможно, распространяя его применение только на определённых пользователей.

`vacuum_freeze_table_age` (`integer`)

Задаёт максимальный возраст для поля `pg_class.relfrozenxid` таблицы, при достижении которого `VACUUM` будет производить агрессивное сканирование. Агрессивное сканирование отличается от обычного сканирования `VACUUM` тем, что затрагивает все страницы, которые могут содержать незамороженные XID или MXID, а не только те, что могут содержать мёртвые кортежи. Значение по умолчанию — 150 миллионов транзакций. Хотя пользователи могут задать любое значение от нуля до двух миллиардов, в `VACUUM` введён внутренний предел для действующего значения, равный 95% от autovacuum_freeze_max_age, чтобы периодически запускаемая вручную команда `VACUUM` имела шансы выполниться, прежде чем для таблицы будет запущена автоочистка для предотвращения зацикливания. За дополнительными сведениями обратитесь к Подразделу 23.1.5.

`vacuum_freeze_min_age` (`integer`)

Задаёт возраст для отсечки (в транзакциях), при достижении которого команда `VACUUM` должна вызывать замораживание страниц с более ранними XID. Значение по умолчанию — 50 миллионов транзакций. Хотя пользователи могут задать любое значение от нуля до одного миллиарда, в `VACUUM` введён внутренний предел для действующего значения, равный половине autovacuum_freeze_max_age, чтобы принудительная автоочистка выполнялась не слишком часто. За дополнительными сведениями обратитесь к Подразделу 23.1.5.

`vacuum_failsafe_age` (`integer`)

Задаёт максимальный возраст (в транзакциях) для поля `pg_class.relfrozenxid` некоторой таблицы, при достижении которого механизм `VACUUM` принимает чрезвычайные меры для защиты от зацикливания счётчика транзакций, приводящего к глобальному сбою. Эта стратегия защиты `VACUUM` применяется в экстренном случае. Обычно защитный режим включается, когда автоочистка для предотвращения зацикливания счётчика транзакций уже выполнялась в течение некоторого времени. Однако данная стратегия может также применяться во время любой операции `VACUUM`.

При срабатывании защитного режима любая действующая задержка на основе стоимости больше не применяется, дальнейшие несущественные задачи обслуживания (такие как очистка индекса) пропускаются, а любая используемая *Стратегия доступа к буферу* отключается, в результате чего `VACUUM` сможет свободно использовать все *общие буферы*.

По умолчанию — 1.6 миллиарда транзакций. Хотя пользователи могут установить любое значение из интервала от нуля до 2.1 миллиарда, `VACUUM` автоматически увеличит действующее значение до 105% от autovacuum_freeze_max_age, если заданное значение меньше.

`vacuum_multixact_freeze_table_age` (`integer`)

Задаёт максимальный возраст для поля `pg_class.relminmxid` таблицы, при достижении которого команда `VACUUM` будет выполнять агрессивное сканирование. Агрессивное сканирование отличается от обычного сканирования `VACUUM` тем, что затрагивает все страницы, которые могут содержать незамороженные XID или MXID, а не только те, что могут содержать мёртвые кортежи. Значение по умолчанию — 150 миллионов мультитранзакций. Хотя пользователи могут задать любое значение от нуля до двух миллиардов, в `VACUUM` введён внутренний предел для действующего значения, равный 95% от autovacuum_multixact_freeze_max_age, чтобы периодически запускаемая вручную команда `VACUUM` имела шансы выполниться, прежде чем для таблицы будет запущена автоочистка для предотвращения зацикливания. За дополнительными сведениями обратитесь к Подразделу 23.1.5.1.

`vacuum_multixact_freeze_min_age` (`integer`)

Задаёт возраст для отсечки (в мультитранзакциях), при достижении которого команда `VACUUM` должна вызывать замораживание страниц с более ранними идентификаторами мультитранзакций. Значение по умолчанию — 5 миллионов мультитранзакций. Хотя пользователи могут задать любое значение от нуля до одного миллиарда, в `VACUUM` введён внутренний предел для действующего значения, равный половине autovacuum_multixact_freeze_max_age, чтобы принудительная автоочистка не выполнялась слишком часто. За дополнительными сведениями обратитесь к Подразделу 23.1.5.1.

`vacuum_multixact_failsafe_age` (`integer`)

Задаёт максимальный возраст (в мультитранзакциях) для поля `pg_class.relminmxid` некоторой таблицы, при достижении которого механизм `VACUUM` принимает чрезвычайные меры для защиты от зацикливания счётчика мультитранзакций, приводящего к глобальному сбою. Эта стратегия защиты `VACUUM` применяется в экстренном случае. Обычно защитный режим включается, когда автоочистка для предотвращения зацикливания счётчика транзакций уже выполнялась в течение некоторого времени. Однако данная стратегия может также применяться во время любой операции `VACUUM`.

При срабатывании защитного режима любая действующая задержка на основе стоимости больше не применяется, а дальнейшие несущественные задачи обслуживания (такие как очистка индекса) пропускаются.

По умолчанию — 1.6 миллиарда мультитранзакций. Хотя пользователи могут установить любое значение из интервала от нуля до 2.1 миллиарда, `VACUUM` автоматически увеличит действующее значение до 105% от autovacuum_multixact_freeze_max_age, если заданное значение меньше.

`vacuum_update_datfrozenxid_only_when_needed` (`boolean`)

Поле `datfrozenxid` обычно обновляется для таблиц во время или после выполнения команды `VACUUM` при последовательном сканировании таблицы `pg_class`. Процесс обновления длится долго, если очищается много таблиц. Если для переменной `vacuum_update_datfrozenxid_only_when_needed` установлено значение `on`, поле `datfrozenxid` для таблицы обновляется при последовательном сканировании только в том случае, если она действительно обновлялась. В результате может значительно повыситься производительность при очистке большого количества таблиц. Значение по умолчанию — `off`.

`bytea_output` (`enum`)

Задаёт выходной формат для значения типа `bytea`. Это может быть формат `hex` (по умолчанию) или `escape` (традиционный формат Postgres Pro). За дополнительными сведениями обратитесь к Разделу 8.4. Входные значения `bytea` воспринимаются в обоих форматах, независимо от данного параметра.

`xmlbinary` (`enum`)

Задаёт способ кодирования двоичных данных в XML. Это кодирование применяется, например, когда значения `bytea` преобразуются в XML функциями `xmlelement` или `xmlforest`. Допустимые варианты, определённые в стандарте XML-схем: `base64` и `hex`. Значение по умолчанию — `base64`. Чтобы узнать больше о функциях для работы с XML, обратитесь к Разделу 9.15.

Конечный выбор в основном дело вкуса, ограничения могут накладываться только клиентскими приложениями. Оба метода поддерживают все возможные значения, хотя результат кодирования в base64 немного компактнее шестнадцатеричного вида (hex).

`xmloption` (`enum`)

Задаёт подразумеваемый по умолчанию тип преобразования между XML и символьными строками (`DOCUMENT` или `CONTENT`). За описанием этого преобразования обратитесь к Разделу 8.13. Значение по умолчанию — `CONTENT` (кроме него допускается значение `DOCUMENT`).

Согласно стандарту SQL этот параметр должен задаваться командой

```
SET XML OPTION { DOCUMENT | CONTENT };
```

Этот синтаксис тоже поддерживается в Postgres Pro.

`gin_pending_list_limit` (integer)

Задаёт максимальный размер очереди записей GIN, которая используется, когда включён режим `fastupdate`. Если размер очереди превышает заданный предел, записи из неё массово переносятся в основную структуру данных индекса GIN, и очередь очищается. Если это значение задаётся без единиц измерения, оно считается заданным в килобайтах. Размер по умолчанию — четыре мегабайта (`4MB`). Этот предел можно переопределить для отдельных индексов GIN, изменив их параметры хранения. За дополнительными сведениями обратитесь к Подразделу 62.4.4.1 и Подразделу 62.4.5.

`createrole_self_grant` (string)

Если пользователь, у которого есть право `CREATEROLE`, но нет права `SUPERUSER`, создаёт роль, и для этого параметра задано непустое значение, созданная роль будет назначена создающему её пользователю с указанными параметрами. Значение должно быть `set`, `inherit` или оба эти значения, разделённые запятыми. Значение по умолчанию — пустая строка, которая отключает эту функциональность.

Этот параметр позволяет пользователю с правом `CREATEROLE`, который не является суперпользователем, автоматически наследовать или получать возможность назначать права командой `SET ROLE` любым создаваемым пользователям. Поскольку пользователю с правом `CREATEROLE` всегда неявно предоставляется право `ADMIN OPTION` для создаваемых ролей, он всегда может выполнить `GRANT`, что приведёт к тому же результату, что и задание этого параметра. Однако автоматическое предоставление прав может быть удобнее. Суперпользователь автоматически наследует права всех ролей и всегда может выполнять `SET ROLE` для любых ролей. Этот параметр позволяет задавать аналогичное поведение пользователям с правом `CREATEROLE` в отношении создаваемых пользователей.

`event_triggers` (boolean)

Разрешать временное отключение событийных триггеров для устранения неполадок и корректировки неисправных событийных триггеров. При значении `false` все событийные триггеры будут отключены. Установка значения `true` позволяет срабатывать всем событийным триггерам, это значение по умолчанию. Изменять этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`restrict_nonsystem_relation_kind` (string)

Определять виды отношений, которым запрещается доступ к несистемным отношениям. Значение — список видов отношений, разделённый запятыми. В настоящее время поддерживаются отношения видов `view` и `foreign-table`.

18.11.2. Языковая среда и форматы

`DateStyle` (string)

Задаёт формат вывода значений даты и времени, а также правила интерпретации неоднозначных значений даты. По историческим причинам эта переменная содержит два независимых компонента: указание выходного формата (`ISO`, `Postgres`, `SQL` и `German`) и указание порядка год(Y)/месяц(M)/день(D) для вводимых и выводимых значений (`DMY`, `MDY` или `YMD`). Эти два компонента могут задаваться по отдельности или вместе. Ключевые слова `Euro` и `European` являются синонимами `DMY`, а ключевые слова `US`, `NonEuro` и `NonEuropean` — синонимы `MDY`. За дополнительными сведениями обратитесь к Разделу 8.5. Встроенное значение по умолчанию — `ISO, MDY`, но initdb при инициализации записывает в файл конфигурации значение, соответствующее выбранной локали `lc_time`.

`IntervalStyle` (enum)

Задаёт формат вывода для значений-интервалов. В формате `sql_standard` интервал выводится в виде, установленном стандартом SQL. В формате `postgres` (выбранном по умолчанию) интервал выводится в виде, применявшемся в PostgreSQL до версии 8.4, когда параметр DateStyle имел значение `ISO`. В формате `postgres_verbose` интервал выводится в виде, применявшемся в PostgreSQL до версии 8.4, когда параметр `DateStyle` имел значение не `ISO`. В формате `iso_8601` выводимая строка будет соответствовать «формату с кодами», определённому в разделе 4.4.3.2 стандарта ISO 8601.

На интерпретацию неоднозначных вводимых данных также влияет параметр `IntervalStyle`. За дополнительными сведениями обратитесь к Подразделу 8.5.4.

`TimeZone` (`string`)

Задаёт часовой пояс для вывода и ввода значений времени. Встроенное значение по умолчанию — `GMT`, но обычно оно переопределяется в `postgresql.conf`; initdb устанавливает в нём значение, соответствующее системному окружению. За дополнительными сведениями обратитесь к Подразделу 8.5.3.

`timezone_abbreviations` (`string`)

Задаёт набор сокращений часовых поясов, которые будут приниматься сервером во вводимых значениях даты и времени. Значение по умолчанию — `'Default'`, которое представляет набор основных сокращений, принятых в мире; допускаются также значения `'Australia'` и `'India'`, кроме них для конкретной инсталляции можно определить и другие наборы. За дополнительными сведениями обратитесь к Разделу B.4.

`extra_float_digits` (`integer`)

Этот параметр корректирует число цифр в текстовом представлении чисел с плавающей точкой, включая значения `float4`, `float8` и геометрических типов.

Если его значение равно 1 (по умолчанию) или больше, числа с плавающей точкой выводятся в кратчайшем точном виде; см. Подраздел 8.1.3. Выводимое фактически количество цифр зависит от выводимого числа, а не от значения данного параметра. Для значений `float8` может потребоваться максимум 17 цифр, а для значений `float4` — максимум 9. Данное представление является и быстрым, и точным, так как оно позволяет при правильном прочтении в точности восстановить двоичное число с плавающей точкой. Ради исторической совместимости этот параметр принимает значения до 3 включительно.

Если его значение равно нулю или отрицательно, выводимое число округляется до заданной десятичной точности. Точность в данном случае определяется обычным количеством цифр для типа (`FLT_DIG` или `DBL_DIG`), уменьшенным на значение этого параметра. (Например, со значением -1 числа типа `float4` будут выводиться округлёнными до 5 значащих цифр, а числа типа `float8` — до 14). Преобразование в такой вид выполняется медленнее и не сохраняет все биты двоичного числа с плавающей точкой, но он может быть более удобным для человека.

Примечание

Поведение этого параметра, а также его значение по умолчанию, изменилось в Postgres Pro 12; подробнее это описывается в Подразделе 8.1.3.

`client_encoding` (`string`)

Задаёт кодировку (набор символов) на стороне клиента. По умолчанию выбирается кодировка базы данных. Наборы символов, которые поддерживает сервер Postgres Pro, перечислены в Подразделе 22.3.1.

`lc_messages` (`string`)

Устанавливает язык выводимых сообщений. Набор допустимых значений зависит от системы; за дополнительными сведениями обратитесь к Разделу 22.1. Если эта переменная определена как пустая строка (по умолчанию), то действующее значение получается из среды выполнения сервера, в зависимости от системы.

В некоторых системах такая категория локали отсутствует, так что даже если задать значение этой переменной, действовать оно не будет. Также может оказаться, что переведённые сообщения для запрошенного языка отсутствуют. В этих случаях вы по-прежнему будете получать сообщения на английском языке.

Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

`lc_monetary` (`string`)

Устанавливает локаль для форматирования денежных сумм, например с использованием функций семейства `to_char`. Набор допустимых значений зависит от системы; за дополнительными сведениями обратитесь к Разделу 22.1. Если эта переменная определена как пустая строка (по умолчанию), то действующее значение получается из среды выполнения сервера, в зависимости от системы.

`lc_numeric` (`string`)

Устанавливает локаль для форматирования чисел, например с использованием функций семейства `to_char`. Набор допустимых значений зависит от системы; за дополнительными сведениями обратитесь к Разделу 22.1. Если эта переменная определена как пустая строка (по умолчанию), то действующее значение получается из среды выполнения сервера, в зависимости от системы.

`lc_time` (`string`)

Устанавливает локаль для форматирования даты и времени, например с использованием функций семейства `to_char`. Набор допустимых значений зависит от системы; за дополнительными сведениями обратитесь к Разделу 22.1. Если эта переменная определена как пустая строка (по умолчанию), то действующее значение получается из среды выполнения сервера, в зависимости от системы.

`icu_validation_level` (`enum`)

Указывает, какой уровень должен использоваться в сообщениях о проблемах с проверкой локали ICU. Допустимые значения: `DISABLED`, `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR` и `LOG`.

Если установлено значение `DISABLED`, сообщения о проблемах проверки не выдаются. В противном случае выдаются сообщения на выбранном уровне. По умолчанию используется `WARNING`.

`default_text_search_config` (`string`)

Выбирает конфигурацию текстового поиска для тех функций текстового поиска, которым не передаётся аргумент, явно указывающий конфигурацию. За дополнительной информацией обратитесь к Главе 12. Встроенное значение по умолчанию — `pg_catalog.simple`, но initdb при инициализации записывает в файл конфигурации сервера значение, соответствующее выбранной локали `lc_ctype`, если удастся найти такую конфигурацию текстового поиска.

18.11.3. Предзагрузка разделяемых библиотек

Для настройки предварительной загрузки разделяемых библиотек в память сервера, в целях подключения дополнительной функциональности или увеличения быстродействия, предназначены несколько параметров. Значения этих параметров задаются однотипно, например, со значением `'$libdir/mylib'` в память будет загружена `mylib.so` (или в некоторых ОС, `mylib.sl`) из стандартного каталога библиотек данной инсталляции сервера. Различаются эти параметры тем, когда они вступают в силу и какие права требуются для их изменения.

Таким же образом можно загрузить библиотеки на процедурных языках Postgres Pro, обычно в виде `'$libdir/plXXX'`, где `XXX` — имя языка: `pgsql`, `perl`, `tcl` или `python`.

Этим способом можно загрузить только разделяемые библиотеки, предназначенные специально для использования с Postgres Pro. Postgres Pro при загрузке библиотеки проверяет наличие «отличительного блока» для гарантии совместимости. Поэтому загрузить библиотеки не для Postgres Pro таким образом нельзя. Для этого вы можете воспользоваться средствами операционной системы, например, переменной окружения `LD_PRELOAD`.

В общем случае, чтобы узнать, какой способ рекомендуется для загрузки модуля, следует обратиться к документации этого модуля.

`local_preload_libraries` (`string`)

В этом параметре задаются одна или несколько разделяемых библиотек, которые будут загружаться при установлении соединения. Он содержит разделённый запятыми список имён библиотек, и каждое имя в нём должно восприниматься командой `LOAD`. Пробельные символы между именами игнорируются; если в имя нужно включить пробелы или запятые, заключите его в двойные кавычки. Заданное значение параметра действует только в начале соединения, так что последующие изменения ни на что не влияют. Если указанная в нём библиотека не найдена, установить подключение не удастся.

Этот параметр разрешено устанавливать всем пользователям. Поэтому библиотеки, которые так можно загрузить, ограничиваются теми, что находятся в подкаталоге `plugins` стандартного каталога библиотек установленного сервера. (Ответственность за то, чтобы в этом подкаталоге находились только «безопасные» библиотеки, лежит на администраторе.) В `local_preload_libraries` этот каталог можно задать явно (например, так: `$libdir/plugins/mylib`), либо просто указать имя библиотеки — `mylib` (оно будет воспринято как `$libdir/plugins/mylib`).

Данный механизм предназначен для того, чтобы непривилегированные пользователи могли загружать отладочные или профилирующие библиотеки в избранных сеансах, обходясь без явной команды `LOAD`. Для такого применения этот параметр обычно устанавливается в переменной окружения `PGOPTIONS` на клиенте или с помощью команды `ALTER ROLE SET`.

Обычно этот параметр не следует использовать, если только модуль не предназначен специально для такой загрузки обычными пользователями. Предпочтительная альтернатива ему — session_preload_libraries.

`session_preload_libraries` (`string`)

В этом параметре задаются одна или несколько разделяемых библиотек, которые будут загружаться при установлении соединения. Он содержит разделённый запятыми список имён библиотек, и каждое имя в нём должно восприниматься командой `LOAD`. Пробельные символы между именами игнорируются; если в имя нужно включить пробелы или запятые, заключите его в двойные кавычки. Заданное значение параметра действует только в начале соединения, так что последующие изменения ни на что не влияют. Если указанная в нём библиотека не найдена, установить подключение не удастся. Изменить его могут только суперпользователи и пользователи с соответствующим правом `SET`.

Данный параметр предназначен для загрузки отладочных или профилирующих библиотек в избранных сеансах, без явного выполнения команды `LOAD`. Например, можно загрузить модуль auto_explain во всех сеансах пользователя с заданным именем, установив этот параметр командой `ALTER ROLE SET`. Кроме того, этот параметр можно изменить без перезапуска сервера (хотя изменения вступают в силу только при запуске нового сеанса), так что таким образом проще подгружать новые модули, даже если это нужно сделать для всех сеансов.

В отличие от shared_preload_libraries, этот вариант загрузки библиотеки не даёт большого выигрыша в скорости по сравнению с вариантом загрузки при первом использовании. Однако он оказывается выигрышным, когда используется пул соединений.

`shared_preload_libraries` (`string`)

В этом параметре задаются одна или несколько разделяемых библиотек, которые будут загружаться при запуске сервера. Он содержит разделённый запятыми список имён библиотек, и каждое имя в нём должно восприниматься командой `LOAD`. Пробельные символы между именами игнорируются; если в имя нужно включить пробелы или запятые, заключите его в двойные кавычки. Этот параметр можно задать только при запуске сервера. Если указанная в нём библиотека не будет найдена, сервер не запустится.

Некоторые библиотеки при загрузке должны выполнять операции, которые могут иметь место только при запуске главного процесса, например, выделять разделяемую память, резервировать легковесные блокировки или запускать фоновые рабочие процессы. Такие библиотеки должны загружаться при запуске сервера посредством этого параметра. За подробностями обратитесь к документации библиотек.

Также можно предварительно загрузить и другие библиотеки. Предварительная загрузка позволяет избавиться от задержки, возникающей при первом использовании библиотеки. Однако при этом может несколько увеличиться время запуска каждого нового процесса, даже если он не будет использовать эту библиотеку. Поэтому применять этот параметр рекомендуется только для библиотек, которые будут использоваться большинством сеансов. Кроме того, при изменении этого параметра необходимо перезапускать сервер, так что этот вариант не подходит, например, для краткосрочных задач отладки. В таких случаях используйте вместо него session_preload_libraries.

Примечание

В системе Windows загрузка библиотек при запуске сервера не сокращает время запуска каждого нового серверного процесса; каждый процесс будет заново загружать все библиотеки. Однако параметр `shared_preload_libraries` всё же может быть полезен в Windows для загрузки библиотек, которые должны выполнять некоторые операции при запуске главного процесса.

`jit_provider` (`string`)

В этой переменной указывается библиотека провайдера JIT, которая будет использоваться (см. Подраздел 29.4.2). Значение по умолчанию — `llvmjit`. Этот параметр можно задать только при запуске сервера.

Если указывается несуществующая библиотека, JIT не будет работать, но это не считается ошибкой. Такое поведение позволяет устанавливать поддержку JIT отдельно от основного пакета Postgres Pro.

18.11.4. Другие параметры по умолчанию

`dynamic_library_path` (`string`)

Когда требуется открыть динамически загружаемый модуль и его имя, заданное в команде `CREATE FUNCTION` или `LOAD` не содержит имён каталогов (т. е. в этом имени нет косой черты), система будет искать запрошенный файл в данном пути.

Значением параметра `dynamic_library_path` должен быть список абсолютных путей, разделённых двоеточием (или точкой с запятой в Windows). Если элемент в этом списке начинается со специальной строки `$libdir`, вместо неё подставляется заданный при компиляции путь каталога библиотек Postgres Pro; в этот каталог устанавливаются модули, поставляемые в составе стандартного дистрибутива Postgres Pro. (Чтобы узнать имя этого каталога, можно выполнить `pg_config --pkglibdir`.) Например:

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

Или в среде Windows:

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

Значение по умолчанию этого параметра — `'$libdir'`. Если его значение — пустая строка, автоматический поиск по заданному пути отключается.

Суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр в процессе работы сервера, но такое изменение будет действовать только до завершения клиентского соединения, так что этот вариант следует оставить для целей разработки. Для других целей этот параметр рекомендуется устанавливать в файле конфигурации `postgresql.conf`.

`gin_fuzzy_search_limit` (`integer`)

Задаёт мягкий верхний лимит для размера набора, возвращаемого при сканировании индексов GIN. За дополнительными сведениями обратитесь к Подразделу 62.4.5.

18.12. Управление блокировками

`deadlock_timeout` (`integer`)

Время ожидания блокировки, по истечении которого будет выполняться проверка состояния взаимоблокировки. Эта проверка довольно дорогостоящая, поэтому сервер не выполняет её при всяком ожидании блокировки. Мы оптимистично полагаем, что взаимоблокировки редки в производственных приложениях, и поэтому просто ждём некоторое время, прежде чем пытаться выявить взаимоблокировку. При увеличении значения этого параметра сокращается время, уходящее на ненужные проверки взаимоблокировки, но замедляется реакция на реальные взаимоблокировки. Если это значение задаётся без единиц измерения, оно считается заданным в миллисекундах. Значение по умолчанию — одна секунда (`1s`), что близко к минимальному значению, которое стоит применять на практике. На сервере с большой нагрузкой имеет смысл увеличить его. В идеале это значение должно превышать типичное время транзакции, чтобы повысить шансы на то, что блокировка всё-таки будет освобождена, прежде чем ожидающая транзакция решит проверить состояние взаимоблокировки. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Когда включён параметр log_lock_waits, данный параметр также определяет, спустя какое время в журнал сервера будут записываться сообщения об ожидании блокировки. Если вы пытаетесь исследовать задержки, вызванные блокировками, имеет смысл уменьшить его по сравнению с обычным значением `deadlock_timeout`.

`max_locks_per_transaction` (`integer`)

В общей таблице блокировок может храниться `max_locks_per_transaction` объектов (например, таблиц) для каждого серверного процесса или подготовленной транзакции, таким образом, в любой момент времени может быть заблокировано не больше этого числа различных объектов. Этот параметр ограничивает среднее число блокировок объектов, используемых каждой транзакцией, отдельные транзакции могут заблокировать и больше объектов, если все они умещаются в таблице блокировок. Заметьте, что это *не* число строк, которое может быть заблокировано; их количество не ограничено. Значение по умолчанию, 64, как показала практика, вполне приемлемо, но может возникнуть потребность его увеличить, если запросы обращаются ко множеству различных таблиц в одной транзакции, как например, запрос к родительской таблице со многими потомками. Этот параметр можно задать только при запуске сервера.

Для ведомого сервера значение этого параметра должно быть больше или равно значению на ведущем. В противном случае на ведомом сервере не будут разрешены запросы.

`max_pred_locks_per_transaction` (`integer`)

В общей таблице предикатных блокировок хранится `max_pred_locks_per_transaction` объектов (например, таблиц) для каждого серверного процесса или подготовленной транзакции, таким образом, в один момент времени может быть заблокировано не больше этого числа различных объектов. Этот параметр ограничивает среднее число блокировок объектов, используемых каждой транзакцией; отдельные транзакции могут заблокировать и больше объектов, если все они умещаются в таблице блокировок. Заметьте, что это *не* число строк, которое может быть заблокировано; их количество не ограничено. Значение по умолчанию, 64, как показала практика, вполне приемлемо, но может возникнуть потребность его увеличить, если запросы обращаются ко множеству различных таблиц в одной сериализуемой транзакции, как например, запрос к родительской таблице со многими потомками. Этот параметр можно задать только при запуске сервера.

`max_pred_locks_per_relation` (`integer`)

Этот параметр определяет, для скольких страниц или кортежей одного отношения могут устанавливаться предикатные блокировки, прежде чем вместо них будет затребована одна блокировка для всего отношения. Значения, большие или равные нулю, задают абсолютный предел, а с отрицательным значением пределом будет значение max_pred_locks_per_transaction, делённое на модуль данного. По умолчанию действует значение -2, что даёт то же поведение, что наблюдалось в предыдущих версиях Postgres Pro. Этот параметр можно задать только в файле `postgresql.conf` или в командной строке при запуске сервера.

`max_pred_locks_per_page` (`integer`)

Этот параметр определяет, для скольких строк на одной странице могут устанавливаться предикатные блокировки, прежде чем вместо них будет затребована одна блокировка для всей страницы. Значение по умолчанию — 2. Этот параметр можно задать только в файле `postgresql.conf` или в командной строке при запуске сервера.

18.13. Совместимость с разными версиями и платформами

18.13.1. Предыдущие версии Postgres Pro

`array_nulls` (`boolean`)

Этот параметр определяет, будет ли при разборе вводимого массива распознаваться строка `NULL` без кавычек как элемент массива, равный NULL. Значение по умолчанию, `on`, позволяет задавать NULL в качестве элементов вводимого массива. Однако до версии 8.2 PostgreSQL не поддерживал ввод элементов NULL в массивах, а воспринимал `NULL` как обычный элемент массива со строковым значением «NULL». Для обратной совместимости с приложениями, зависящими от старого поведения, эту переменную можно отключить (присвоив ей `off`).

Заметьте, что массивы, содержащие NULL, можно создать, даже когда эта переменная имеет значение `off`.

`backslash_quote` (`enum`)

Этот параметр определяет, можно ли будет представить знак апострофа в строковой константе в виде `\'`. В стандарте SQL определён другой, предпочитаемый вариант передачи апострофа, дублированием (`''`), но Postgres Pro исторически также принимал вариант `\'`. Однако применение варианта `\'` сопряжено с угрозами безопасности, так как в некоторых клиентских кодировках существуют многобайтные символы, последний байт которых численно равен ASCII-коду `\`. Если код на стороне клиента выполнит экранирование некорректно, это может открыть возможности для SQL-инъекции. Предотвратить этот риск можно, запретив серверу принимать запросы, в которых апостроф экранируется обратной косой. Допустимые значения параметра `backslash_quote`: `on` (принимать `\'` всегда), `off` (не принимать никогда) и `safe_encoding` (принимать, только если клиентская кодировка не допускает присутствия ASCII-кода `\` в многобайтных символах). Значение по умолчанию — `safe_encoding`.

Заметьте, что в строковой константе, записанной согласно стандарту, знаки `\` обозначают просто `\`. Этот параметр влияет только на восприятие строк, не соответствующих стандарту, в том числе с синтаксисом спецпоследовательностей (`E'...'`).

`escape_string_warning` (`boolean`)

Когда этот параметр включён, сервер выдаёт предупреждение, если обратная косая черта (`\`) встречается в обычной строковой константе (с синтаксисом `'...'`) и параметр `standard_conforming_strings` отключён. Значение по умолчанию — `on` (вкл.).

Приложения, которые предпочитают использовать обратную косую в виде спецсимвола, должны перейти к применению синтаксиса спецстрок (`E'...'`), так как по умолчанию теперь в обычных строках обратная косая воспринимается как обычный символ, в соответствии со стандартом SQL. Включение данного параметра помогает найти код, нуждающийся в модификации.

`lo_compat_privileges` (boolean)

В PostgreSQL до версии 9.0 для больших объектов не назначались права доступа, и поэтому они были всегда доступны на чтение и запись для всех пользователей. Если установить для этого параметра значение `on`, существующие теперь проверки прав отключаются для совместимости с предыдущими версиями. Значение по умолчанию — `off`. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

Установка данного параметра не приводит к отключению всех проверок безопасности, связанных с большими объектами — затрагиваются только те проверки, которые изменились в PostgreSQL 9.0.

`quote_all_identifiers` (boolean)

Принудительно заключать в кавычки все идентификаторы, даже если это не ключевые слова (сегодня), при получении SQL из базы данных. Это касается вывода `EXPLAIN`, а также результатов функций типа `pg_get_viewdef`. См. также описание аргумента `--quote-all-identifiers` команд pg_dump и pg_dumpall.

`nul_byte_replacement_on_import` (string)

Заменять нулевые байты (`'\0'`) ASCII-символом с указанным десятичным кодом при загрузке данных с использованием команды `COPY FROM`. Такая замена может потребоваться при загрузке данных из другой СУБД, так как Postgres Pro не принимает нулевые байты в данных. ASCII-символ замены должен отличаться от символов `QUOTE` и `DELIMITER`, применяемых при выполнении `COPY FROM`; в противном случае результат может быть неожиданным. Значение по умолчанию — `'\0'`, то есть замена не производится.

`standard_conforming_strings` (boolean)

Этот параметр определяет, будет ли обратная косая черта в обычных строковых константах (`'...'`) восприниматься буквально, как того требует стандарт SQL. Начиная с версии PostgreSQL 9.1, он имеет значение `on` (в предыдущих версиях значение по умолчанию было `off`). Приложения могут выяснить, как обрабатываются строковые константы, проверив этот параметр. Наличие этого параметра может также быть признаком того, что поддерживается синтаксис спецпоследовательностей (`E'...'`). Этот синтаксис (Подраздел 4.1.2.2) следует использовать, если приложению нужно, чтобы обратная косая воспринималась как спецсимвол.

`synchronize_seqscans` (boolean)

Этот параметр включает синхронизацию обращений при последовательном сканировании больших таблиц, чтобы эти операции читали один блок примерно в одно и то же время, и, таким образом, нагрузка разделялась между ними. Когда он включён, сканирование может начаться в середине таблицы, чтобы синхронизироваться со сканированием, которое уже выполняется. По достижении конца таблицы сканирование «заворачивается» к началу и завершает обработку пропущенных строк. Это может привести к непредсказуемому изменению порядка строк, возвращаемых запросами, в которых отсутствует предложение `ORDER BY`. Когда этот параметр выключен (имеет значение `off`), реализуется поведение, принятое до версии 8.3, когда последовательное сканирование всегда начиналось с начала таблицы. Значение по умолчанию — `on`.

18.13.2. Совместимость с разными платформами и клиентами

`unicode_nul_character_replacement_in_jsonb` (integer)

Заменять `'\u0000'` заданным символом Unicode при вызове функции, обрабатывающей JSONB, поскольку функции `jsonb*` не принимают `'\u0000'`. Замещающий символ задаётся числовым значением от 0 до 65535. Значение этого параметра конфигурации может быть изменено в любой момент любым пользователем. Значение по умолчанию равно `0`, то есть замена не происходит.

`transform_null_equals` (`boolean`)

Когда этот параметр включён, проверки вида *`выражение`* `= NULL` (или `NULL =` *`выражение`*) воспринимаются как *`выражение`* `IS NULL`, то есть они истинны, если *`выражение`* даёт значение NULL, и ложны в противном случае. Согласно спецификации SQL, сравнение *`выражение`* `= NULL` должно всегда возвращать NULL (неизвестное значение). Поэтому по умолчанию этот параметр выключен (равен `off`).

Однако формы фильтров в Microsoft Access генерируют запросы, в которых проверка на значение NULL записывается как *`выражение`* `= NULL`, так что если вы используете этот интерфейс для обращения к базе данных, имеет смысл включить данный параметр. Так как проверки вида *`выражение`* `= NULL` всегда возвращают значение NULL (следуя правилам стандарта SQL), они не очень полезны и не должны встречаться в обычных приложениях, так что на практике от включения этого параметра не будет большого вреда. Однако начинающие пользователи часто путаются в семантике выражений со значениями NULL, поэтому по умолчанию этот параметр выключен.

Заметьте, что этот параметр влияет только на точную форму сравнения `= NULL`, но не на другие операторы сравнения или выражения, результат которых может быть равнозначен сравнению с применением оператора равенства (например, конструкцию `IN`). Поэтому данный параметр не может быть универсальной защитой от плохих приёмов программирования.

За сопутствующей информацией обратитесь к Разделу 9.2.

`allow_alter_system` (`boolean`)

Если для `allow_alter_system` установлено значение `off`, при выполнении команды `ALTER SYSTEM` возвращается ошибка. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера. Значение по умолчанию — `on`.

Обратите внимание, что этот параметр не следует рассматривать как средство защиты. Он отключает только команду `ALTER SYSTEM`, но не мешает суперпользователю изменять конфигурацию с помощью других команд SQL. Суперпользователю доступно множество способов выполнения команд оболочки на уровне операционной системы, поэтому он может изменять `postgresql.auto.conf` независимо от значения этого параметра.

Отключение этого параметра предназначено для сред, где конфигурация Postgres Pro управляется внешними инструментами. В таких условиях суперпользователь, действующий из лучших побуждений, может *по ошибке* изменить конфигурацию с помощью `ALTER SYSTEM` вместо использования внешнего инструмента. Это может привести к непредвиденным последствиям, например, внешний инструмент может позднее перезаписать изменение при обновлении конфигурации. Установка для этого параметра значения `off` может помочь избежать подобных ошибок.

Этот параметр управляет только использованием команды `ALTER SYSTEM`. Параметры в `postgresql.auto.conf` применяются, даже если для `allow_alter_system` установлено значение `off`.

18.14. Очистка памяти

Postgres Pro Standard предоставляет возможности очистки памяти, то есть перезаписи данных нулевыми байтами перед удалением. Для управления этими возможностями имеются соответствующие параметры конфигурации. За подробностями обратитесь к Разделу 30.1.

`wipe_file_on_delete` (`boolean`)

Перезаписывать файлы нулевыми байтами перед удалением.

По умолчанию: `on` (вкл.)

`wipe_heaptuple_on_delete` (`boolean`)

Перезаписывать версии строк нулевыми байтами перед удалением.

По умолчанию: `on` (вкл.)

`wipe_memctx_on_free` (`boolean`)

Очищать блоки в оперативной памяти перед освобождением. Этот параметр затрагивает память, выделенную в рамках определённого контекста.

По умолчанию: `on` (вкл.)

`wipe_mem_on_free` (`boolean`)

Очищать блоки в оперативной памяти перед освобождением. Этот параметр затрагивает память, не принадлежащую никакому контексту.

По умолчанию: `on` (вкл.)

`wipe_xlog_on_free` (`boolean`)

Перезаписывать сегменты WAL нулевыми байтами перед удалением или повторным использованием.

По умолчанию: `on` (вкл.)

18.15. Обработка ошибок

`exit_on_error` (`boolean`)

Если этот параметр включён, любая ошибка приведёт к прерыванию текущего сеанса. По умолчанию он отключён, так что сеанс будет прерываться только при критических ошибках.

`restart_after_crash` (`boolean`)

Когда этот параметр включён (это состояние по умолчанию), Postgres Pro будет автоматически перезагружаться после сбоя серверного процесса. Такой вариант позволяет обеспечить максимальную степень доступности базы данных. Однако в некоторых обстоятельствах, например, когда Postgres Pro управляется кластерным ПО, такую перезагрузку лучше отключить, чтобы кластерное ПО могло вмешаться и выполнить, возможно, более подходящие действия.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`data_sync_retry` (`boolean`)

При выключенном значении этого параметра (по умолчанию) Postgres Pro будет выдавать ошибку уровня PANIC в случае неудачи при попытке сохранить изменённые данные в файловой системе. В результате сервер баз данных остановится аварийно. Задать этот параметр можно только при запуске сервера.

В некоторых операционных системах состояние данных в кеше внутри ядра оказывается неопределённым при ошибке записи. В каких-то случаях эти данные могут быть просто утеряны, и повторять попытку записи небезопасно: вторая попытка может оказаться успешной, тогда как на деле данные не сохранены. В этих обстоятельствах единственный способ избежать потери данных — восстановить их из WAL после такого сбоя, но перед этим желательно выяснить причину проблемы и, возможно, заменить нерабочее оборудование.

Если включить этот параметр, Postgres Pro в случае сбоя при записи выдаст ошибку, но продолжит работу в расчёте повторить операцию сохранения данных при последующей контрольной точке. Включать его следует, только если достоверно известно, как поступает система с данными в буфере при ошибке записи.

`recovery_init_sync_method` (enum)

Со значением `fsync` (это значение по умолчанию), Postgres Pro будет рекурсивно открывать и синхронизировать все файлы в каталоге данных до начала восстановления после сбоя. Поиск файлов будет осуществляться по символическим ссылкам для каталога WAL и каждого настроенного табличного пространства (но не другим символическим ссылкам). Это делается для того, чтобы убедиться, что все файлы WAL и данных надёжно сохранены на диске перед воспроизведением изменений, и применяется при запуске кластера базы данных, который не был остановлен штатным образом (это касается и копий созданных программой pg_basebackup).

В Linux возможен вариант `syncfs`, когда от ОС требуется синхронизировать каждую из файловых систем, содержащих каталог данных, файлы WAL и табличные пространства (но не те файловые системы, которые подключены по символическим ссылкам). При этом не нужно открывать каждый отдельный файл, поэтому данный вариант может работать гораздо быстрее, чем `fsync`. С другой стороны, он может быть медленнее, если файловую систему совместно используют и другие приложения, изменяющие множество файлов, поскольку эти файлы также будут записываться на диск. Более того, Linux версий до 5.8 может не всегда сообщать Postgres Pro об ошибках ввода-вывода, произошедших при записи данных на диск, так что соответствующие сообщения можно увидеть только в журналах ядра.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`crash_info` (boolean)

Если установлено значение по умолчанию `on`, Postgres Pro записывает диагностическую информацию о сбое сервера в файл.

Этот параметр можно задать только при запуске сервера.

`crash_info_dump` (text)

Указывает разделённый запятыми список текстовых строк, содержащих источники данных для аварийных дампов. Возможны следующие значения строк:

- `queries` — тексты запросов.
- `memory_context` — контекст памяти Postgres Pro (за подробностями обратитесь к Разделу 45.3).
- `system` — информация об ОС.
- `module` — информация о модулях, загруженных в процесс `postgres`.
- `cpuinfo` — информация о процессоре: число ядер, набор инструкций, и т. д.
- `cpu_context`:
  - Регистры контекста процессора.
  - Общая информация об ошибке: номер сигнала, информация о коде сигнала, PID, PID родительского процесса, идентификатор пользователя и т. д.
- `virtual_memory` — информация о виртуальной памяти
- `instruction_pointer` — байты около RIP (return instruction pointer, указатель возврата инструкции, который указывает на адрес памяти, по которому можно судить о прогрессе выполнения программы в памяти).
- `stack` — размер стека (байты). Стек выгружается в отдельный файл из каталога, куда записывается диагностическая информация о сбоях сервера (по умолчанию `PGDATA/crash_info`). Этот файл называется так же, как и журнал сбоев, но имеет расширение `.data`. Адреса верхней и нижней границ стека записываются в журнал сбоев.

Значение по умолчанию — `system,module,queries,cpu_context,instruction_pointer`.

Этот параметр можно задать только при запуске сервера.

`crash_info_location` (`string`)

Указывает каталог, в котором должна храниться информация о сбое сервера. При значении `stderr` информация о сбое отправляется в каталог stderr. Если в этом параметре указана пустая строка `''` (по умолчанию), используется каталог `$PGDATA/crash_info`. Если нужно сохранять файлы в другом месте, заранее создайте целевой каталог и выдайте соответствующие права.

Этот параметр можно задать только при запуске сервера.

`crash_info_timer` (`boolean`)

Включает или отключает выгрузку состояния длительных запросов по таймеру, что позволяет профилировать такие запросы. Таймер запускается по интервалу, значение которого указывается в crash_info_timer_interval, и измеряет время обработки запроса. Когда время выполнения запроса превышает пороговое значение, указанное в crash_info_query_threshold, состояние запроса выгружается на диск. Значение по умолчанию — `off`. Чтобы таймер работал, для crash_info_dump должно быть установлено значение по крайней мере `queries`. Изменить этот параметр могут только суперпользователи командой `SET` в текущем сеансе или глобально в файле конфигурации.

Этот параметр можно задать только при запуске сервера в файле `postgresql.conf`. Чтобы перечитать значение параметра, перезапустите сервер или отправьте сигнал SIGHUP главному серверному процессу.

`crash_info_timer_interval` (`integer`)

Задаёт временной интервал для запуска таймера (в миллисекундах). Значение по умолчанию — `1000`. Изменить этот параметр могут только суперпользователи командой `SET` в текущем сеансе или глобально в файле конфигурации.

Этот параметр можно задать только при запуске сервера в файле `postgresql.conf`. Чтобы перечитать значение параметра, перезапустите сервер или отправьте сигнал SIGHUP главному серверному процессу.

`crash_info_query_threshold` (`integer`)

Задаёт пороговое значение времени обработки запроса (в миллисекундах). При достижении значения состояние запроса выгружается на диск. Значение по умолчанию — `3000`. Изменить этот параметр могут только суперпользователи командой `SET` в текущем сеансе или глобально в файле конфигурации.

Этот параметр можно задать только при запуске сервера в файле `postgresql.conf`. Чтобы перечитать значение параметра, перезапустите сервер или отправьте сигнал SIGHUP главному серверному процессу.

18.16. Предопределённые параметры

Следующие «параметры» доступны только для чтения. По этой причине они отсутствуют в примере файла `postgresql.conf`. Эти параметры отражают различные аспекты поведения Postgres Pro, которые могут быть полезны в определённых приложениях, например, клиентских средствах администрирования. Большинство из них задаётся при компиляции или при установке Postgres Pro.

`block_size` (`integer`)

Сообщает размер блока на диске. Он определяется значением `BLCKSZ` при сборке сервера. Значение по умолчанию — 8192 байта. Значение `block_size` влияет на некоторые другие переменные конфигурации (например, shared_buffers). Об этом говорится в Разделе 18.4.

`data_checksums` (`boolean`)

Сообщает, включён ли в этом кластере контроль целостности данных. За дополнительными сведениями обратитесь к Контрольные суммы данных.

`data_directory_mode` (`integer`)

В Unix-системах этот параметр показывает разрешения (определённые data_directory), которые были установлены для каталога данных на момент запуска сервера. (В Microsoft Windows этот параметр всегда показывает `0700`). За дополнительными сведениями обратитесь к Доступ группы.

`debug_assertions` (`boolean`)

Сообщает, был ли Postgres Pro собран с проверочными утверждениями. Это имеет место, когда при сборке Postgres Pro определяется макрос `USE_ASSERT_CHECKING` (например, при выполнении `configure` с флагом `--enable-cassert`). По умолчанию Postgres Pro собирается без проверочных утверждений.

`huge_pages_status` (`enum`)

Сообщает текущее состояние использования огромных страниц в экземпляре: `on`, `off` или `unknown` (если выводится с помощью `postgres -C`). Этот параметр позволяет определить, получилось ли использовать огромные страницы после указания `huge_pages=try`. За дополнительной информацией обратитесь к в huge_pages.

`integer_datetimes` (`boolean`)

Сообщает, был ли Postgres Pro собран с поддержкой даты и времени в 64-битных целых. Начиная с Postgres Pro версии 10, он всегда равен `on`.

`in_hot_standby` (`boolean`)

Сообщает, находится ли сервер в настоящий момент в режиме горячего резерва. Когда этот параметр имеет значение `on` (вкл.), все транзакции ограничивается режимом «только чтение». В рамках сеанса это может измениться только в том случае, если сервер повышается до ведущего. За подробностями обратитесь к Разделу 25.4.

`lc_collate` (`string`)

Сообщает локаль, по правилам которой выполняется сортировка текстовых данных. За дополнительными сведениями обратитесь к Разделу 22.1. Это значение определяется при создании базы данных.

`lc_ctype` (`string`)

Сообщает локаль, определяющую классификацию символов. За дополнительными сведениями обратитесь к Разделу 22.1. Это значение определяется при создании базы данных. Обычно оно не отличается от `lc_collate`, но для некоторых приложений оно может быть определено по-другому.

`max_function_args` (`integer`)

Сообщает верхний предел для числа аргументов функции. Он определяется константой `FUNC_MAX_ARGS` при сборке сервера. По умолчанию установлен предел в 100 аргументов.

`max_identifier_length` (`integer`)

Сообщает максимальную длину идентификатора. Она определяется числом на 1 меньше, чем `NAMEDATALEN`, при сборке сервера. По умолчанию константа `NAMEDATALEN` равна 64; следовательно `max_identifier_length` по умолчанию равна 63 байтам, но число символов в многобайтной кодировке будет меньше.

`max_index_keys` (`integer`)

Сообщает верхний предел для числа ключей индекса. Он определяется константой `INDEX_MAX_KEYS` при сборке сервера. По умолчанию установлен предел в 32 ключа.

`pgpro_build` (`string`)

Сообщает идентификатор состояния исходного кода, из которого скомпилирован Postgres Pro.

`pgpro_edition` (`string`)

Показывает редакцию Postgres Pro в виде текстовой строки, например `standard` или `enterprise`.

`pgpro_version` (`string`)

Показывает версию сервера Postgres Pro в виде текстовой строки.

`segment_size` (`integer`)

Сообщает, сколько блоков (страниц) можно сохранить в одном файловом сегменте. Это число определяется константой `RELSEG_SIZE` при сборке сервера. Максимальный размер сегмента в файлах равен произведению `segment_size` и `block_size`; по умолчанию это 1 гигабайт.

`server_encoding` (`string`)

Сообщает кодировку базы данных (набор символов). Она определяется при создании базы данных. Обычно клиентов должно интересовать только значение client_encoding.

`server_version` (`string`)

Сообщает номер версии сервера. Она определяется константой `PG_VERSION` при сборке сервера.

`server_version_num` (`integer`)

Сообщает номер версии сервера в виде целого числа. Она определяется константой `PG_VERSION_NUM` при сборке сервера.

`shared_memory_size` (`integer`)

Сообщает размер основной области разделяемой памяти, округлённый до ближайшего мегабайта.

`shared_memory_size_in_huge_pages` (`integer`)

Сообщает количество огромных страниц, необходимых для основной области разделяемой памяти при заданном значении huge_page_size. Если огромные страницы не поддерживаются, выдаётся значение `-1`.

Этот параметр поддерживается только в Linux, а на других платформах всегда имеет значение `-1`. За более подробной информацией об использовании огромных страниц в Linux обратитесь к Подразделу 17.4.5.

`ssl_library` (`string`)

Сообщает имя библиотеки SSL, с которой был собран данный сервер Postgres Pro (даже если SSL для данного экземпляра не настроен или не используется), например, `OpenSSL`, либо пустую строку, если сборка производилась без такой библиотеки.

`wal_block_size` (`integer`)

Сообщает размер блока WAL на диске. Он определяется константой `XLOG_BLCKSZ` при сборке сервера. Значение по умолчанию — 8192 байта.

`wal_segment_size` (`integer`)

Сообщает размер сегментов журнала предзаписи. Значение по умолчанию — 16 МБ. За дополнительными сведениями обратитесь к Разделу 27.5.

18.17. Внесистемные параметры

Поддержка внесистемных параметров была реализована, чтобы дополнительные модули (например, процедурные языки) могли добавлять собственные параметры, неизвестные серверу Postgres Pro. Это позволяет единообразно настраивать модули расширения.

Имена параметров расширений записываются следующим образом: имя расширения, точка и затем собственно имя параметра, подобно полным именам объектов в SQL. Например: `plpgsql.variable_conflict`.

Так как внесистемные параметры могут быть установлены в процессах, не загружающих соответствующий модуль расширения, Postgres Pro принимает значения для любых имён с двумя компонентами. Такие параметры воспринимаются как заготовки и не действуют до тех пор, пока не будет загружен определяющий их модуль. Когда модуль расширения загружается, он добавляет свои определения параметров и присваивает все заготовленные значения этим параметрам. Если при этом имена каких-то заготовок, начинающиеся с имени данного расширения, оказываются нераспознанными, выдаются предупреждения и эти заготовленные параметры удаляются.

18.18. Параметры для разработчиков

Следующие параметры предназначены для тестирования в процессе разработки, и их никогда не следует применять в производственной среде. (Однако в некоторых случаях они могут помочь восстановить сильно повреждённые базы данных.) По этим причинам они отсутствуют в примере файла `postgresql.conf`. Заметьте, что для использования многих из этих параметров требуются специальные флаги компиляции.

`allow_in_place_tablespaces` (`boolean`)

Позволяет создавать табличные пространства как каталоги в `pg_tblspc`, когда в качестве пути расположения в команде `CREATE TABLESPACE` задана пустая строка. Эта возможность предназначена для тестирования сценариев репликации, когда ведущий и ведомый серверы работают на одном компьютере. Существование каталогов в этом месте может оказаться неожиданным для средств резервного копирования, которые рассчитывают найти там только символические ссылки. Изменять этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`allow_system_table_mods` (`boolean`)

Разрешает модификации структуры системных таблиц, а также некоторые другие потенциально опасные операции с системными таблицами. Если данный параметр отключён, эти действия не разрешены даже суперпользователям. Непродуманное использование этого параметра чревато неисправимыми повреждениями данных и разрушением всей СУБД. Изменить этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`backtrace_functions` (`string`)

В этом параметре задаётся разделённый запятыми список имён функций C. В случае возникновения ошибки в функции, имя которой присутствует в этом списке, в журнал сервера помимо сообщения об ошибке будет выводиться трассировка стека. Это может быть полезно для отладки в определённых местах исходного кода.

Поддержка трассировки стека имеется не на всех платформах, а информационная ценность трассировки зависит от параметров компиляции.

Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

`debug_discard_caches` (`integer`)

Если установлено значение `1`, каждая запись кеша системного каталога становится недействительной при первой же возможности, независимо от того, произошло ли на самом деле что-то, что могло бы сделать её недействительной. В результате кеширование системных каталогов фактически отключается, поэтому сервер будет работать очень медленно. При увеличении значения этого параметра аннулирование кеша производится рекурсивно, что приводит к ещё большему замедлению и полезно только для тестирование собственно логики кеширования. Значение по умолчанию — `0` — включает обычное поведение кеширования каталога.

Этот параметр может быть очень полезен при выявлении трудновоспроизводимых ошибок, связанных с одновременными изменениями в каталоге, но редко бывает нужен в других случаях.

Этот параметр поддерживается, если при компиляции был определён макрос `DISCARD_CACHES_ENABLED` (он определяется автоматически при выполнении configure с ключом `--enable-cassert`). В обычных сборках его значение всегда будет `0`; при попытке изменить это значение возникнет ошибка.

`debug_io_direct` (`string`)

Обращается к ядру для минимизации кеширования данных отношений и файлов WAL, используя `O_DIRECT` (большинство Unix-подобных систем), `F_NOCACHE` (macOS) или `FILE_FLAG_NO_BUFFERING` (Windows).

Значение может представлять собой пустую строку (по умолчанию), чтобы отключить использование прямого ввода-вывода, или список операций, разделённых запятыми, в которых может использоваться прямой ввод-вывод. Варианты операций: `data` для основных файлов данных, `wal` для файлов WAL и `wal_init` для файлов WAL при первоначальном размещении.

Некоторые операционные и файловые системы не поддерживают прямой ввод-вывод, так что нестандартные значения параметра могут не приниматься при запуске или вызывать ошибки.

В настоящее время эта функциональность снижает производительность и предназначена только для тестирования.

`debug_parallel_query` (`enum`)

Позволяет распараллеливать запрос в целях тестирования, даже когда от этого не ожидается никакого выигрыша в скорости. Допустимые значения параметра `debug_parallel_query` — `off` (использовать параллельный режим только когда ожидается увеличение производительности), `on` (принудительно распараллеливать все запросы, для которых это безопасно) и `regress` (как `on`, но с дополнительными изменениями поведения, описанными ниже).

Говоря точнее, со значением `on` узел `Gather` добавляется в вершину любого плана запроса, для которого допускается распараллеливание, так что запрос выполняется внутри параллельного исполнителя. Даже когда параллельный исполнитель недоступен или не может быть использован, такие операции, как запуск подтранзакции, которые не должны выполняться в контексте параллельного запроса, не будут выполняться в этом режиме, если только планировщик не решит, что это приведёт к ошибке запроса. Если при включении этого параметра возникают ошибки или выдаются неожиданные результаты, вероятно, некоторые функции, задействованные в этом запросе, нужно пометить как `PARALLEL UNSAFE` (или, возможно, `PARALLEL RESTRICTED`).

Значение `regress` действует так же, как и значение `on`, с некоторыми дополнительными особенностями, предназначенными для облегчения автоматического регрессионного тестирования. Обычно сообщения от параллельных исполнителей включают строку контекста, отмечающую это, но значение `regress` подавляет эту строку, так что вывод не отличается от выполнения в не параллельном режиме. Кроме того, узлы `Gather`, добавляемые в планы с этим значением параметра, скрываются в выводе `EXPLAIN`, чтобы вывод соответствовал тому, что будет получен при отключении этого параметра (со значением `off`).

`ignore_system_indexes` (`boolean`)

Отключает использование индексов при чтении системных таблиц (при этом индексы всё же будут изменяться при записи в эти таблицы). Это полезно для восстановления работоспособности при повреждённых системных индексах. Этот параметр нельзя изменить после запуска сеанса.

`post_auth_delay` (`integer`)

Этот параметр задаёт задержку при запуске нового серверного процесса после выполнения процедуры аутентификации. Он предназначен для того, чтобы разработчики имели возможность подключить отладчик к серверному процессу. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. При нулевом значении (по умолчанию) задержка отсутствует. Этот параметр нельзя изменить после начала сеанса.

`pre_auth_delay` (`integer`)

Этот параметр задаёт задержку, добавляемую сразу после порождения нового серверного процесса, до выполнения процедуры аутентификации. Он предназначен для того, чтобы разработчики имели возможность подключить отладчик к серверному процессу при решении проблем с аутентификацией. Если это значение задаётся без единиц измерения, оно считается заданным в секундах. При нулевом значении (по умолчанию) задержка отсутствует. Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`trace_notify` (`boolean`)

Включает вывод очень подробной отладочной информации при выполнении команд `LISTEN` и `NOTIFY`. Чтобы эти сообщения передавались клиенту или в журнал сервера, параметр client_min_messages или log_min_messages, соответственно, должен иметь значение `DEBUG1` или ниже.

`trace_sort` (`boolean`)

Включает вывод информации об использовании ресурсов во время операций сортировки. Этот параметр доступен, только если при сборке Postgres Pro был определён макрос `TRACE_SORT`. (По умолчанию макрос `TRACE_SORT` определён.)

`trace_locks` (`boolean`)

Включает вывод подробных сведений о блокировках. В эти сведения входит вид операции блокировки, тип блокировки и уникальный идентификатор объекта, который блокируется или разблокируется. Кроме того, в их составе выводятся битовые маски для типов блокировок, уже полученных для данного объекта, и для типов блокировок, ожидающих его освобождения. В дополнение к этому выводится количество полученных и ожидающих блокировок для каждого типа блокировок, а также их общее количество. Ниже показан пример вывода в журнал:

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`trace_lwlocks` (`boolean`)

Включает вывод информации об использовании легковесных блокировок. Такие блокировки предназначены в основном для взаимоисключающего доступа к общим структурам данных в памяти.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`trace_userlocks` (`boolean`)

Включает вывод информации об использовании пользовательских блокировок. Она выводится в том же формате, что и с `trace_locks`, но по рекомендательным блокировкам.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`trace_lock_oidmin` (`integer`)

Если этот параметр установлен, при трассировке блокировок не будут отслеживаться таблицы с OID меньше заданного (это используется для исключения из трассировки системных таблиц).

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`trace_lock_table` (`integer`)

Безусловно трассировать блокировки для таблицы с заданным OID.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`debug_deadlocks` (`boolean`)

Включает вывод информации обо всех текущих блокировках при тайм-ауте взаимоблокировки.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `LOCK_DEBUG`.

`log_btree_build_stats` (`boolean`)

Включает вывод статистики использования системных ресурсов (памяти и процессора) при различных операциях с B-деревом.

Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `BTREE_BUILD_STATS`.

`wal_consistency_checking` (`string`)

Этот параметр предназначен для проверки ошибок в процедурах воспроизведения WAL. Когда он включён, в записи WAL добавляются полные образы страниц всех изменяемых буферов, Когда запись впоследствии воспроизводится, система сначала применяет эту запись, а затем проверяет, соответствуют ли буферы, изменённые записью, сохранённым образам. В определённых случаях (например, во вспомогательных битах) небольшие изменения допускаются и будут игнорироваться. Если выявляются неожиданные различия, это считается критической ошибкой и восстановление прерывается.

По умолчанию значение этого параметра — пустая строка, так что эта функциональность отключена. Его значением может быть `all` (будут проверяться все записи) или список имён менеджеров ресурсов через запятую (будут проверяться записи, выдаваемые этими менеджерами). В настоящее время поддерживаются менеджеры `heap`, `heap2`, `btree`, `hash`, `gin`, `gist`, `sequence`, `spgist`, `brin` и `generic`. В расширениях могут определяться и другие менеджеры ресурсов. Изменять этот параметр могут только суперпользователи и пользователи с соответствующим правом `SET`.

`wal_debug` (`boolean`)

Включает вывод отладочной информации, связанной с WAL. Этот параметр доступен, только если при компиляции Postgres Pro был определён макрос `WAL_DEBUG`.

`ignore_checksum_failure` (`boolean`)

Этот параметр действует, только если включён Контрольные суммы данных.

При обнаружении ошибок контрольных сумм при чтении Postgres Pro обычно сообщает об ошибке и прерывает текущую транзакцию. Если параметр `ignore_checksum_failure` включён, система игнорирует проблему (но всё же предупреждает о ней) и продолжает обработку. Это поведение может *привести к краху, распространению или сокрытию повреждения данных* *и другим серьёзными проблемам*. Однако включив его, вы можете обойти ошибку и получить неповреждённые данные, которые могут находиться в таблице, если цел заголовок блока. Если же повреждён заголовок, будет выдана ошибка, даже когда этот параметр включён. По умолчанию этот параметр отключён (имеет значение `off`). Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

`zero_damaged_pages` (`boolean`)

При выявлении повреждённого заголовка страницы Postgres Pro обычно сообщает об ошибке и прерывает текущую транзакцию. Если параметр `zero_damaged_pages` включён, вместо этого система выдаёт предупреждение, обнуляет повреждённую страницу в памяти и продолжает обработку. Это поведение *разрушает данные*, а именно все строки в повреждённой странице. Однако включив его, вы можете обойти ошибку и получить строки из неповреждённых страниц, которые могут находиться в таблице. Это бывает полезно для восстановления данных, испорченных в результате аппаратной или программной ошибки. Обычно включать его следует только тогда, когда не осталось никакой другой надежды на восстановление данных в повреждённых страницах таблицы. Обнулённые страницы не сохраняются на диск, поэтому прежде чем выключать этот параметр, рекомендуется пересоздать проблемные таблицы или индексы. По умолчанию этот параметр отключён (имеет значение `off`). Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

`ignore_invalid_pages` (`boolean`)

Когда этот параметр имеет значение `off` (по умолчанию), обнаружение в WAL записей, ссылающихся на некорректные страницы, в процессе восстановления приводит к остановке Postgres Pro с ошибкой критического уровня и, как следствие, к прерыванию восстановления. Когда параметр `ignore_invalid_pages` включён (`on`), система игнорирует подобные недействительные ссылки в записях WAL (но всё же выдаёт предупреждения) и продолжает восстановление. Это поведение может *привести к краху, потере данных, распространению или сокрытию повреждения данных и другим серьёзными проблемам*. Однако включив этот параметр, вы можете обойти критическую ошибку и закончить восстановление, чтобы сервер всё же запустился. Задать этот параметр можно только при запуске сервера. Действует он только при восстановлении или в режиме ведомого.

`jit_debugging_support` (`boolean`)

При наличии требуемой функциональности LLVM регистрировать генерируемые функции в GDB. Это позволяет упростить отладку. Значение по умолчанию — `off` (выкл.). Изменить этот параметр можно только при запуске сервера.

`jit_dump_bitcode` (`boolean`)

Записывать сгенерированный LLVM IR-код в файловую систему, в каталог data_directory. Это полезно только для работы с внутренним механизмом JIT. Значение по умолчанию — `off` (выкл.). Только суперпользователи и пользователи с соответствующим правом `SET` могут изменить этот параметр.

`jit_expressions` (`boolean`)

Определяет, будут ли JIT-компилироваться выражения, когда JIT-компиляция включена (см. Раздел 29.2). Значение по умолчанию — `on` (вкл.).

`jit_profiling_support` (`boolean`)

При наличии требуемой функциональности LLVM выдавать данные, необходимые для профилирования с помощью perf функций, которые генерирует JIT. Выходные файлы записываются в каталог `~/.debug/jit/`; удалять их при желании должен сам пользователь. Значение по умолчанию — `off` (выкл.). Задать этот параметр можно только при запуске сервера.

`jit_tuple_deforming` (`boolean`)

Определяет, будет ли JIT-компилироваться преобразование кортежей, когда JIT-компиляция включена (см. Раздел 29.2). Значение по умолчанию — `on` (вкл.).

`remove_temp_files_after_crash` (`boolean`)

Если этот параметр включён (состояние по умолчанию), Postgres Pro автоматически удаляет временные файлы после сбоя сервера. Если он выключен, файлы будут сохранены и могут использоваться, например, для отладки. Однако повторяющиеся сбои могут привести к накоплению бесполезных файлов. Этот параметр можно задать только в `postgresql.conf` или в командной строке при запуске сервера.

`send_abort_for_crash` (`boolean`)

По умолчанию после сбоя сервера управляющий процесс postmaster останавливает оставшиеся дочерние процессы, отправляя им сигналы SIGQUIT, что позволяет им завершить работу более или менее корректно. Если для этого параметра установлено значение `on`, вместо SIGQUIT отправляется SIGABRT. Обычно это приводит к созданию файла дампа памяти для каждого такого дочернего процесса. Это может быть удобно для исследования состояний других процессов после сбоя. Но такие файлы также могут занимать много места на диске в случае повторяющихся сбоев, поэтому не включайте параметр в системах без тщательного мониторинга. Имейте в виду, что автоматическое удаление дампов памяти не поддерживается. Этот параметр можно задать только в файле `postgresql.conf` или в командной строке при запуске сервера.

`send_abort_for_kill` (`boolean`)

По умолчанию после попытки остановить дочерний процесс, используя сигнал SIGQUIT, управляющий процесс postmaster ждёт пять секунд, а затем отправляет SIGKILL для немедленного завершения. Если для этого параметра установлено значение `on`, вместо SIGKILL отправляется SIGABRT. Обычно это приводит к созданию файла дампа памяти для каждого такого дочернего процесса. Это может быть удобно для исследования состояний «зависших» дочерних процессов. Но такие файлы также могут занимать много места на диске в случае повторяющихся сбоев, поэтому не включайте параметр в системах без тщательного мониторинга. Имейте в виду, что автоматическая очистка файлов дампа не поддерживается. Этот параметр можно задать только в файле `postgresql.conf` или в командной строке при запуске сервера.

`debug_logical_replication_streaming` (`enum`)

Допустимые значения: `buffered` и `immediate`. По умолчанию используется `buffered`. Этот параметр предназначен для тестирования логического декодирования и репликации больших транзакций. Параметр `debug_logical_replication_streaming` на публикующем сервере и на подписчике работает по-разному:

На стороне публикующего сервера `debug_logical_replication_streaming` позволяет выполнять потоковую передачу или сериализацию изменений сразу же при логическом декодировании. Если установлено значение `immediate`, каждое изменение передаётся в потоковом режиме, если подписка была создана с параметром `streaming`, в противном случае изменения сериализуются. Если установлено значение `buffered`, при декодировании изменения передаются в потоке или сериализуются при достижении значения `logical_decoding_work_mem`.

На стороне подписчика, если для параметра `streaming` задано значение `parallel`, можно использовать `debug_logical_replication_streaming`, чтобы указать ведущему процессу применения изменений отправлять изменения в очередь общей памяти или для сериализации всех изменений в файле. Если задано значение `buffered`, ведущий процесс отправляет изменения параллельным рабочим процессам применения изменений через очередь общей памяти. Если установлено значение `immediate`, ведущий процесс сериализует все изменения в файлы и уведомляет параллельные рабочие процессы применения изменений о необходимости их чтения и применения в конце транзакции.

18.19. Краткие аргументы

Для удобства с некоторыми параметрами сопоставлены однобуквенные аргументы командной строки. Все они описаны в Таблице 18.4. Некоторые из этих сопоставлений существуют по историческим причинам, так что наличие однобуквенного синонима не обязательно является признаком того, что этот аргумент часто используется.

Таблица 18.4. Ключ краткого аргумента

Краткий аргумент	Эквивалент
`-B` *x*	`shared_buffers =` *x*
`-d` *x*	`log_min_messages = DEBUG` *x*
`-e`	`datestyle = euro`
`-fb`, `-fh`, `-fi`, `-fm`, `-fn`, `-fo`, `-fs`, `-ft`	`enable_bitmapscan = off`, `enable_hashjoin = off`, `enable_indexscan = off`, `enable_mergejoin = off`, `enable_nestloop = off`, `enable_indexonlyscan = off`, `enable_seqscan = off`, `enable_tidscan = off`
`-F`	`fsync = off`
`-h` *x*	`listen_addresses =` *x*
`-i`	`listen_addresses = '*'`
`-k` *x*	`unix_socket_directories =` *x*
`-l`	`ssl = on`
`-N` *x*	`max_connections =` *x*
`-O`	`allow_system_table_mods = on`
`-p` *x*	`port =` *x*
`-P`	`ignore_system_indexes = on`
`-s`	`log_statement_stats = on`
`-S` *x*	`work_mem =` *x*
`-tpa`, `-tpl`, `-te`	`log_parser_stats = on`, `log_planner_stats = on`, `log_executor_stats = on`
`-W` *x*	`post_auth_delay =` *x*

Глава 19. Аутентификация клиентского приложения

При подключении к серверу базы данных, клиентское приложение указывает имя пользователя Postgres Pro, так же как и при обычном входе пользователя на компьютер с ОС Unix. При работе в среде SQL по имени пользователя определяется, какие у него есть права доступа к объектам базы данных (подробнее это описывается в Главе 20). Следовательно, важно указать на этом этапе, к каким базам пользователь имеет право подключиться.

> ### Примечание
>
> Как можно узнать из Главы 20, Postgres Pro управляет правами и привилегиями, используя так называемые «роли». В этой главе под *пользователем* мы подразумеваем «роль с привилегией `LOGIN`».

Аутентификация это процесс идентификации клиента сервером базы данных, а также определение того, может ли клиентское приложение (или пользователь запустивший приложение) подключиться с указанным именем пользователя.

Postgres Pro предлагает несколько различных методов аутентификации клиентов. Метод аутентификации конкретного клиентского соединения может основываться на адресе компьютера клиента, имени базы данных, имени пользователя.

Имена пользователей базы данных Postgres Pro не имеют прямой связи с пользователями операционной системы на которой запущен сервер. Если у всех пользователей базы данных заведена учётная запись в операционной системе сервера, то имеет смысл назначить им точно такие же имена для входа в Postgres Pro. Однако сервер, принимающий удалённые подключения, может иметь большое количество пользователей базы данных, у которых нет учётной записи в ОС. В таких случаях не требуется соответствие между именами пользователей базы данных и именами пользователей операционной системы.

19.1. Файл `pg_hba.conf`

Аутентификация клиентов управляется конфигурационным файлом, который традиционно называется `pg_hba.conf` и расположен в каталоге с данными кластера базы данных. (HBA расшифровывается как host-based authentication — аутентификации по имени узла.) Файл `pg_hba.conf`, со стандартным содержимым, создаётся командой initdb при инициализации каталога с данными. Однако его можно разместить в любом другом месте; см. конфигурационный параметр hba_file.

Файл `pg_hba.conf` прочитывается при запуске системы, а также в тот момент, когда основной сервер получает сигнал SIGHUP. Если вы редактируете файл во время работы системы, необходимо послать сигнал процессу postmaster (используя `pg_ctl reload`, вызвав SQL-функцию `pg_reload_conf()` или выполнив `kill -HUP`), чтобы он прочел обновлённый файл.

> ### Примечание
>
> Предыдущее утверждение не касается Microsoft Windows: там любые изменения в `pg_hba.conf` сразу применяются к последующим подключениям.

Системное представление `pg_hba_file_rules` может быть полезно для предварительной проверки изменений в файле `pg_hba.conf` или для диагностики проблем, когда перезагрузка этого файла не даёт желаемого эффекта. Строки в этом представлении, содержащие в поле `error` не NULL, указывают на проблемы в соответствующих строках файла.

Общий формат файла `pg_hba.conf` — набор записей, по одной на строку. Пустые строки игнорируются, как и любой текст комментария после знака `#`. Запись может быть продолжена на следующей строке, для этого нужно завершить строку обратной косой чертой. (Обратная косая черта является спецсимволом только в конце строки.) Запись состоит из нескольких полей, разделённых пробелами и/или табуляциями. Поля могут содержать пробелы, если содержимое этих полей заключено в кавычки. Если в кавычки берётся какое-либо ключевое слово в поле базы данных, пользователя или адресации (например, `all` или `replication`), то слово теряет своё особое значение и просто обозначает базу данных, пользователя или сервер с данным именем. Обратная косая черта обозначает перенос строки даже в тексте в кавычках или комментариях.

Каждая запись аутентификации обозначает тип соединения, диапазон IP-адресов клиента (если он соотносится с типом соединения), имя базы данных, имя пользователя и способ аутентификации, который будет использован для соединения в соответствии с этими параметрами. Первая запись с соответствующим типом соединения, адресом клиента, указанной базой данных и именем пользователя применяется для аутентификации. Процедур «fall-through» или «backup» не предусмотрено: если выбрана запись и аутентификация не прошла, последующие записи не рассматриваются. Если же ни одна из записей не подошла, в доступе будет отказано.

Каждая запись может быть директивой включения или записью аутентификации. Директивы включения указывают, что нужно обработать дополнительные файлы с записями. Записи будут вставлены вместо директив включения. Директивы включения содержат только два поля: директиву `include`, `include_if_exists` или `include_dir` и включаемый файл или каталог. Путь к файлу или каталогу может задаваться и как абсолютный, и как относительный и заключаться в двойные кавычки. Для формы `include_dir` будут включены все файлы, не начинающиеся с `.` и заканчивающиеся на `.conf`. Набор файлов во включаемом каталоге обрабатывается по порядку имён (определяемому правилами, принятыми в C, т. е. цифры идут перед буквами, а буквы в верхнем регистре — перед буквами в нижнем).

Записи могут иметь следующие форматы:

```
local         база  пользователь  метод-аутентификации [параметры-аутентификации]
host          база  пользователь  адрес     метод-аутентификации  [параметры-аутентификации]
hostssl       база  пользователь  адрес     метод-аутентификации  [параметры-аутентификации]
hostnossl     база  пользователь  адрес     метод-аутентификации  [параметры-аутентификации]
hostgssenc    база  пользователь  адрес     метод-аутентификации  [параметры-аутентификации]
hostnogssenc  база  пользователь  адрес     метод-аутентификации  [параметры-аутентификации]
host          база  пользователь  IP-адрес  IP-маска      метод-аутентификации  [параметры-аутентификации]
hostssl       база  пользователь  IP-адрес  IP-маска      метод-аутентификации  [параметры-аутентификации]
hostnossl     база  пользователь  IP-адрес  IP-маска      метод-аутентификации  [параметры-аутентификации]
hostgssenc    база  пользователь  IP-адрес  IP-маска      метод-аутентификации  [параметры-аутентификации]
hostnogssenc  база  пользователь  IP-адрес  IP-маска      метод-аутентификации  [параметры-аутентификации]
include             файл
include_if_exists   файл
include_dir         каталог
```

Значения полей описаны ниже:

`local`

Управляет подключениями через Unix-сокеты. Без подобной записи подключения через Unix-сокеты невозможны.

`host`

Управляет подключениями, устанавливаемыми по TCP/IP. Записи `host` соответствуют подключениям с SSL и без SSL, а также подключениям, защищённым механизмами GSSAPI и не защищённым GSSAPI.

> **Примечание**
>
> Удалённое соединение по TCP/IP невозможно, если сервер запущен без определения соответствующих значений для параметра конфигурации listen_addresses, поскольку по умолчанию система принимает подключения по TCP/IP только для локального адреса замыкания `localhost`.

`hostssl`

Управляет подключениями, устанавливаемыми по TCP/IP с применением шифрования SSL.

Чтобы использовать эту возможность, сервер должен быть собран с поддержкой SSL. Более того, механизм SSL должен быть включён параметром конфигурации ssl (подробнее об этом в Разделе 17.9). В противном случае запись `hostssl` не играет роли (не считая предупреждения о том, что ей не будут соответствовать никакие подключения).

`hostnossl`

Этот тип записей противоположен `hostssl`, ему соответствуют только подключения по TCP/IP без шифрования SSL.

`hostgssenc`

Управляет подключениями, устанавливаемыми по TCP/IP, для которых применяется шифрование GSSAPI.

Чтобы использовать эту возможность, сервер должен быть собран с поддержкой GSSAPI. В противном случае запись `hostgssenc` не играет роли (не считая предупреждения о том, что ей не будут соответствовать никакие подключения).

`hostnogssenc`

Этот тип записей противоположен `hostgssenc`; ему соответствуют только подключения по TCP/IP без шифрования GSSAPI.

`база`

Определяет, каким именам баз данных соответствует эта запись. Значение `all` определяет, что подходят все базы данных. Значение `sameuser` определяет, что данная запись соответствует, только если имя запрашиваемой базы данных совпадает с именем запрашиваемого пользователя. Значение `samerole` определяет, что запрашиваемый пользователь должен быть членом роли с таким же именем, как и у запрашиваемой базы данных. (`samegroup` — это устаревший, но допустимый вариант значения `samerole`.) Суперпользователи, несмотря на то, что они имеют полные права, считаются включёнными в `samerole`, только когда они явно входят в такую роль, непосредственно или косвенно. Значение `replication` показывает, что запись соответствует, когда запрашивается подключение для физической репликации, но не когда запрашивается подключение для логической. Имейте в виду, что для подключений физической репликации не указывается какая-то конкретная база данных, в то время как для подключений логической репликации должна указываться конкретная база. Любое другое значение воспринимается как имя определённой базы Postgres Pro или регулярное выражение. Несколько регулярных выражений и/или имён баз данных можно указать, разделяя их запятыми.

Если имя базы данных начинается с косой черты (`/`), оставшаяся часть имени обрабатывается как регулярное выражение. (Подробнее синтаксис регулярных выражений Postgres Pro описан в Подразделе 9.7.3.1.)

Также можно задать отдельный файл с именами баз данных и/или регулярными выражениями, написав имя файла после знака `@`.

пользователь

Указывает, какому имени (или именам) пользователя базы данных соответствует эта запись. Значение `all` указывает, что запись соответствует всем пользователям. Любое другое значение задаёт либо имя конкретного пользователя базы данных, либо регулярное выражение (если это значение начинается с `/`), либо имя группы (если это значение начинается с `+`). (Напомним, что в Postgres Pro нет никакой разницы между пользователем и группой; знак `+` означает «совпадение любых ролей, которые прямо или косвенно являются членами роли», тогда как имя без знака `+` является подходящим только для этой конкретной роли.) В связи с этим, суперпользователь рассматривается как член роли, только если он явно является членом этой роли, прямо или косвенно, а не только потому, что он является суперпользователем. Несколько регулярных выражений и/или имён пользователей можно указать, разделяя их запятыми.

Если имя пользователя начинается с косой черты (`/`), оставшаяся часть имени обрабатывается как регулярное выражение. (Подробнее синтаксис регулярных выражений Postgres Pro описан в Подразделе 9.7.3.1.)

Также можно задать отдельный файл с именами пользователей и/или регулярными выражениями, написав имя файла после знака `@`.

адрес

Указывает адрес (или адреса) клиентской машины, которым соответствует данная запись. Это поле может содержать или имя компьютера, или диапазон IP-адресов, или одно из нижеупомянутых ключевых слов.

Диапазон IP-адресов указывается в виде начального адреса диапазона, дополненного косой чертой (`/`) и длиной маски CIDR. Длина маски задаёт количество старших битов клиентского IP-адреса, которые должны совпадать с битами IP-адреса диапазона. Биты, находящиеся правее, в указанном IP-адресе должны быть нулевыми. Между IP-адресом, знаком `/` и длиной маски CIDR не должно быть пробельных символов.

Типичные примеры диапазонов адресов IPv4, указанных таким образом: `172.20.143.89/32` для одного компьютера, `172.20.143.0/24` для небольшой и `10.6.0.0/16` для крупной сети. Диапазон адресов IPv6 может выглядеть как `::1/128` для одного компьютера (это адрес замыкания IPv6) или как `fe80::7a31:c1ff:0000:0000/96` для небольшой сети. `0.0.0.0/0` представляет все адреса IPv4, а `::0/0` — все адреса IPv6. Чтобы указать один компьютер, используйте длину маски 32 для IPv4 или 128 для IPv6. Опускать замыкающие нули в сетевом адресе нельзя.

Запись, сделанная в формате IPv4, подойдёт только для подключений по IPv4, а запись в формате IPv6 подойдёт только для подключений по IPv6, даже если представленный адрес находится в диапазоне IPv4-в-IPv6.

Вы также можете прописать значение `all`, чтобы указать любой IP-адрес, `samehost`, чтобы указать любые IP-адреса данного сервера, или `samenet`, чтобы указать любой адрес любой подсети, к которой сервер подключён напрямую.

Если определено имя компьютера (всё, что не является диапазоном IP-адресов или специальным ключевым словом, воспринимается как имя компьютера), то оно сравнивается с результатом обратного преобразования IP-адреса клиента (например, обратного DNS-запроса, если используется DNS). При сравнении имён компьютеров регистр не учитывается. Если имена совпали, выполняется прямое преобразование имени (например, прямой DNS-запрос) для проверки, относится ли клиентский IP-адрес к адресам, соответствующим имени. Если двусторонняя проверка пройдена, запись считается соответствующей компьютеру. (В качестве имени узла в файле `pg_hba.conf` должно указываться то, что возвращается при преобразовании IP-адреса клиента в имя, иначе строка не будет соответствовать узлу. Некоторые базы данных имён позволяют связать с одним IP-адресом несколько имён узлов, но операционная система при попытке разрешить IP-адрес возвращает только одно имя.)

Указание имени, начинающееся с точки (`.`), соответствует суффиксу актуального имени узла. Так, `.example.com` будет соответствовать `foo.example.com` (а не только `example.com`).

Когда в `pg_hba.conf` указываются имена узлов, следует добиться, чтобы разрешение имён выполнялось достаточно быстро. Для этого может быть полезен локальный кеш разрешения имён, например, `nscd`. Вы также можете включить конфигурационный параметр `log_hostname`, чтобы видеть в журналах имя компьютера клиента вместо IP-адреса.

Эти поля не применимы к записям `local`.

Примечание

Пользователи часто задаются вопросом, почему имена серверов обрабатываются таким сложным, на первый взгляд, способом, с разрешением двух имён, включая обратный запрос клиентского IP-адреса. Это усложняет процесс в случае, если обратная DNS-запись клиента не установлена или включает в себя нежелательное имя узла. Такой способ избран, в первую очередь, для повышения эффективности: в этом случае соединение требует максимум два запроса разрешения, один прямой и один обратный. Если есть проблема разрешения с каким-то адресом, то она остаётся проблемой этого клиента. Гипотетически, могла бы быть реализована возможность во время каждой попытки соединения выполнять только прямой запрос для разрешения каждого имени сервера, упомянутого в `pg_hba.conf`. Но если список имён велик, процесс был бы довольно медленным, а в случае наличия проблемы разрешения у одного имени сервера, это стало бы общей проблемой.

Также обратный запрос необходим для того, чтобы реализовать возможность соответствия суффиксов, поскольку для сопоставления с шаблоном требуется знать фактическое имя компьютера клиента.

Обратите внимание, что такое поведение согласуется с другими популярными реализациями контроля доступа на основе имён, такими как Apache HTTP Server и TCP Wrappers.

`IP-адрес`
`IP-маска`

Эти два поля могут быть использованы как альтернатива записи *`IP-адрес/длина-маски`*. Вместо того, чтобы указывать длину маски, в отдельном столбце указывается сама маска. Например, `255.0.0.0` представляет собой маску CIDR для IPv4 длиной 8 бит, а `255.255.255.255` представляет маску CIDR длиной 32 бита.

Эти поля не применимы к записям `local`.

`метод-аутентификации`

Указывает метод аутентификации, когда подключение соответствует этой записи. Варианты выбора приводятся ниже; подробности в Разделе 19.3. Все значения воспринимаются с учётом регистра и должны быть записаны в нижнем регистре, даже аббревиатуры типа `ldap`.

`trust`

Разрешает безусловное подключение. Этот метод позволяет тому, кто может подключиться к серверу с базой данных Postgres Pro, войти под любым желаемым пользователем Postgres Pro без введения пароля и без какой-либо другой аутентификации. За подробностями обратитесь к Разделу 19.4.

`reject`

Отклоняет подключение безусловно. Эта возможность полезна для «фильтрации» некоторых серверов группы, например, строка `reject` может отклонить попытку подключения одного компьютера, при этом следующая строка позволяет подключиться остальным компьютерам в той же сети.

`scram-sha-256`

Проверяет пароль пользователя, производя аутентификацию SCRAM-SHA-256. За подробностями обратитесь к Разделу 19.5.

`md5`

Проверяет пароль пользователя, производя аутентификацию SCRAM-SHA-256 или MD5. За подробностями обратитесь к Разделу 19.5.

`password`

Требует для аутентификации введения клиентом незашифрованного пароля. Поскольку пароль посылается простым текстом через сеть, такой способ не стоит использовать, если сеть не вызывает доверия. За подробностями обратитесь к Разделу 19.5.

`gss`

Для аутентификации пользователя использует GSSAPI. Этот способ доступен только для подключений по TCP/IP. За подробностями обратитесь к Разделу 19.6. Он может применяться в сочетании с шифрованием GSSAPI.

`sspi`

Для аутентификации пользователя использует SSPI. Способ доступен только для Windows. За подробностями обратитесь к Разделу 19.7.

`ident`

Получает имя пользователя операционной системы клиента, связываясь с сервером Ident, и проверяет, соответствует ли оно имени пользователя базы данных. Аутентификация ident может использоваться только для подключений по TCP/IP. Для локальных подключений применяется аутентификация peer. За подробностями обратитесь к Разделу 19.8.

`peer`

Получает имя пользователя операционной системы клиента из операционной системы и проверяет, соответствует ли оно имени пользователя запрашиваемой базы данных. Доступно только для локальных подключений. За подробностями обратитесь к Разделу 19.9.

`ldap`

Проводит аутентификацию, используя сервер LDAP. За подробностями обратитесь к Разделу 19.10.

`radius`

Проводит аутентификацию, используя сервер RADIUS. За подробностями обратитесь к Разделу 19.11.

`cert`

Проводит аутентификацию, используя клиентский сертификат SSL. За подробностями обратитесь к Разделу 19.12.

`pam`

Проводит аутентификацию, используя службу подключаемых модулей аутентификации (PAM), предоставляемую операционной системой. За подробностями обратитесь к Разделу 19.13.

`bsd`

Проводит аутентификацию, используя службу аутентификации BSD, предоставляемую операционной системой. За подробностями обратитесь к Разделу 19.14.

параметры-аутентификации

После поля *метод-аутентификации* может идти поле (поля) вида *имя=значение*, определяющее параметры метода аутентификации. Подробнее о параметрах, доступных для различных методов аутентификации, рассказывается ниже.

Помимо описанных далее параметров, относящихся к различным методам, есть один общий параметр аутентификации `clientcert`, который можно задать в любой записи `hostssl`. Он может принимать значения `verify-ca` и `verify-full`. В обоих случаях клиент должен предоставить годный (доверенный) сертификат SSL, но вариант `verify-full` дополнительно требует, чтобы значение `cn` (Common Name, Общее имя) в сертификате совпадало с именем пользователя или применимым сопоставлением. Это указание работает подобно методу аутентификации `cert` (см. Раздел 19.12), но позволяет связать проверку клиентских сертификатов с любым методом аутентификации, поддерживающим записи `hostssl`.

В любой записи, включающей аутентификацию по клиентскому сертификату (то есть в записи с методом аутентификации `cert` или с параметром `clientcert`), в дополнительном параметре `clientname` можно выбрать сопоставляемый при проверке элемент учётных данных в клиентском сертификате. Этот параметр может иметь одно из двух значений. Если указано значение `clientname=CN` (оно подразумевается по умолчанию), имя пользователя сопоставляется с полем `Common Name (CN)`. Если указано значение `clientname=DN` — с полем `Distinguished name (DN)`. Этот вариант, вероятно, будет лучше использовать с сопоставлениями имён пользователей. Сравнение `DN` выполняется в формате *RFC 2253*. Извлечь поле `DN` в этом формате из клиентского сертификата можно так:

```
openssl x509 -in myclient.crt -noout -subject -nameopt RFC2253 | sed "s/^subject=//"
```

Использовать данный вариант следует с осторожностью, особенно когда для сопоставлений `DN` применяются регулярные выражения.

`include`

Эта строка будет заменена на содержимое указанного файла.

`include_if_exists`

Эта строка будет заменена на содержимое указанного файла, если файл существует. В противном случае в журнал вносится сообщение о том, что файл был пропущен.

`include_dir`

Эта строка будет заменена на содержимое всех файлов, найденных в каталоге, если они не начинаются с `.` и заканчиваются на `.conf`, обработанными в алфавитном порядке (определяемому правилами, принятыми в C, т. е. цифры идут перед буквами, а буквы в верхнем регистре — перед буквами в нижнем).

Файлы, включённые в конструкции, начинающиеся с `@`, читаются, как список имён, разделённых запятыми или пробелами. Комментарии предваряются знаком `#`, как и в файле `pg_hba.conf`, и вложенные `@` конструкции допустимы. Если только имя файла, начинающегося с `@` не является абсолютным путём.

Поскольку записи файла `pg_hba.conf` рассматриваются последовательно для каждого подключения, порядок записей имеет большое значение. Обычно более ранние записи определяют чёткие критерии для соответствия параметров подключения, но для методов аутентификации допускают послабления. Напротив, записи более поздние смягчают требования к соответствию параметров подключения, но усиливают их в отношении методов аутентификации. Например, некто желает использовать `trust` аутентификацию для локального подключения по TCP/IP, но при этом запрашивать пароль для удалённых подключений по TCP/IP. В этом случае запись, устанавливающая аутентификацию `trust` для подключения адреса 127.0.0.1, должна предшествовать записи, определяющей аутентификацию по паролю для более широкого диапазона клиентских IP-адресов.

Подсказка

Чтобы подключиться к конкретной базе данных, пользователь не только должен пройти все проверки файла `pg_hba.conf`, но должен иметь привилегию `CONNECT` для подключения к базе данных. Если вы хотите ограничить доступ к базам данных для определённых пользователей, проще предоставить/отозвать привилегию `CONNECT`, нежели устанавливать правила в записях файла `pg_hba.conf`.

Примеры записей файла `pg_hba.conf` показаны в Примере 19.1. Обратитесь к следующему разделу за более подробной информацией по методам аутентификации.

Пример 19.1. Примеры записей `pg_hba.conf`

```
# Позволяет любому пользователю локальной системы подключаться
# к любой базе данных, используя любое имя пользователя баз данных, через
# Unix-сокеты (по умолчанию для локальных подключений).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust

# То же, но для локальных замкнутых подключений по TCP/IP.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             127.0.0.1/32            trust

# То же, что и в предыдущей строке, но с указанием
# сетевой маски в отдельном столбце
#
# TYPE  DATABASE        USER            IP-ADDRESS      IP-MASK             METHOD
host    all             all             127.0.0.1       255.255.255.255     trust

# То же для IPv6.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             ::1/128                 trust

# То же самое, но с использованием имени узла
# (обычно покрывает и IPv4, и IPv6).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             localhost               trust

# То же самое с использованием регулярного выражения для DATABASE, позволяющего
 подключиться
# к любым базам данных с именем, начинающимся с "db" и заканчивающимся
# на число, содержащее от 2 до 4 цифр (например, "db1234" или "db12").
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    "/^db\d{2,4}$"  all             localhost               trust

# Позволяет любому пользователю любого компьютера с IP-адресом
# 192.168.93.x подключаться к базе данных "postgres"
# с именем, которое сообщает для данного подключения ident
# (как правило, имя пользователя операционной системы).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.93.0/24         ident
```

```
# Позволяет любому пользователю компьютера 192.168.12.10 подключаться
# к базе данных "postgres", если он передаёт правильный пароль.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        scram-sha-256

# Позволяет любым пользователям с компьютеров в домене example.com
# подключаться к любой базе данных, если передаётся правильный пароль.
#
# Для всех пользователей требуется аутентификация SCRAM, за исключением
# пользователя 'mike', который использует старый клиент, не поддерживающий
# аутентификацию SCRAM.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             mike            .example.com            md5
host    all             all             .example.com            scram-sha-256

# В случае отсутствия предшествующих строчек с "host", следующие две строки
# откажут в подключении с 192.168.54.1 (поскольку данная запись будет
# выбрана первой), но разрешат подключения GSSAPI с любых других
# адресов. С нулевой маской ни один бит из IP-адреса компьютера
# не учитывается, так что этой строке соответствует любой компьютер.
# Незашифрованные средствами GSSAPI подключения (они "опускаются"
# до третьей строки, так как записи "hostgssenc"
# соответствуют только
# зашифрованные подключения) принимаются, но только с адреса 192.168.12.10.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
hostgssenc all          all             0.0.0.0/0               gss
host    all             all             192.168.12.10/32        gss

# Позволяет пользователям с любого компьютера 192.168.x.x подключаться
# к любой базе данных, если они проходят проверку ident. Если же ident
# говорит, например, что это пользователь "bryanh" и он запрашивает
# подключение как пользователь Postgres Pro "guest1", подключение
# будет разрешено, если в файле pg_ident.conf есть сопоставление
# "omicron", позволяющее пользователю "bryanh" подключаться как "guest1".
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.0.0/16          ident map=omicron

# Если для локальных подключений предусмотрены только эти четыре строки,
# они позволят локальным пользователям подключаться только к своим
# базам данных (базам данных с именами, совпадающими с
# именами пользователей баз данных), кроме пользователей, чьё имя
# заканчивается на "helpdesk", администраторов
# или членов роли "support", которые могут подключиться к любой БД.
# Список имён администраторов содержится в файле $PGDATA/admins.
# Пароли требуются в любом случае.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   sameuser        all                                     md5
local   all             /^.*helpdesk$                           md5
local   all             @admins                                 md5
local   all             +support                                md5

# Последние две строчки выше могут быть объединены в одну:
```

```
local   all             @admins,+support                        md5

# В столбце DATABASE также могут указываться списки и имена файлов:
local   db1,db2,@demodbs  all                                   md5
```

19.2. Файл сопоставления имён пользователей

Когда используется внешняя система аутентификации, например Ident или GSSAPI, имя пользователя операционной системы, устанавливающего подключение, может не совпадать с именем целевого пользователя (роли) базы данных. В этом случае можно применить сопоставление имён пользователей, чтобы сменить имя пользователя операционной системы на имя пользователя БД. Чтобы задействовать сопоставление имён, укажите `map=`*`имя-сопоставления`* в поле параметров в `pg_hba.conf`. Этот параметр поддерживается для всех методов аутентификации, которые принимают внешние имена пользователей. Так как для разных подключений могут требоваться разные сопоставления, сопоставление определяется параметром *`имя-сопоставления`* в `pg_hba.conf` для каждого отдельного подключения.

Сопоставления имён пользователя определяются в файле сопоставления ident, который по умолчанию называется `pg_ident.conf`и хранится в каталоге данных кластера. (Файл сопоставления может быть помещён и в другое место, обратитесь к информации о настройке параметра ident_file.) Файл сопоставления ident содержит строки общих форм:

```
имя-сопоставления имя-системного пользователя имя-пользователя-БД
include файл
include_if_exists файл
include_dir каталог
```

Комментарии и пробелы применяются так же, как и в файле `pg_hba.conf`. *`имя-сопоставления`* является произвольным именем, на которое будет ссылаться файл сопоставления файла `pg_hba.conf`. Два других поля указывают имя пользователя операционной системы и соответствующее имя пользователя базы данных. *`имя-сопоставления`* может быть использовано неоднократно, чтобы указывать множественные сопоставления пользовательских имён в рамках одного файла сопоставления.

Что касается `pg_hba.conf`, строки в этом файле могут быть директивами включения, которые следуют тем же правилам.

Файл `pg_ident.conf` прочитывается при запуске системы, а также в тот момент, когда основной сервер получает сигнал SIGHUP. Если вы редактируете файл во время работы системы, необходимо послать сигнал процессу postmaster (используя `pg_ctl reload`, вызвав SQL-функцию `pg_reload_conf()` или выполнив `kill -HUP`), чтобы он прочел обновлённый файл.

Системное представление `pg_ident_file_mappings` может быть полезно для предварительной проверки изменений в файле `pg_ident.conf` или для диагностики проблем, когда перезагрузка этого файла не даёт желаемого эффекта. Строки в этом представлении, содержащие в поле `error` не NULL, указывают на проблемы в соответствующих строках файла.

Нет никаких ограничений по количеству пользователей баз данных, на которые может ссылаться пользователь операционной системы, и наоборот. Тем не менее записи в файле скорее подразумевают, что «пользователь этой операционной системы может подключиться как пользователь этой базы данных», нежели показывают, что эти имена пользователей эквивалентны. Подключение разрешается, если существует запись в файле сопоставления, соединяющая имя, полученное от внешней системы аутентификации, с именем пользователя базы данных, под которым пользователь хочет подключиться. Значение `all` можно использовать в качестве *`имени-пользователя-БД`*, чтобы указать, что если оно совпадает с *`именем-системного-пользователя`*, то этому пользователю разрешено входить в систему как любому из существующих пользователей базы данных. Ключевое слово `all` теряет своё особое значение.

Если *`имя-пользователя-БД`* начинается с символа `+`, пользователь операционной системы может войти в систему как любой пользователь, включённый в эту роль, аналогично тому, как имена пользователей, начинающиеся с `+`, обрабатываются в `pg_hba.conf`. Таким образом, знак `+` означает «совпадение любых ролей, которые прямо или косвенно являются членами роли», тогда как имя без знака + является подходящим только для этой конкретной роли. Заключение имени пользователя, начинающегося с +, в кавычки приводит к тому, что + теряет своё особое значение.

Если поле *`имени-системного-пользователя`* начинается со знака (`/`), оставшаяся его часть рассматривается как регулярное выражение. (Подробнее синтаксис регулярных выражений Postgres Pro описан в Подразделе 9.7.3.1.) Регулярное выражение может включать в себя одну группу или заключённое в скобки подвыражение. На часть имени системного пользователя, совпавшую с названием группы, можно сослаться в поле *`database-username`*, написав `\1` (с одной обратной косой). Это позволяет сопоставить несколько имён пользователя с одной строкой, что особенно удобно для простых замен. Например, эти строки

```
mymap   /^(.*)@mydomain\.com$      \1
mymap   /^(.*)@otherdomain\.com$   guest
```

удалят часть домена для имён пользователей, которые заканчиваются на `@mydomain.com`, и позволят пользователям, чьё имя пользователя системы заканчивается на `@otherdomain.com`, подключиться как `guest`. Заключение *`имени-пользователя-БД`*, содержащего `\1`, в кавычки *не* заставит `\1` потерять своё особое значение.

Если поле *`имя-пользователя-БД`* начинается со знака (`/`), оставшаяся его часть рассматривается как регулярное выражение. Когда поле *`имя-пользователя-БД`* является регулярным выражением, в нём нельзя использовать значение `\1` для ссылки на название группы из поля *`имя-системного-пользователя`*.

> ### Подсказка
>
> Помните, что по умолчанию, регулярное выражение может совпасть только с частью строки. Разумным выходом будет использование символов `^` и `$`, как показано в примере выше, для принудительного совпадения со всем именем пользователя операционной системы

Файл `pg_ident.conf`, который может быть использован в сочетании с файлом `pg_hba.conf` (см. Пример 19.1), показан в Примере 19.2. В этом примере любым пользователям компьютеров в сети 192.168 с именами, отличными от `bryanh`, `ann` или `robert`, будет отказано в доступе. Пользователь системы `robert` получит доступ только тогда, когда подключается как пользователь Postgres Pro `bob`, а не как `robert`, или какой-либо другой пользователь. Пользователь `ann` сможет подключиться только как `ann`. Пользователь `bryanh` сможет подключиться как `bryanh` или как `guest1`.

Пример 19.2. Пример файла `pg_ident.conf`

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

omicron         bryanh                  bryanh
omicron         ann                     ann
# на этих машинах у пользователя bob имя robert
omicron         robert                  bob
# bryanh также может подключаться как guest1
omicron         bryanh                  guest1
```

19.3. Методы аутентификации

Postgres Pro предлагает к использованию широкий набор методов аутентификации пользователей:

- Аутентификация trust, при которой сервер доверяет пользователям, никак не проверяя их.
- Аутентификация password, требующая ввода пароля пользователем.
- Аутентификация GSSAPI, использующая библиотеку безопасности, совместимую с GSSAPI. Обычно этот метод применяется при использовании специальной службы аутентификации, Kerberos или Microsoft Active Directory.

- Аутентификация SSPI, использующая протокол, подобный GSSAPI, но предназначенный для Windows.
- Аутентификация ident, для которой используется служба, реализующая «Identification Protocol» (*RFC 1413*) на клиентском компьютере. (Для подключений через локальный сокет Unix этот метод работает как peer.)
- Аутентификация peer, которая полагается на средства операционной системы, позволяющие узнать пользователя процесса на другой стороне локального подключения. Для удалённых подключений она не поддерживается.
- Аутентификация LDAP, работающая с сервером аутентификации LDAP.
- Аутентификация RADIUS, работающая с сервером аутентификации RADIUS.
- Аутентификация по сертификату, требующая использования клиентами SSL-подключения и построенная на проверке передаваемых ими сертификатов SSL.
- Аутентификация PAM, реализуемая с использованием библиотеки PAM.
- Аутентификация BSD, основанная на использовании механизма аутентификации BSD (в настоящее время поддерживается только в системе OpenBSD).

Для локальных подключений обычно рекомендуется использовать метод peer, хотя в некоторых обстоятельствах может быть достаточно и режима trust. Для удалённых подключений самой простой будет аутентификация по паролю. Все остальные варианты требуют использования некоторой внешней инфраструктуры безопасности (обычно это служба аутентификации или центр сертификации, выдающий сертификаты SSL) либо поддерживаются не на всех платформах.

Более подробно все эти методы аутентификации описываются в следующих разделах.

19.4. Аутентификация trust

Когда указан способ аутентификации `trust`, Postgres Pro предполагает, что любой подключающийся к серверу авторизован для доступа к базе данных вне зависимости от указанного имени пользователя базы данных (даже если это имя суперпользователя). Конечно, ограничения, прописанные в столбцах `база` и `пользователь`, продолжают работать. Этот метод должен применяться только в том случае, когда на уровне операционной системы обеспечена адекватная защита от подключений к серверу.

Аутентификация `trust` очень удобна для локальных подключений на однопользовательской рабочей станции. Но сам по себе этот метод обычно *не* подходит для машин с несколькими пользователями. Однако вы можете использовать `trust` даже на многопользовательской машине, если ограничите доступ к файлу Unix-сокета сервера на уровне файловой системы. Для этого установите конфигурационные параметры `unix_socket_permissions` (и, возможно, `unix_socket_group`) как описано в Разделе 18.3. Либо вы можете установить конфигурационный параметр `unix_socket_directories`, чтобы разместить файл сокета в должным образом защищённом каталоге.

Установка разрешений на уровне файловой системы помогает только в случае подключений через Unix-сокеты. На локальные подключения по TCP/IP ограничения файловой системы не влияют. Поэтому, если вы хотите использовать разрешения файловой системы для обеспечения локальной безопасности, уберите строку `host ... 127.0.0.1 ...` из `pg_hba.conf` или смените метод аутентификации.

Метод аутентификации `trust` для подключений по TCP/IP допустим только в случае, если вы доверяете каждому пользователю компьютера, получившему разрешение на подключение к серверу строками файла `pg_hba.conf`, указывающими метод `trust`. Не стоит использовать `trust` для любых подключений по TCP/IP, отличных от localhost (127.0.0.1).

19.5. Аутентификация password

Существует несколько методов аутентификации по паролю. Они работают примерно одинаково, но различаются тем, как пароли пользователей хранятся на сервере и как пароль передаётся от клиента по каналу связи.

`scram-sha-256`

С методом `scram-sha-256` выполняется аутентификация SCRAM-SHA-256, как описано в *RFC 7677*. Она производится по схеме вызов-ответ, которая предотвращает перехват паролей через недоверенные соединения и поддерживает хранение паролей на сервере в виде криптографического хеша, что считается безопасным.

Это наиболее безопасный из существующих на данный момент методов, но он не поддерживается старыми клиентскими библиотеками.

`md5`

Для метода `md5` реализован менее безопасный механизм вызов-ответ. Он предотвращает перехват паролей и предусматривает хранение паролей на сервере в зашифрованном виде, но не защищает в случае похищения хешей паролей с сервера. Кроме того, алгоритм хеширования MD5 в наши дни уже может не защитить от целенаправленных атак.

Для облегчения перехода от метода `md5` к более новому методу SCRAM, если в качестве метода аутентификации в `pg_hba.conf` указан `md5`, но пароль пользователя на сервере зашифрован для SCRAM (см. ниже), автоматически будет производиться аутентификация на базе SCRAM.

`password`

С методом `password` пароль передаётся в открытом виде и поэтому является уязвимым для атак с перехватом трафика. Его следует избегать всегда, если это возможно. Однако если подключение защищено SSL, метод `password` может быть безопасен. (Хотя аутентификация по сертификату SSL может быть лучшим выбором когда используется SSL).

Пароли баз данных Postgres Pro отделены от паролей пользователей операционной системы. Пароли всех пользователей базы данных хранятся в системном каталоге `pg_authid`. Управлять паролями можно либо используя SQL-команды CREATE ROLE и ALTER ROLE, например, **`CREATE ROLE foo WITH LOGIN PASSWORD 'secret'`**, либо с помощью команды psql `\password`. Если пароль для пользователя не задан, вместо него хранится NULL, и пройти аутентификацию по паролю этот пользователь не сможет.

Доступность различных методов аутентификации по паролю зависит от того, как пароли пользователей шифруются на сервере (или, говоря точнее, хешируются). Это определяется параметром конфигурации password_encryption в момент назначения пароля. Если пароль шифруется в режиме `scram-sha-256`, его можно будет использовать для методов аутентификации `scram-sha-256` и `password` (но в последнем случае он будет передаваться открытым текстом). В случае указания метода аутентификации `md5` при этом произойдёт автоматический переход к использованию `scram-sha-256`, как сказано выше, так что этот вариант тоже будет работать. Если пароль шифруется в режиме `md5`, его можно будет использовать только для методов аутентификации `md5` и `password` (и в последнем случае он так же будет передаваться открытым текстом). (Ранние версии Postgres Pro поддерживали хранение паролей на сервере в открытом виде, но теперь это невозможно.) Чтобы просмотреть хранящиеся в БД хеши паролей, обратитесь к системному каталогу `pg_authid`.

Для перевода существующей инсталляции с `md5` на `scram-sha-256`, после того как все клиентские библиотеки будут обновлены до версий, поддерживающих SCRAM, задайте `password_encryption = 'scram-sha-256'` в `postgresql.conf`, добейтесь, чтобы все пользователи сменили свои пароли, а затем поменяйте указания метода аутентификации в `pg_hba.conf` на `scram-sha-256`.

19.6. Аутентификация GSSAPI

GSSAPI — стандартизированный протокол для безопасной аутентификации, определённый в *RFC 2743*. Postgres Pro поддерживает применение GSSAPI для аутентификации или шифрования соединения, а также для шифрования с аутентификацией. GSSAPI обеспечивает автоматическую проверку подлинности (единый вход) для систем, которые её поддерживают. Проверка подлинности реализуется безопасно, но данные, передаваемые через подключение к БД, будут защищены, только если используется шифрование GSSAPI или SSL.

Поддержка GSSAPI должна быть включена при сборке Postgres Pro.

При работе с Kerberos GSSAPI использует стандартные имена принципалов служб (аутентификационные идентификаторы) в формате `имя_службы/имя_сервера@область`. Имя принципала, используемое конкретным установленным экземпляром Postgres Pro, не зашито в сервере ни в каком виде; оно задаётся в файле *keytab*, прочитав который сервер определяет назначенный ему идентификатор. Если в данном файле задано несколько принципалов, сервер выберет любой из них. В качестве области сервера выбирается предпочитаемая область, заданная в доступных серверу файлах конфигурации Kerberos.

Подключающийся к серверу клиент должен знать имя принципала данного сервера. Обычно в компоненте `имя_службы` принципала фигурирует `postgres`, но параметр подключения krbsrvname в libpq позволяет задать и другое значение. В компоненте `имя_сервера` задаётся полностью определённое имя узла, к которому будет подключаться libpq. В качестве области выбирается предпочитаемая область, заданная в доступных клиенту файлах конфигурации Kerberos.

У клиента также должно быть назначенное ему имя принципала (и он должен иметь действительный билет для этого принципала). Чтобы GSSAPI проверил подлинность клиента, имя принципала клиента должно быть связано с именем пользователя базы Postgres Pro. Определить такие сопоставления можно в файле `pg_ident.conf`; например, `pgusername@realm` можно сопоставить с именем `pgusername`. Также возможно использовать в качестве имени роли в Postgres Pro полное имя принципала `username@realm` без какого-либо сопоставления.

Postgres Pro также может сопоставлять принципалы клиентов именам пользователей, просто убирая компонент области из имени принципала. Эта возможность оставлена для обратной совместимости, и использовать её крайне нежелательно, так как при этом оказывается невозможно различить разных пользователей, имеющих одинаковые имена, но приходящих из разных областей. Чтобы включить её, установите для `include_realm` значение 0. В простых конфигурациях с одной областью исключение области в сочетании с параметром `krb_realm` (который позволяет ограничить область пользователя одним значением, заданным в `krb_realm` parameter) будет безопасным, но менее гибким вариантом по сравнению с явным описанием сопоставлений в `pg_ident.conf`.

Размещение файла keytab на сервере задаётся параметром конфигурации krb_server_keyfile. По соображениям безопасности рекомендуется использовать отдельный файл для сервера Postgres Pro, а не разрешать серверу читать системный keytab. Предоставьте пользователю, от имени которого работает сервер Postgres Pro, право чтения этого файла (право записи давать не нужно). (См. также Раздел 17.1.)

Файл таблицы ключей генерируется с использованием ПО Kerberos; подробнее это описано в документации Kerberos. Следующий пример показывает, как его сгенерировать, используя программу kadmin в MIT-реализации Kerberos:

```
kadmin% addprinc -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

Для метода аутентификации GSSAPI поддерживаются следующие параметры:

`include_realm`

> Когда этот параметр равен 0, из принципала аутентифицированного пользователя убирается область, и оставшееся имя проходит сопоставление имён (см. Раздел 19.2). Этот вариант не рекомендуется и поддерживается в основном для обратной совместимости, так как он небезопасен в окружениях с несколькими областями, если только дополнительно не задаётся `krb_realm`. Более предпочтительный вариант — оставить значение `include_realm` по умолчанию (1) и задать в `pg_ident.conf` явное сопоставление для преобразования имён принципалов в имена пользователей Postgres Pro.

`map`

Разрешает сопоставление принципалов клиентов пользователям баз данных. За подробностями обратитесь к Разделу 19.2. Для принципала GSSAPI/Kerberos, такого как `username@EXAMPLE.COM` (или более редкого `username/hostbased@EXAMPLE.COM`), именем пользователя в сопоставлении будет `username@EXAMPLE.COM` (или `username/hostbased@EXAMPLE.COM`, соответственно), если `include_realm` не равно 0; в противном случае именем системного пользователя в сопоставлении будет `username` (или `username/hostbased`).

`krb_realm`

Устанавливает область, с которой будут сверяться имена принципалов пользователей. Если этот параметр задан, подключаться смогут только пользователи из этой области. Если не задан, подключаться смогут пользователи из любой области, в зависимости от установленного сопоставления имён пользователей.

В дополнение к этим параметрам, которые могут быть разными в разных записях `pg_hba.conf`, существует параметр конфигурации krb_caseins_users, действующий на уровне сервера. Если он равен true, принципалы клиентов сопоставляются с записями пользователей без учёта регистра. В значении `krb_realm`, если оно задано, регистр символов тоже не учитывается.

19.7. Аутентификация SSPI

SSPI — технология Windows для защищённой аутентификации с единственным входом. Postgres Pro использует SSPI в режиме `negotiate`, который применяет Kerberos, когда это возможно, и автоматически возвращается к NTLM в других случаях. SSPI и GSSAPI взаимодействуют как клиенты и серверы, например, клиент SSPI может проходить аутентификацию на сервере GSSAPI. Рекомендуется использовать SSPI для клиентов и серверов, работающих в Windows, и GSSAPI на других платформах.

Если используется аутентификация Kerberos, SSPI работает так же, как GSSAPI; подробнее об этом рассказывается в Разделе 19.6.

Для SSPI доступны следующие параметры конфигурации:

`include_realm`

Когда этот параметр равен 0, из принципала аутентифицированного пользователя убирается область, и оставшееся имя проходит сопоставление имён (см. Раздел 19.2). Этот вариант не рекомендуется и поддерживается в основном для обратной совместимости, так как он небезопасен в окружениях с несколькими областями, если только дополнительно не задаётся `krb_realm`. Более предпочтительный вариант — оставить значение `include_realm` по умолчанию (1) и задать в `pg_ident.conf` явное сопоставление для преобразования имён принципалов в имена пользователей Postgres Pro.

`compat_realm`

Если равен 1, для параметра `include_realm` применяется имя домена, совместимое с SAM (также известное как имя NetBIOS). Это вариант по умолчанию. Если он равен 0, для имени принципала Kerberos применяется действительное имя области.

Этот параметр можно отключить, только если ваш сервер работает под именем доменного пользователя (в том числе, виртуального пользователя службы на компьютере, включённом в домен) и все клиенты, проходящие проверку подлинности через SSPI, также используют доменные учётные записи; в противном случае аутентификация не будет выполнена.

`upn_username`

Если этот параметр включён вместе с `compat_realm`, для аутентификации применяется имя Kerberos UPN. Если он отключён (по умолчанию), применяется SAM-совместимое имя пользователя. По умолчанию у новых учётных записей эти два имени совпадают.

Заметьте, что libpq использует имя, совместимое с SAM, если имя не задано явно. Если вы применяете libpq или драйвер на его базе, этот параметр следует оставить отключённым, либо явно задавать имя пользователя в строке подключения.

`map`

Позволяет сопоставить пользователей системы с пользователями баз данных. За подробностями обратитесь к Разделу 19.2. Для принципала SSPI/Kerberos, такого как `username@EXAMPLE.COM` (или более редкого `username/hostbased@EXAMPLE.COM`), именем пользователя в сопоставлении будет `username@EXAMPLE.COM` (или `username/hostbased@EXAMPLE.COM`, соответственно), если `include_realm` не равно 0; в противном случае именем системного пользователя в сопоставлении будет `username` (или `username/hostbased`).

`krb_realm`

Устанавливает область, с которой будут сверяться имена принципалов пользователей. Если этот параметр задан, подключаться смогут только пользователи из этой области. Если не задан, подключаться смогут пользователи из любой области, в зависимости от установленного сопоставления имён пользователей.

19.8. Аутентификация ident

Метод аутентификации ident работает, получая имя пользователя операционной системы клиента от сервера Ident и используя его в качестве разрешённого имени пользователя базы данных (с возможным сопоставлением имён пользователя). Способ доступен только для подключений по TCP/IP.

Примечание

Когда для локального подключения (не TCP/IP) указан ident, вместо него используется метод аутентификации peer (см. Раздел 19.9).

Для метода `ident` доступны следующие параметры конфигурации:

`map`

Позволяет сопоставить имена пользователей системы и базы данных. За подробностями обратитесь к Разделу 19.2.

Протокол «Identification» (Ident) описан в *RFC 1413*. Практически каждая Unix-подобная операционная система поставляется с сервером Ident, по умолчанию слушающим TCP-порт 113. Базовая функция этого сервера — отвечать на вопросы, вроде «Какой пользователь инициировал подключение, которое идет через твой порт *`X`* и подключается к моему порту *`Y`*?». Поскольку после установления физического подключения Postgres Pro знает и *`X`*, и *`Y`*, он может опрашивать сервер Ident на компьютере клиента и теоретически может определять пользователя операционной системы при каждом подключении.

Недостатком этой процедуры является то, что она зависит от интеграции с клиентом: если клиентская машина не вызывает доверия или скомпрометирована, злоумышленник может запустить любую программу на порту 113 и вернуть любое имя пользователя на свой выбор. Поэтому этот метод аутентификации подходит только для закрытых сетей, где каждая клиентская машина находится под жёстким контролем и где администраторы операционных систем и баз данных работают в тесном контакте. Другими словами, вы должны доверять машине, на которой работает сервер Ident. Помните предупреждение:

> Протокол Ident не предназначен для использования в качестве протокола авторизации и контроля доступа.
>
> —RFC 1413

У некоторых серверов Ident есть нестандартная возможность, позволяющая зашифровать возвращаемое имя пользователя, используя ключ, который известен только администратору исходного компьютера. Эту возможность *нельзя* использовать с Postgres Pro, поскольку Postgres Pro не сможет расшифровать возвращаемую строку и получить фактическое имя пользователя.

19.9. Аутентификация peer

Метод аутентификации peer работает, получая имя пользователя операционной системы клиента из ядра и используя его в качестве разрешённого имени пользователя базы данных (с возможностью сопоставления имён пользователя). Этот метод поддерживается только для локальных подключений.

Для метода `peer` доступны следующие параметры конфигурации:

`map`

> Позволяет сопоставить имена пользователей системы и базы данных. За подробностями обратитесь к Разделу 19.2.

Аутентификация peer доступна только в операционных системах, поддерживающих функцию `getpeereid()`, параметр сокета `SO_PEERCRED` или подобные механизмы. В настоящее время это Linux, большая часть разновидностей BSD, включая macOS, и Solaris.

19.10. Аутентификация LDAP

Данный метод аутентификации работает сходным с методом `password` образом, за исключением того, что он использует LDAP как метод подтверждения пароля. LDAP используется только для подтверждения пары «имя пользователя/пароль». Поэтому пользователь должен уже существовать в базе данных до того, как для аутентификации будет использован LDAP.

Аутентификация LDAP может работать в двух режимах. Первый режим называется простое связывание. В ходе аутентификации сервер связывается с характерным именем, составленным следующим образом: *`prefix`* *`username`* *`suffix`*. Обычно, параметр *`prefix`* используется для указания `cn=` или *`DOMAIN`*`\` в среде Active Directory. *`suffix`* используется для указания оставшейся части DN или в среде, отличной от Active Directory.

Во втором режиме, который мы называем поиск+связывание, сервер сначала связывается с каталогом LDAP с предопределённым именем пользователя и паролем, указанным в *`ldapbinddn`* и *`ldapbindpasswd`*, и выполняет поиск пользователя, пытающегося подключиться к базе данных. Если имя пользователя и пароль не определены, сервер пытается связаться с каталогом анонимно. Поиск выполняется в поддереве *`ldapbasedn`*, при этом проверятся точное соответствие имени пользователя атрибуту *`ldapsearchattribute`*. Как только при поиске находится пользователь, сервер заново связывается с каталогом уже как этот пользователь, с паролем, переданным клиентом, чтобы удостовериться, что учётная запись корректна. Этот же режим используется в схемах LDAP-аутентификации в другом программном обеспечении, например, в `pam_ldap` и `mod_authnz_ldap` в Apache. Данный вариант даёт больше гибкости в выборе расположения объектов пользователей, но при этом серверу LDAP отправляются два дополнительных запроса.

В обоих режимах используются следующие параметры конфигурации:

`ldapserver`

> Имена и IP-адреса LDAP-серверов для связи. Можно указать несколько серверов, разделяя их пробелами.

`ldapport`

> Номер порта для связи с LDAP-сервером. Если порт не указан, используется установленный по умолчанию порт библиотеки LDAP.

`ldapscheme`

> Значение `ldaps` выбирает протокол LDAPS. Это нестандартный способ использования LDAP поверх SSL, поддерживаемый некоторыми серверами LDAP. Альтернативную возможность предоставляет параметр `ldaptls`.

`ldaptls`

Значение 1 включает TLS-шифрование для защиты соединения Postgres Pro с LDAP-сервером. При этом используется операция `StartTLS`, описанная в *RFC 4513*. Альтернативную возможность предоставляет параметр `ldapscheme`.

Заметьте, что при использовании `ldapscheme` или `ldaptls` шифруется только трафик между сервером Postgres Pro и сервером LDAP. Соединение между сервером Postgres Pro и клиентом остаётся незашифрованным, если только и для него не включён SSL.

Следующие параметры используются только в режиме простого связывания:

`ldapprefix`

Эта строка подставляется перед именем пользователя во время формирования DN для связывания при аутентификации в режиме простого связывания.

`ldapsuffix`

Эта строка размещается после имени пользователя во время формирования DN для связывания, при аутентификации в режиме простого связывания.

Следующие параметры используются только в режиме поиск+связывание:

`ldapbasedn`

Корневая папка DN для начала поиска пользователя при аутентификации в режиме поиск+связывание.

`ldapbinddn`

DN пользователя для связи с каталогом при выполнении поиска в ходе аутентификации в режиме поиск+связывание.

`ldapbindpasswd`

Пароль пользователя для связывания с каталогом при выполнении поиска в ходе аутентификации в режиме поиск+связывание.

`ldapsearchattribute`

Атрибут для соотнесения с именем пользователя в ходе аутентификации поиск+связывание. Если атрибут не указан, будет использован атрибут `uid`.

`ldapsearchfilter`

Фильтр поиска, используемый для аутентификации в режиме поиск+связывание. Вхождения `$username` в нём будут заменяться именем пользователя. Это позволяет задавать более гибкие фильтры поиска, чем `ldapsearchattribute`.

`ldapurl`

Адрес LDAP по стандарту *RFC 4516*. Это альтернативный способ записи некоторых других параметров LDAP в более компактном и стандартном виде. Формат адреса таков:

```
ldap[s]://сервер[:порт]/basedn[?[атрибут][?[scope][?[фильтр]]]]
```

Здесь *scope* принимает значение `base`, `one` или `sub` (обычно последнее). По умолчанию подразумевается `base`, что не очень полезно при таком применении. В качестве *атрибута* может указываться один атрибут; в этом случае он используется как значение параметра `ldapsearchattribute`. Если *атрибут* не указан, в качестве значения `ldapsearchfilter` может использоваться *фильтр*.

Схема адреса `ldaps` выбирает для установления LDAP-подключений поверх SSL метод LDAPS, что равнозначно указанию `ldapscheme=ldaps`. Для применения шифрования LDAP с использованием операции `StartTLS` используйте обычную схему URL `ldap` и укажите параметр `ldaptls` в дополнение к `ldapurl`.

Для неанонимного связывания `ldapbinddn` и `ldapbindpasswd` должны быть указаны как раздельные параметры.

В настоящее время URL-адреса LDAP поддерживаются только с OpenLDAP и не поддерживаются в Windows.

Нельзя путать параметры конфигурации для режима простого связывания с параметрами для режима поиск+связывание, это ошибка.

В режиме поиск+связывание поиск может выполняться либо по одному атрибуту, указанному в `ldapsearchattribute`, либо по произвольному фильтру поиска, заданному в `ldapsearchfilter`. Указание `ldapsearchattribute=foo` равнозначно указанию `ldapsearchfilter="(foo=$username)"`. Если не указан ни один параметр, по умолчанию подразумевается `ldapsearchattribute=uid`.

Если Postgres Pro был скомпилирован с OpenLDAP в качестве клиентской библиотеки LDAP, параметр `ldapserver` может быть опущен. В этом случае за списком имён серверов и портов сервер обращается в DNS, к SRV-записи (по стандарту *RFC 2782*) с именем `_ldap._tcp.DOMAIN`, где `DOMAIN` извлекается из `ldapbasedn`.

Это пример конфигурации LDAP для простого связывания:

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example,
 dc=net"
```

Когда запрашивается подключение к серверу базы данных в качестве пользователя базы данных `someuser`, Postgres Pro пытается связаться с LDAP-сервером, используя DN `cn=someuser, dc=example, dc=net` и пароль, предоставленный клиентом. Если это подключение удалось, то доступ к базе данных будет открыт.

Пример конфигурации для режима поиск+связывание:

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net"
 ldapsearchattribute=uid
```

Когда запрашивается подключение к серверу базы данных в качестве пользователя базы данных `someuser`, Postgres Pro пытается связаться с сервером LDAP анонимно (поскольку `ldapbinddn` не был указан), выполняет поиск для `(uid=someuser)` под указанной базой DN. Если запись найдена, проводится попытка связывание с использованием найденной информации и паролем, предоставленным клиентом. Если вторая попытка связывания проходит успешно, предоставляется доступ к базе данных.

Пример той же конфигурации для режима поиск+связывание, но записанной в виде URL:

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

Такой URL-формат используется и другим программным обеспечением, поддерживающим аутентификацию по протоколу LDAP, поэтому распространять такую конфигурацию будет легче.

Пример конфигурации поиск+связывание, в котором `ldapsearchfilter` используется вместо `ldapsearchattribute` для прохождения аутентификации по идентификатору или почтовому адресу пользователя:

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net"
 ldapsearchfilter="(|(uid=$username)(mail=$username))"
```

Пример конфигурации поиск+связывание, в котором используется поиск SRV-записи в DNS для получения имени сервера и порта службы LDAP в домене `example.net`:

```
host ... ldap ldapbasedn="dc=example,dc=net"
```

Подсказка

Поскольку LDAP часто применяет запятые и пробелы для разделения различных частей DN, необходимо использовать кавычки при определении значения параметров, как показано в наших примерах.

19.10.1. Рекомендации по настройке общих LDAP-серверов

Чтобы настроить аутентификацию LDAP на сервере Postgres Pro, должны быть выполнены следующие требования:

- Сервер Postgres Pro должен иметь доступ к LDAP-серверу.
- Запись о методе аутентификации LDAP должна быть первой по списку в файле конфигурации `pg_hba.conf` на сервере Postgres Pro.
- Если аутентификация LDAP настроена с использованием параметра `ldapbinddn`, то соответствующий DN должен уже существовать на LDAP-сервере.
- Если аутентификация LDAP настроена с использованием параметра `ldapsearchfilter` для фильтрации пользователей путём проверки их членства в группах, то соответствующая группа должна уже существовать на LDAP-сервере.

Хотя некоторые LDAP-серверы (например, FreeIPA) по умолчанию разрешают анонимное связывание (anonymous binding) в режиме `search+bind`, другие (например, ALD Pro) этого не делают. В целях безопасности не рекомендуется разрешать анонимное связывание на любом LDAP-сервере.

Примечание

Postgres Pro поддерживает аутентификацию пользователей с помощью ALD Pro `2.4.0` или выше.

Примеры конфигурационных записей, приведённые ниже, совместимы с серверами LDAP FreeIPA и ALD Pro.

Пример конфигурации LDAP с использованием простого связывания (simple-bind):

```
host ... ldap ldapserver=ldap.example.net \
              ldapport=389 \
              ldapbasedn="cn=users,cn=accounts,dc=example,dc=net"
```

Пример конфигурации LDAP с использованием метода `search+bind`:

```
host ... ldap ldapserver=ldap.example.net \
              ldapport=389 \
              ldapbasedn="cn=users,cn=accounts,dc=example,dc=net" \
              ldapbinddn="uid=pgpro,cn=users,cn=accounts,dc=example,dc=net" \
              ldapbindpasswd="<pgpro password>" \
              ldapsearchattribute=uid
```

При запросе на подключение к базе данных от имени пользователя `someuser` Postgres Pro попытается связаться с LDAP-сервером от имени пользователя `pgpro` и выполнить поиск записи `(uid=someuser)` в указанном базовом DN. Если соответствующая запись найдена, сервер базы данных попытается связаться с LDAP, используя найденные данные и пароль, предоставленный клиентом. Если второе подключение будет успешным, доступ к базе данных будет предоставлен.

Пример конфигурации LDAP с использованием метода `search+bind`, где применяется параметр `ldapsearchfilter` для фильтрации пользователей, являющихся членами группы `pgpro_access`:

```
host ... ldap ldapserver=ldap.example.net \
              ldapport=389 \
              ldapbasedn="cn=users,cn=accounts,dc=example,dc=net" \
              ldapbinddn="uid=pgpro,cn=users,cn=accounts,dc=example,dc=net" \
              ldapbindpasswd="<pgpro password>" \
              ldapsearchfilter="(&(objectClass=person)(uid=$username)
(memberOf=cn=pgpro_access,cn=groups,cn=accounts,dc=example,dc=net))"
```

При запросе на подключение к базе данных от имени пользователя `someuser` Postgres Pro попытается связаться с LDAP-сервером от имени пользователя `pgpro` и выполнить поиск записи `(uid=someuser)` в указанном базовом DN в соответствии с заданным фильтром поиска. В этом примере пользователь должен быть членом группы `pgpro_access` согласно фильтру. Если соответствующая запись найдена, сервер базы данных попытается связаться с LDAP, используя найденные данные и пароль, предоставленный клиентом. Если второе подключение будет успешным, доступ к базе данных будет предоставлен.

19.11. Аутентификация RADIUS

Данный метод аутентификации работает сходным с методом `password` образом, за исключением того, что он использует RADIUS как метод проверки пароля. RADIUS используется только для подтверждения пары «имя пользователя/пароль». Поэтому пользователь должен уже существовать в базе данных до того, как для аутентификации будет использован RADIUS.

В ходе аутентификации RADIUS настроенному RADIUS-серверу посылается запрос доступа. Это сообщение типа `Authenticate Only` (Только аутентификация), которое включает в себя параметры `user name` (имя пользователя), `password` (зашифрованный пароль) и `NAS Identifier` (идентификатор NAS). Запрос зашифровывается с использованием общего с сервером секрета. RADIUS-сервер отвечает на этот запрос либо `Access Accept` (Доступ принят), либо `Access Reject` (Доступ отклонён). Система ведения учёта RADIUS не поддерживается.

Для данного метода можно указать адреса нескольких серверов RADIUS, тогда они будут перебираться по очереди. В случае получения от любого сервера отрицательного ответа произойдёт сбой аутентификации. Если ответ не будет получен, последует попытка подключения к следующему серверу в списке. Чтобы задать имена нескольких серверов, разделите их имена запятыми и заключите список в двойные кавычки. При этом все остальные параметры RADIUS должны указываться так же в списках через запятую, чтобы каждый сервер получил собственное значение. Возможно также задавать их единственным значением, тогда это значение будет применяться ко всем серверам.

Для метода RADIUS доступны следующие параметры конфигурации:

`radiusservers`

DNS-имена или IP-адреса целевых серверов RADIUS. Это обязательный параметр.

`radiussecrets`

Общие секреты, используемые при контактах с серверами RADIUS. Значение этого параметра должно быть одинаковым на серверах Postgres Pro и RADIUS. Рекомендуется использовать строку как минимум из 16 символов. Это обязательный параметр.

Примечание

Шифровальный вектор будет достаточно эффективен только в том случае, если Postgres Pro собран с поддержкой OpenSSL. В противном случае передача данных серверу RADIUS

> будет лишь замаскированной, но не защищённой, поэтому необходимо принять дополнительные меры безопасности.

`radiusports`

Номера портов для подключения к серверам RADIUS. Если порты не указываются, используется стандартный порт RADIUS (`1812`).

`radiusidentifiers`

Строки, используемые в запросах RADIUS как `NAS Identifier` (Идентификатор NAS). Этот параметр может использоваться, например, для обозначения кластера БД, к которому пытается подключаться пользователь, что позволяет выбрать соответствующую политику на сервере RADIUS. Если никакой идентификатор не задан, по умолчанию используется `postgresql`.

Если в значении параметра RADIUS необходимо передать запятую или пробел, это можно сделать, заключив это значение в двойные кавычки, хотя это может быть не очень удобно, так как потребуются два уровня двойных кавычек. Например, так добавляются пробелы в строки секретов RADIUS:

```
host ... radius radiusservers="server1,server2" radiussecrets="""secret one"",""secret
 two"""
```

19.12. Аутентификация по сертификату

Для аутентификации в рамках этого метода используется клиентский сертификат SSL, поэтому данный способ применим только для SSL-подключений (инструкции для конфигурации SSL находятся в Подразделе 17.9.2). Когда используется этот метод, сервер потребует от клиента предъявления действительного и доверенного сертификата. Пароль у клиента не запрашивается. Атрибут `cn` (Обычное имя) сертификата сравнивается с запрашиваемым именем пользователя базы данных, и если они соответствуют, вход разрешается. Если `cn` отличается от имени пользователя базы данных, то может быть использовано сопоставление имён пользователей.

Для аутентификации по SSL сертификату доступны следующие параметры конфигурации:

`map`

Позволяет сопоставить имена пользователей системы и базы данных. За подробностями обратитесь к Разделу 19.2.

Указывать параметр `clientcert` для режима аутентификации `cert` излишне, так как режим `cert` по сути реализуется как аутентификация `trust` со свойством `clientcert=verify-full`.

19.13. Аутентификация PAM

Данный метод аутентификации работает подобно методу `password`, но использует в качестве механизма проверки подлинности PAM (Pluggable Authentication Modules, Подключаемые модули аутентификации). По умолчанию имя службы PAM — `postgresql`. PAM используется только для проверки пар «имя пользователя/пароль» и может дополнительно проверять имя или IP-адрес удалённого компьютера. Поэтому пользователь должен уже существовать в базе данных, чтобы PAM можно было использовать для аутентификации. За дополнительной информацией о PAM обратитесь к *Странице описания Linux-PAM*.

Для аутентификации PAM доступны следующие параметры конфигурации:

`pamservice`

Имя службы PAM

`pam_use_hostname`

Указывает, предоставляется ли модулям PAM через поле `PAM_RHOST` IP-адрес либо имя удалённого компьютера. По умолчанию выдаётся IP-адрес. Установите в этом параметре 1, чтобы использовать имя узла. Разрешение имени узла может приводить к задержкам при подключении. (Обычно конфигурации PAM не задействуют эту информацию, так что этот параметр следует учитывать, только если создана специальная конфигурация, в которой он используется.)

Примечание

Если PAM настроен для чтения `/etc/shadow`, произойдёт сбой аутентификации, потому что сервер Postgres Pro запущен не пользователем root. Однако это не имеет значения, когда PAM настроен для использования LDAP или других методов аутентификации.

19.14. Аутентификация BSD

Данный метод аутентификации работает подобно методу `password`, но использует для проверки пароля механизм аутентификации BSD. Аутентификация BSD используется только для проверки пар «имя пользователя/пароль». Поэтому роль пользователя должна уже существовать в базе данных, чтобы эта аутентификация была успешной. Механизм аутентификации BSD в настоящее время может применяться только в OpenBSD.

Для аутентификации BSD в Postgres Pro применяется тип входа `auth-postgresql` и класс `postgresql`, если он определён в `login.conf`. По умолчанию этот класс входа не существует и Postgres Pro использует класс входа по умолчанию.

Примечание

Для использования аутентификации BSD необходимо сначала добавить учётную запись пользователя Postgres Pro (то есть, пользователя ОС, запускающего сервер) в группу `auth`. Группа `auth` существует в системах OpenBSD по умолчанию.

19.15. Проблемы аутентификации

Сбои и другие проблемы с аутентификацией обычно дают о себе знать через сообщения об ошибках, например:

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database
 "testdb"
```

Это сообщение вы, скорее всего, получите, если сможете связаться с сервером, но он не захочет с вами общаться. В сообщении содержится предположение, что сервер отказывает вам в подключении, поскольку не может найти подходящую запись в файле `pg_hba.conf`.

```
FATAL:  password authentication failed for user "andym"
```

Такое сообщение показывает, что вы связались с сервером, он готов общаться с вами, но только после того, как вы прошли авторизацию по методу, указанному в файле `pg_hba.conf`. Проверьте пароль, который вы вводите, и как настроен Kerberos или ident, если в сообщении упоминается один из этих типов аутентификации.

```
FATAL:  user "andym" does not exist
```

Указанное имя пользователя базы данных не найдено.

```
FATAL:  database "testdb" does not exist
```

База данных, к которой вы пытаетесь подключиться, не существует. Имейте в виду, что если не указано имя базы данных, по умолчанию берётся имя пользователя базы данных.

Подсказка

В журнале сервера может содержаться больше информации, чем в выдаваемых клиенту сообщениях об ошибке аутентификации, поэтому, если вас интересуют причины сбоя, проверьте журнал сервера.

Глава 20. Роли базы данных

Postgres Pro использует концепцию ролей (*roles*) для управления разрешениями на доступ к базе данных. Роль можно рассматривать как пользователя базы данных или как группу пользователей, в зависимости от того, как роль настроена. Роли могут владеть объектами базы данных (например, таблицами и функциями) и выдавать другим ролям разрешения на доступ к этим объектам, управляя тем, кто имеет доступ и к каким объектам. Кроме того, можно предоставить одной роли *членство* в другой роли, таким образом одна роль может использовать права других ролей.

Концепция ролей включает в себя концепцию пользователей («users») и групп («groups»). До версии 8.1 в PostgreSQL пользователи и группы были отдельными сущностями, но теперь есть только роли. Любая роль может использоваться в качестве пользователя, группы, и того и другого.

В этой главе описывается, как создавать роли и управлять ими. Дополнительную информацию о правах доступа ролей к различным объектам баз данных можно найти в Разделе 5.8.

20.1. Роли базы данных

Роли базы данных концептуально полностью отличаются от пользователей операционной системы. На практике поддержание соответствия между ними может быть удобным, но не является обязательным. Роли базы данных являются глобальными для всей установки кластера базы данных (не для отдельной базы данных). Для создания роли используется команда SQL `CREATE ROLE`:

```
CREATE ROLE имя;
```

Здесь *`имя`* соответствует правилам именования идентификаторов SQL: либо обычное, без специальных символов, либо в двойных кавычках. (На практике, к команде обычно добавляются другие указания, такие как `LOGIN`. Подробнее об этом ниже.) Для удаления роли используется команда `DROP ROLE`:

```
DROP ROLE имя;
```

Для удобства поставляются программы createuser и dropuser, которые являются обёртками для этих команд SQL и вызываются из командной строки оболочки ОС:

```
createuser имя
dropuser имя
```

Чтобы получить список существующих ролей, обратитесь к представлению `pg_roles`, например:

```
SELECT rolname FROM pg_roles;
```

Чтобы проверить, какие роли могут подключаться к базе данных, выполните:

```
SELECT rolname FROM pg_roles WHERE rolcanlogin;
```

Метакоманда `\du` программы psql также полезна для получения списка существующих ролей.

Для начальной настройки кластера базы данных система сразу после инициализации всегда содержит одну предопределённую роль с возможностью подключения. Эта роль всегда является суперпользователем («superuser») и по умолчанию имеет такое же имя, как и пользователь операционной системы, инициализирующий кластер баз данных командой `initdb`, если при её запуске не указано другое имя. Обычно, но необязательно, эта роль называется `postgres`. Для создания других ролей вначале нужно подключиться с этой ролью.

Каждое подключение к серверу базы данных выполняется под именем конкретной роли, и эта роль определяет начальные права доступа для команд, выполняемых в этом соединении. Имя роли для конкретного подключения к базе данных указывается клиентской программой характерным для неё способом, таким образом инициируя запрос на подключение. Например, программа `psql` для указания роли использует аргумент командной строки `-U`. Многие приложения предполагают, что по умолчанию нужно использовать имя пользователя операционной системы (включая `createuser` и `psql`). Поэтому часто бывает удобным поддерживать соответствие между именами ролей и именами пользователей операционной системы.

Список доступных для подключения ролей, который могут использовать клиенты, определяется конфигурацией аутентификации, как описывалось в Главе 19. (Поэтому клиент не ограничен только ролью, соответствующей имени пользователя операционной системы, так же как и имя для входа может не соответствовать реальному имени.) Так как с ролью связан набор прав, доступных для клиента, в многопользовательской среде распределять права следует с осторожностью.

20.2. Атрибуты ролей

Роль базы данных может иметь атрибуты, определяющие её полномочия и взаимодействие с системой аутентификации клиентов.

Право подключения

Только роли с атрибутом `LOGIN` могут использоваться для начального подключения к базе данных. Роль с атрибутом `LOGIN` можно рассматривать как пользователя базы данных. Для создания такой роли можно использовать любой из вариантов:

```
CREATE ROLE имя LOGIN;
CREATE USER имя;
```

(Команда `CREATE USER` эквивалентна `CREATE ROLE` за исключением того, что `CREATE USER` по умолчанию включает атрибут `LOGIN`, в то время как `CREATE ROLE` — нет.)

Статус суперпользователя

Суперпользователь базы данных обходит все проверки прав доступа, за исключением права на вход в систему. Это опасная привилегия и она не должна использоваться небрежно. Лучше всего выполнять большую часть работы не как суперпользователь. Для создания нового суперпользователя используется `CREATE ROLE имя SUPERUSER`. Это нужно выполнить из под роли, которая также является суперпользователем.

Создание базы данных

Роль должна явно иметь разрешение на создание базы данных (за исключением суперпользователей, которые пропускают все проверки). Для создания такой роли используется `CREATE ROLE имя CREATEDB`.

Создание роли

Роль должна явно иметь разрешение на создание других ролей (за исключением суперпользователей, которые пропускают все проверки). Для создания такой роли используется `CREATE ROLE имя CREATEROLE`. Роль с правом `CREATEROLE` может изменять и удалять роли, которые были назначены пользователю с правами `CREATEROLE` и `ADMIN OPTION`. Такое назначение ролей происходит автоматически, когда пользователь с правом `CREATEROLE`, не являющийся суперпользователем, создаёт новую роль. Поэтому по умолчанию пользователь с правом `CREATEROLE` может изменять и удалять созданные им роли. Изменение роли включает большинство действий команды `ALTER ROLE`, например смену пароля, а также действия команд `COMMENT` и `SECURITY LABEL` в отношении ролей.

Однако, имея право `CREATEROLE`, нельзя ни создавать, ни каким-либо образом влиять на роли `SUPERUSER`. Кроме того, с правом `CREATEROLE` нельзя ни создавать пользователей `REPLICATION`, ни предоставлять или отзывать право `REPLICATION`, ни изменять свойства ролей таких пользователей. Тем не менее, это право позволяет использовать команды `ALTER ROLE ... SET` и `ALTER ROLE ... RENAME` в отношении ролей `REPLICATION`, а также `COMMENT ON ROLE`, `SECURITY LABEL ON ROLE` и `DROP ROLE`. Наконец, право `CREATEROLE` не даёт возможности предоставлять или отзывать право `BYPASSRLS`.

Запуск репликации

Роль должна иметь явное разрешение на запуск потоковой репликации (за исключением суперпользователей, которые пропускают все проверки). Роль, используемая для потоковой репликации, также должна иметь атрибут `LOGIN`. Для создания такой роли используется `CREATE ROLE *имя* REPLICATION LOGIN`.

Пароль

Пароль имеет значение, если метод аутентификации клиентов требует, чтобы пользователи предоставляли пароль при подключении к базе данных. Методы аутентификации `password` и `md5` используют пароли. База данных и операционная система используют раздельные пароли. Пароль указывается при создании роли: `CREATE ROLE *имя* PASSWORD '*строка*'`.

Наследование прав

По умолчанию роль наследует права ролей, членом которых она является. Однако можно создать роль, которая не будет наследовать права по умолчанию. Для этого выполните `CREATE ROLE *имя* NOINHERIT`. Как вариант, атрибуты наследования отдельных прав можно переопределить, указав `WITH INHERIT TRUE` или `WITH INHERIT FALSE`.

Игнорирование защиты на уровне строк

Разрешение обходить все политики защиты на уровне строк (RLS) нужно давать роли явно. Создать роль с таким разрешением можно, выполнив `CREATE ROLE *имя* BYPASSRLS` от имени суперпользователя.

Ограничение соединений

Для роли можно ограничить число одновременных соединений. -1 (по умолчанию) означает отсутствие ограничения. Для создания роли с ограничением используется `CREATE ROLE *name* CONNECTION LIMIT '*число*'`.

Атрибуты ролей могут быть изменены после создания командой `ALTER ROLE`. Более детальная информация в справке по командам CREATE ROLE и ALTER ROLE.

На уровне ролей можно устанавливать многие конфигурационные параметры времени выполнения, описанные в Главе 18. Например, если по некоторым причинам всякий раз при подключении к базе данных требуется отключить использование индексов (подсказка: плохая идея), можно выполнить:

```
ALTER ROLE myname SET enable_indexscan TO off;
```

Установленное значение параметра будет сохранено (но не будет применено сразу). Для последующих подключений с этой ролью это будет выглядеть как выполнение команды `SET enable_indexscan TO off` перед началом сеанса. Но это только значение по умолчанию; в течение сеанса этот параметр можно изменить. Для удаления установок на уровне ролей для параметров конфигурации используется `ALTER ROLE *имя_роли* RESET *имя_переменной*`. Обратите внимание, что установка параметров конфигурации на уровне роли без права `LOGIN` лишено смысла, т. к. они никогда не будут применены.

Когда не суперпользователь с правом `CREATEROLE` создаёт роль, создающая роль автоматически получает членство в создаваемой, как если бы начальный суперпользователь выполнил команду `GRANT created_user TO creating_user WITH ADMIN TRUE, SET FALSE, INHERIT FALSE`. Поскольку пользователь с правом `CREATEROLE` может пользоваться специальными правами в отношении существующей роли только при наличии атрибута `ADMIN OPTION`, такого права вполне достаточно для того, чтобы пользователь с правом `CREATEROLE` мог администрировать созданную им роль. Тем не менее, поскольку роль создаётся с атрибутами `INHERIT FALSE, SET FALSE`, пользователь с правом `CREATEROLE` не будет наследовать права созданной роли и не сможет переключиться на неё с помощью команды `SET ROLE`. Однако поскольку любой пользователь с правом `ADMIN OPTION` для роли может разрешать членство в этой роли любым пользователям, пользователь с правом `CREATEROLE` может стать членом созданной им роли, просто назначив эту роль на себя при помощи `INHERIT` и/или `SET`. Поэтому факт того, что права не наследуются по умолчанию и что право не назначается по умолчанию командой `SET ROLE`, — это способ оградиться от случайных ошибок, а не средство защиты. Кроме того, обратите внимание, что это право автоматически выдаётся от имени начального суперпользователя, поэтому пользователь с правом `CREATEROLE` не сможет удалить или изменить его. Тем не менее любой суперпользователь может отозвать, модифицировать и/или выдать дополнительно такие права другим пользователям с правом `CREATEROLE`. Любой пользователь с правами `CREATEROLE` и `ADMIN OPTION` в этой роли может её администрировать.

20.3. Членство в роли

Часто бывает удобно сгруппировать пользователей для упрощения управления правами: права можно выдавать для всей группы и у всей группы забирать. В Postgres Pro для этого создаётся роль, представляющая группу, а затем *членство* в этой группе выдаётся ролям индивидуальных пользователей.

Для настройки групповой роли сначала нужно создать саму роль:

```
CREATE ROLE имя;
```

Обычно групповая роль не имеет атрибута `LOGIN`, хотя при желании его можно установить.

После того как групповая роль создана, в неё можно добавлять или удалять членов, используя команды `GRANT` и `REVOKE`:

```
GRANT групповая_роль TO роль1, ... ;
REVOKE групповая_роль FROM роль1, ... ;
```

Членом роли может быть и другая групповая роль (потому что в действительности нет никаких различий между групповыми и не групповыми ролями). При этом база данных не допускает замыкания членства по кругу. Также не допускается управление членством роли `PUBLIC` в других ролях.

Члены групповой роли могут использовать её права двумя способами. Во-первых, члены группы, которым было предоставлено членство с атрибутом `SET`, могут выполнить команду `SET ROLE`, чтобы временно «стать» групповой ролью. В этом состоянии сеанс базы данных имеет доступ к правам групповой роли вместо исходной роли, от имени которой был выполнен вход в систему. При этом для всех создаваемых объектов базы данных владельцем считается групповая, а не исходная роль. Во-вторых, члены группы, которым было предоставлено членство с атрибутом `INHERIT`, автоматически используют права ролей, членами которых они являются прямо или косвенно, вплоть до роли без атрибута `INHERIT`. Например:

```
CREATE ROLE joe LOGIN;
CREATE ROLE admin;
CREATE ROLE wheel;
CREATE ROLE island;
GRANT admin TO joe WITH INHERIT TRUE;
GRANT wheel TO admin WITH INHERIT FALSE;
GRANT island TO joe WITH INHERIT TRUE, SET FALSE;
```

После подключения с ролью `joe` сеанс базы данных будет использовать права, выданные напрямую `joe`, и права, выданные ролям `admin` и `island`, так как `joe` «наследует» эти права. Однако права, выданные `wheel`, будут недоступны, потому что, хотя `joe` косвенно является членом `wheel`, это членство получено через роль `admin`, которая была выдана с атрибутом `WITH INHERIT FALSE`. После выполнения команды:

```
SET ROLE admin;
```

сеанс будет использовать только права, выданные `admin`, а не права, выданные роли `joe` или `island`. После выполнения команды:

```
SET ROLE wheel;
```

сеанс будет использовать только права, выданные `wheel`, а не права, выданные `joe` или `admin`. Начальный набор прав можно восстановить любой из команд:

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

Примечание

Команда `SET ROLE` в любой момент разрешает выбрать любую роль, прямым или косвенным членом которой является оригинальная роль, под которой был выполнен вход в систему, при условии, что существует цепочка членства в ролях, каждая из которых имеет атрибут `SET TRUE` (установленный по умолчанию). Поэтому в примере выше не обязательно сначала становиться `admin` перед тем как стать `wheel`. С другой стороны, стать `island` не получится вовсе, а `joe` может получить доступ к правам только через механизм наследования.

Примечание

В стандарте SQL есть чёткое различие между пользователями и ролями. При этом пользователи, в отличие от ролей, не наследуют права автоматически. Такое поведение может быть получено в Postgres Pro, если для ролей, используемых как роли в стандарте SQL, устанавливать атрибут `INHERIT`, а для ролей-пользователей в стандарте SQL — атрибут `NOINHERIT`. Однако в Postgres Pro все роли по умолчанию имеют атрибут `INHERIT`. Это сделано для обратной совместимости с версиями до 8.1, в которых пользователи всегда могли использовать права групп, членами которых они являются.

Атрибуты роли `LOGIN`, `SUPERUSER`, `CREATEDB` и `CREATEROLE` можно рассматривать как особые права, но они никогда не наследуются как обычные права на объекты базы данных. Чтобы ими воспользоваться, необходимо переключиться на роль, имеющую этот атрибут, с помощью команды `SET ROLE`. Продолжая предыдущий пример, можно установить атрибуты `CREATEDB` и `CREATEROLE` для роли `admin`. Затем при входе с ролью `joe`, получить доступ к этим правам будет возможно только после выполнения `SET ROLE admin`.

Для удаления групповой роли используется `DROP ROLE`:

```
DROP ROLE имя;
```

Любое членство в групповой роли будет автоматически отозвано (в остальном на членов этой роли это никак не повлияет).

20.4. Удаление ролей

Так как роли могут владеть объектами баз данных и иметь права доступа к объектам других, удаление роли не сводится к немедленному действию `DROP ROLE`. Сначала должны быть удалены и переданы другим владельцами все объекты, принадлежащие роли; также должны быть отозваны все права, данные роли.

Владение объектами можно передавать в индивидуальном порядке, применяя команду `ALTER`, например:

```
ALTER TABLE bobs_table OWNER TO alice;
```

Кроме того, для переназначения какой-либо другой роли владения сразу всеми объектами, принадлежащими удаляемой роли, можно применить команду `REASSIGN OWNED`. Так как `REASSIGN OWNED` не может обращаться к объектам в других базах данных, её необходимо выполнить в каждой базе, которая содержит объекты, принадлежащие этой роли. (Заметьте, что первая такая команда `REASSIGN OWNED` изменит владельца для всех разделяемых между базами объектов, то есть для баз данных или табличных пространств, принадлежащих удаляемой роли.)

После того как все ценные объекты будут переданы новым владельцам, все оставшиеся объекты, принадлежащие удаляемой роли, могут быть удалены с помощью команды `DROP OWNED`. И эта команда не может обращаться к объектам в других базах данных, так что её нужно запускать в каждой базе, которая содержит объекты, принадлежащие роли. Также заметьте, что `DROP OWNED` не удаляет табличные пространства или базы данных целиком, так что это необходимо сделать вручную, если роли принадлежат базы или табличные пространства, не переданные новым владельцам.

`DROP OWNED` также удаляет все права, которые даны целевой роли для объектов, не принадлежащих ей. Так как `REASSIGN OWNED` такие объекты не затрагивает, обычно необходимо запустить и `REASSIGN OWNED`, и `DROP OWNED` (в этом порядке!), чтобы полностью ликвидировать зависимости удаляемой роли.

С учётом этого, общий рецепт удаления роли, которая владела объектами, вкратце таков:

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;
-- повторить предыдущие команды для каждой базы в кластере
DROP ROLE doomed_role;
```

Когда не все объекты нужно передать одному новому владельцу, лучше сначала вручную отработать исключения, а в завершение выполнить показанные выше действия.

При попытке выполнить `DROP ROLE` для роли, у которой сохраняются зависимые объекты, будут выданы сообщения, говорящие, какие объекты нужно передать другому владельцу или удалить.

20.5. Предопределённые роли

В Postgres Pro имеется набор предопределённых ролей, которые дают доступ к некоторым часто востребованным, но не общедоступным функциям и данным. Администраторы (включая роли с правами `CREATEROLE`) могут назначать (`GRANT`) эти роли пользователям и/или ролям в своей среде, таким образом открывая этим пользователям доступ к указанной функциональности и информации.

Имеющиеся предопределённые роли описаны в Таблица 20.1. Заметьте, что конкретные разрешения для каждой из таких ролей в будущем могут изменяться по мере добавления дополнительной функциональности. Администраторы должны следить за этими изменениями, просматривая замечания к выпускам.

Таблица 20.1. Предопределённые роли

Роль	Разрешаемый доступ
pg_read_all_data	Читать все данные (таблицы, представления, последовательности), как будто роль имеет права `SELECT` на эти объекты и права USAGE на все схемы, но при этом явным образом такие права ей не назначены. Атрибут `BYPASSRLS` для этой роли не установлен. Если используется политика RLS, администратор может задать атрибут `BYPASSRLS` членам этой роли.
pg_write_all_data	Записывать все данные (таблицы, представления, последовательности), как будто роль имеет права `INSERT`, `UPDATE` и `DELETE` на эти объекты и права USAGE на все схемы, но при этом явным образом такие права ей не назначены. Атрибут `BYPASSRLS` для этой роли не установлен. Если используется политика RLS, администратор может задать атрибут `BYPASSRLS` членам этой роли.
pg_read_all_settings	Читать все конфигурационные переменные, даже те, что обычно видны только суперпользователям.
pg_read_all_stats	Читать все представления pg_stat_* и использовать различные расширения, связанные со статистикой, даже те, что обычно видны только суперпользователям.
pg_stat_scan_tables	Выполнять функции мониторинга, которые могут устанавливать блокировки `ACCESS SHARE` в таблицах, возможно, на длительное время.

Роль	**Разрешаемый доступ**
pg_monitor	Читать/выполнять различные представления и функции для мониторинга. Эта роль включена в роли `pg_read_all_settings` , `pg_read_all_stats` и `pg_stat_scan_tables` .
pg_database_owner	Никакого. Неявным образом включает в себя владельца текущей базы данных.
pg_signal_backend	Передавать другим обслуживающим процессам сигналы для отмены запроса или завершения сеанса.
pg_read_server_files	Читать файлы в любом месте файловой системы, куда имеет доступ СУБД на сервере, выполняя COPY и другие функции работы с файлами.
pg_write_server_files	Записывать файлы в любом месте файловой системы, куда имеет доступ СУБД на сервере, выполняя COPY и другие функции работы с файлами.
pg_execute_server_program	Выполнять программы на сервере (от имени пользователя, запускающего СУБД) так же, как это делает команда COPY и другие функции, выполняющие программы на стороне сервера.
pg_checkpoint	Выполнять команду `CHECKPOINT`.
pg_maintain	Выполнять команды `VACUUM`, `ANALYZE`, `CLUSTER`, `REFRESH MATERIALIZED VIEW`, `REINDEX` и `LOCK TABLE` для всех отношений, как будто роль имеет права `MAINTAIN` на эти объекты, но при этом явным образом такие права ей не назначены.
pg_use_reserved_connections	Использовать слоты подключения, зарезервированные при помощи параметра reserved_connections.
pg_create_subscription	Разрешить пользователям с правом `CREATE` для базы данных выполнять команду `CREATE SUBSCRIPTION`.
pg_create_tablespace	Выполнять команду `CREATE TABLESPACE` без прав суперпользователя.
pg_manage_profiles	Выполнять команды `CREATE PROFILE`, `ALTER PROFILE` и `DROP PROFILE` без прав суперпользователя.

Роли `pg_monitor`, `pg_read_all_settings`, `pg_read_all_stats` и `pg_stat_scan_tables` созданы для того, чтобы администраторы могли легко настроить роль для мониторинга сервера БД. Эти роли наделяют своих членов набором общих прав, позволяющих читать различные полезные параметры конфигурации, статистику и другую системную информацию, что обычно доступно только суперпользователям.

Единственным членом роли `pg_database_owner` неявным образом является владелец текущей базы данных. Как и любая другая роль, она может владеть объектами или получать права доступа. Следовательно, права, данные роли `pg_database_owner` в базе-шаблоне, приобретёт владелец каждой базы, созданной из данного шаблона. Роль `pg_database_owner` не может быть членом какой-либо роли, и в неё не могут быть явно включены члены. Изначально эта роль владеет схемой `public`, то есть владелец базы данных управляет использованием данной схемы в своей базе.

Роль `pg_signal_backend` предназначена для того, чтобы администраторы могли давать доверенным, но не имеющим прав суперпользователя ролям право посылать сигналы другим обслуживающим процессам. В настоящее время эта роль позволяет передавать сигналы для отмены запроса, выполняющегося в другом процессе, или для завершения сеанса. Однако пользователь, включённый в эту роль, не может отправлять сигналы процессам, принадлежащим суперпользователям. См. Подраздел 9.28.2.

Роли `pg_read_server_files`, `pg_write_server_files` и `pg_execute_server_program` предназначены для того, чтобы администраторы могли выделить доверенные, но не имеющие права суперпользователей роли для доступа к файлам и запуска программ на сервере БД от имени пользователя, запускающего СУБД. Так как эти роли могут напрямую обращаться к любым файлам в файловой системе сервера, они обходят все проверки разрешений на уровне базы данных, а значит, воспользовавшись ими, можно получить права суперпользователя. Поэтому назначать их пользователям следует со всей осторожностью.

Управлять членством в этих ролях следует осмотрительно, чтобы они использовались только по необходимости и только с пониманием, что они открывают доступ к закрытой информации.

Администраторы могут давать пользователям доступ к этим ролям, используя команду `GRANT`. Например:

```
GRANT pg_signal_backend TO admin_user;
```

20.6. Безопасность функций

Функции, триггеры и политики защиты на уровне строк позволяют пользователям внедрять код в обслуживающие процессы, который может быть непреднамеренно выполнен другими пользователями. Таким образом эти механизмы позволяют пользователям запускать «троянский код» относительно просто. Лучшая защита от этого — строгое ограничение круга лиц, которые могут создавать объекты. Там где это невозможно, пишите запросы так, чтобы они ссылались только на объекты с доверенными владельцами. Удалите из `search_path` любые схемы, в которых могут создавать объекты недоверенные пользователи.

Функции выполняются внутри серверного процесса с полномочиями пользователя операционной системы, запускающего сервер базы данных. Если используемый для функций язык программирования разрешает неконтролируемый доступ к памяти, то это даёт возможность изменить внутренние структуры данных сервера. Таким образом, помимо всего прочего, такие функции могут обойти ограничения доступа к системе. Языки программирования, допускающие такой доступ, считаются «недоверенными» и создавать функции на этих языках Postgres Pro разрешает только суперпользователям.

Глава 21. Управление базами данных

Каждый работающий экземпляр сервера Postgres Pro обслуживает одну или несколько баз данных. Поэтому базы данных представляют собой вершину иерархии SQL-объектов («объектов базы данных»). Данная глава описывает свойства баз данных, процессы создания, управления и удаления.

21.1. Обзор

Некоторые объекты, включая роли, базы данных и табличные пространства, определяются на уровне кластера и сохраняются в табличном пространстве `pg_global`. Внутри кластера существуют базы данных, которые отделены друг от друга, но могут обращаться к объектам уровня кластера. Внутри каждой базы данных имеются схемы, содержащие такие объекты, как таблицы и функции. Таким образом, полная иерархия выглядит следующим образом: кластер, база данных, схема, таблица (или иной объект, например функция).

При подключении к серверу баз данных клиент должен указать имя базы в запросе подключения. Обращаться к нескольким базам через одно подключение нельзя, однако клиенты могут открыть несколько подключений к одной базе или к разным. Безопасность на уровне базы обеспечивают две составляющие: управление подключениями (см. Раздел 19.1), которое осуществляется на уровне соединения, и управление доступом к объектам (см. Раздел 5.8), для которого реализована система прав. Обёртки сторонних данных (см. postgres_fdw) позволяют создать в одной базе данных объекты, скрывающие за собой объекты в других базах или кластерах. Подобную же функциональность предоставляет и более старый модуль dblink (см. dblink). По умолчанию все пользователи могут подключаться ко всем базам данных, используя все методы подключения.

В случаях, когда в кластере Postgres Pro планируется размещать данные несвязанных проектов или с ним будут работать пользователи, которые в принципе не должны никак взаимодействовать, рекомендуется использовать отдельные базы данных и организовать управление подключением и доступом к объектам соответствующим образом. Если же проекты или пользователи взаимосвязаны и должны иметь возможность использовать ресурсы друг друга, они должны размещаться в одной базе данных, но, возможно, в отдельных схемах. Таким образом будет создана модульная структура с изолированными пространствами имён и управлением доступа. Подробнее об управлении схемами рассказывается в Разделе 5.10.

Хотя в одном кластере можно создать несколько баз данных, прежде чем это делать, рекомендуется тщательно взвесить все связанные с этим риски и ограничения. В частности, наличие общего WAL (см. Главу 27) может повлиять на возможности резервного копирования и восстановления данных. Отдельные базы в кластере изолированы друг от друга с точки зрения пользователя, но с точки зрения администратора баз данных они тесно связаны.

Базы данных создаются командой `CREATE DATABASE` (см. Раздел 21.2), а удаляются командой `DROP DATABASE` (см. Раздел 21.5). Список существующих баз данных можно посмотреть в системном каталоге `pg_database`, например,

```
SELECT datname FROM pg_database;
```

Метакоманда `\l` или ключ `-l` командной строки приложения psql также позволяют вывести список существующих баз данных.

> **Примечание**
>
> Стандарт SQL называет базы данных «каталогами», но на практике у них нет отличий.

21.2. Создание базы данных

Для создания базы данных сервер Postgres Pro должен быть развёрнут и запущен (см. Раздел 17.3).

База данных создаётся SQL-командой CREATE DATABASE:

```
CREATE DATABASE имя;
```

где *имя* подчиняется правилам именования идентификаторов SQL. Текущий пользователь автоматически назначается владельцем. Владелец может удалить свою базу, что также приведёт к удалению всех её объектов, в том числе, имеющих других владельцев.

Создание баз данных это привилегированная операция. Как предоставить права доступа, описано в Разделе 20.2.

Поскольку для выполнения команды `CREATE DATABASE` необходимо подключение к серверу базы данных, возникает вопрос как создать самую первую базу данных. Первая база данных всегда создаётся командой `initdb` при инициализации пространства хранения данных (см. Раздел 17.2.) Эта база данных называется `postgres`. Далее для создания первой «обычной» базы данных можно подключиться к `postgres`.

Две дополнительные базы данных `template1`и `template0`также создаются во время инициализации кластера. При каждом создании новой базы данных в рамках кластера по факту производится клонирование шаблона `template1`. При этом любые изменения, сделанные в `template1`, распространяются на все созданные впоследствии базы данных. Следует избегать создания объектов в `template1`, за исключением ситуации, когда их необходимо автоматически добавлять в новые базы. База `template0` представляет собой чистую копию исходного содержимого `template1`. Поэтому её можно клонировать вместо `template1`, когда важно создать базу данных без каких-либо локальных дополнений. Подробнее об этом рассказывается в Разделе 21.3.

Для удобства, есть утилита командной строки для создания баз данных, `createdb`.

```
createdb dbname
```

Утилита `createdb` не делает ничего волшебного, она просто подключается к базе данных `postgres` и выполняет ранее описанную SQL-команду `CREATE DATABASE`. Подробнее о её вызове можно узнать в createdb. Обратите внимание, что команда `createdb` без параметров создаст базу данных с именем текущего пользователя.

Примечание

Глава 19 содержит информацию о том, как ограничить права на подключение к заданной базе данных.

Иногда необходимо создать базу данных для другого пользователя и назначить его владельцем, чтобы он мог конфигурировать и управлять ею. Для этого используйте одну из следующих команд:

```
CREATE DATABASE имя_базы OWNER имя_роли;
```

из среды SQL, или:

```
createdb -O имя_роли имя_базы
```

из командной строки ОС. Лишь суперпользователь может создавать базы данных для других (для ролей, членом которых он не является).

21.3. Шаблоны баз данных

По факту команда `CREATE DATABASE` выполняет копирование существующей базы данных. По умолчанию копируется стандартная системная база `template1`. Таким образом, `template1` это шаблон, на основе которого создаются новые базы. Если добавить объекты в `template1`, то впоследствии они будут копироваться в новые базы данных. Это позволяет внести изменения в стандартный набор объектов. Например, если в `template1` установить процедурный язык PL/Perl, то он будет доступен в новых базах без дополнительных действий.

Однако команда `CREATE DATABASE` не копирует права `GRANT` уровня базы из исходной БД. Новая база получает права уровня базы по умолчанию.

Также существует вторая системная база `template0`. При инициализации она содержит те же самые объекты, что и `template1`, предопределённые в рамках устанавливаемой версии Postgres Pro. В `template0` не следует вносить никакие изменения после инициализации кластера. Если в команде `CREATE DATABASE` указать в качестве шаблона `template0` вместо `template1`, вы сможете получить «чистую» пользовательскую базу данных (в которой никаких пользовательских объектов нет, есть только системные объекты в первозданном виде), не содержащую ничего, что могло быть добавлено на месте в `template1`. Это особенно полезно при восстановлении дампа `pg_dump`: скрипт выгруженного дампа должен восстанавливаться в чистую базу, чтобы он мог воссоздать нужное содержимое базы, избежав конфликтов с объектами, которые могли быть добавлены в `template1`.

Другая причина, для копирования `template0` вместо `template1` заключается в том, что можно указать новые параметры локали и кодировку при копировании `template0`, в то время как для копий `template1` они не должны меняться. Это связано с тем, что `template1` может содержать данные в специфических кодировках и локалях, в отличие от `template0`.

Для создания базы данных на основе `template0`, используйте:

```
CREATE DATABASE dbname TEMPLATE template0;
```

из среды SQL, или:

```
createdb -T template0 dbname
```

из командной строки ОС.

Можно создавать дополнительные шаблоны баз данных, и, более того, можно копировать любую базу данных кластера, если указать её имя в качестве шаблона в команде `CREATE DATABASE`. Важно понимать, что это (пока) не рассматривается в качестве основного инструмента для реализации возможности «`COPY DATABASE`». Важным является то, что при копировании все сеансы к копируемой базе данных, должны быть закрыты. `CREATE DATABASE` выдаст ошибку, если есть другие подключения; во время операции копирования новые подключения к этой базе данных не разрешены.

В таблице `pg_database`есть два полезных флага для каждой базы данных: столбцы `datistemplate` и `datallowconn`. `datistemplate` указывает на факт того, что база данных может выступать в качестве шаблона в команде `CREATE DATABASE`. Если флаг установлен, то для пользователей с правом `CREATEDB` клонирование доступно; если флаг не установлен, то лишь суперпользователь и владелец базы данных могут её клонировать. Если `datallowconn` не установлен, то новые подключения к этой базе не допустимы (однако текущие сеансы не закрываются при сбросе этого флага). База `template0` обычно помечена как `datallowconn = false` для избежания любых её модификаций. И `template0`, и `template1` всегда должны быть помечены флагом `datistemplate = true`.

Примечание

`template1` и `template0` не выделены как-то особенно, кроме того факта, что `template1` используется по умолчанию в команде `CREATE DATABASE`. Например, можно удалить `template1` и безболезненно создать заново из `template0`. Это можно посоветовать в случае, если `template1` был замусорен. (Чтобы удалить `template1`, необходимо сбросить флаг `pg_database.datistemplate = false`.)

База данных `postgres` также создаётся при инициализации кластера. Она используется пользователями и приложениями для подключения по умолчанию. Представляет собой всего лишь копию `template1`, и может быть удалена и повторно создана при необходимости.

21.4. Конфигурирование баз данных

Обратившись к Главе 18 можно выяснить, что сервер Postgres Pro имеет множество параметров конфигурации времени выполнения. Можно выставить специфичные для базы данных значения по умолчанию.

Например, если по какой-то причине необходимо выключить GEQO оптимизатор в какой-то из баз, то можно, либо выключить его для всех баз данных одновременно, либо убедиться, что все клиенты заботятся об этом, выполняя команду `SET geqo TO off`. Для того чтобы это действовало по умолчанию в конкретной базе данных, необходимо выполнить команду:

```
ALTER DATABASE mydb SET geqo TO off;
```

Установка сохраняется, но не применяется тотчас. В последующих подключениях к этой базе данных, эффект будет таким, будто перед началом сеанса была выполнена команда `SET geqo TO off;`. Стоит обратить внимание, что пользователь по-прежнему может изменять этот параметр во время сеанса; ведь это просто значение по умолчанию. Чтобы сбросить такое установленное значение, используйте `ALTER DATABASE` *`dbname`* `RESET` *`varname`*.

21.5. Удаление базы данных

Базы данных удаляются командой DROP DATABASE:

```
DROP DATABASE имя;
```

Лишь владелец базы данных или суперпользователь могут удалить базу. При удалении также удаляются все её объекты. Удаление базы данных это необратимая операция.

Невозможно выполнить команду `DROP DATABASE` пока существует хоть одно подключение к заданной базе. Однако можно подключиться к любой другой, в том числе и `template1`. `template1` может быть единственной возможностью при удалении последней пользовательской базы данных кластера.

Также существует утилита командной строки для удаления баз данных dropdb:

```
dropdb dbname
```

(В отличие от команды `createdb` утилита не использует имя текущего пользователя по умолчанию).

21.6. Табличные пространства

Табличные пространства в Postgres Pro позволяют администраторам организовать логику размещения файлов объектов базы данных в файловой системе. К однажды созданному табличному пространству можно обращаться по имени на этапе создания объектов.

Табличные пространства позволяют администратору управлять дисковым пространством для инсталляции Postgres Pro. Это полезно минимум по двум причинам. Во-первых, это нехватка места в разделе, на котором был инициализирован кластер и невозможность его расширения. Табличное пространство можно создать в другом разделе и использовать его до тех пор, пока не появится возможность переконфигурирования системы.

Во-вторых, табличные пространства позволяют администраторам оптимизировать производительность согласно бизнес-процессам, связанным с объектами базы данных. Например, часто используемый индекс можно разместить на очень быстром и надёжном, но дорогом SSD-диске. В то же время таблица с архивными данными, которые редко используются и скорость к доступа к ним не важна, может быть размещена в более дешёвом и медленном хранилище.

Предупреждение

Несмотря на внешнее размещение относительно основного каталога хранения данных Postgres Pro, табличные пространства являются неотъемлемой частью кластера и *не могут* трактоваться, как самостоятельная коллекция файлов данных. Они зависят от метаданных, расположенных в главном каталоге, и потому не могут быть подключены к другому кластеру, или копироваться по отдельности. Также, в случае потери табличного пространства (при удалении файлов, сбое диска и т. п.), кластер может оказаться недоступным или не сможет запуститься. Таким образом, при размещении табличного пространства во временной файловой системе, например, в RAM-диске, возникает угроза надёжности всего кластера.

Для создания табличного пространства используется команда CREATE TABLESPACE, например::

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

Каталог должен существовать, быть пустым и принадлежать пользователю ОС, под которым запущен Postgres Pro. Все созданные впоследствии объекты, принадлежащие целевому табличному пространству, будут храниться в файлах расположенных в этом каталоге. Каталог не должен размещаться на съёмных или устройствах временного хранения, так как кластер может перестать функционировать из-за потери этого пространства.

Примечание

Обычно нет смысла создавать более одного пространства на одну логическую файловую систему, так как нет возможности контролировать расположение отдельных файлов в файловой системе. Однако Postgres Pro не накладывает никаких ограничений в этом отношении, и более того, напрямую не заботится о точках монтирования файловой системы. Просто осуществляется хранение файлов в указанных каталогах.

Создавать табличное пространство должен суперпользователь базы данных или пользователь с правами роли `pg_create_tablespace`, но после этого можно разрешить обычным пользователям работать с ним. Для этого необходимо предоставить им право `CREATE`.

Таблицы, индексы и целые базы данных могут храниться в отдельных табличных пространствах. Для этого пользователь с правом `CREATE` на табличное пространство должен указать его имя в качестве параметра соответствующей команды. Например, далее создаётся таблица в табличном пространстве `space1`:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

Как вариант, используйте параметр default_tablespace:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

Когда `default_tablespace` имеет значение отличное от пустой строки, он будет использоваться неявно в качестве значения параметра `TABLESPACE` в командах `CREATE TABLE` и `CREATE INDEX`, если в самой команде не задано иное.

Существует параметр temp_tablespaces, который указывает на размещение временных таблиц и индексов, а также файлов, создаваемых, например, при операциях сортировки больших наборов данных. Предпочтительнее, в качестве значения этого параметра, указывать не одно имя, а список из нескольких табличных пространств. Это поможет распределить нагрузку, связанную с временными объектами, по различным табличным пространствам. При каждом создании временного объекта будет случайным образом выбираться имя из указанного списка табличных пространств.

Табличное пространство, связанное с базой данных, также используется для хранения её системных каталогов. Более того, это табличное пространство используется по умолчанию для таблиц, индексов и временных файлов, создаваемых в базе данных, если не указано иное в выражении `TABLESPACE`, или переменной `default_tablespace`, или `temp_tablespaces` (соответственно). Если база данных создана без указания конкретного табличного пространства, то используется пространство, к которому принадлежит копируемый шаблон.

При инициализации кластера автоматически создаются два табличных пространства. Табличное пространство `pg_global` используется только для общих системных каталогов. Табличное пространство `pg_default` используется по умолчанию для баз данных `template1` и `template0` (в свою очередь, также является пространством по умолчанию для других баз данных, пока не будет явно указано иное в выражении `TABLESPACE` команды `CREATE DATABASE`).

После создания, табличное пространство можно использовать в рамках любой базы данных, при условии, что у пользователя имеются необходимые права. Это означает, что табличное пространство невозможно удалить до тех пор, пока не будут удалены все объекты баз данных, использующих это пространство.

Для удаления пустого табличного пространства используйте команду DROP TABLESPACE.

Чтобы получить список табличных пространств можно сделать запрос к системному каталогу `pg_tablespace`, например,

```
SELECT spcname FROM pg_tablespace;
```

Метакоманда `\db` утилиты psql также позволяет отобразить список существующих табличных пространств.

Каталог `$PGDATA/pg_tblspc` содержит символические ссылки, которые указывают на внешние табличные пространства кластера. Хоть и не рекомендуется, но возможно регулировать табличные пространства вручную, переопределяя эти ссылки. Ни при каких обстоятельствах эти операции нельзя проводить, пока запущен сервер баз данных. Обратите внимание, что в версии PostgreSQL 9.1 и более ранних также необходимо обновить информацию в `pg_tablespace` о новых расположениях. (Если это не сделать, то `pg_dump` будет продолжать выводить старые расположения табличных пространств.)

Глава 22. Локализация

Данная глава описывает доступные возможности локализации с точки зрения администратора. Postgres Pro поддерживает два средства локализации:

- Использование средств локализации операционной системы для обеспечения определяемых локалью порядка правил сортировки, форматирования чисел, перевода сообщений и прочих аспектов. Это рассматривается в Разделе 22.1 и Разделе 22.2.
- Обеспечение возможностей использования различных кодировок для хранения текста на разных языках и перевода кодировок между клиентом и сервером. Это рассматривается в Разделе 22.3.

22.1. Поддержка языковых стандартов

Поддержка *языковых стандартов* в приложениях относится к культурным предпочтениям, которые касаются алфавита, порядка сортировки, форматирования чисел и т. п. Postgres Pro использует соответствующие стандартам ISO C и POSIX возможности локали, предоставляемые операционной системой сервера. За дополнительной информацией обращайтесь к документации вашей системы.

22.1.1. Обзор

Поддержка локали автоматически инициализируется, когда кластер базы данных создаётся при помощи `initdb`. `initdb` инициализирует кластер баз данных по умолчанию с локалью из окружения выполнения. Поэтому, если ваша система уже использует локаль, которую вы хотите выбрать для вашего кластера баз данных, вам не требуется дополнительно совершать никаких действий. Если вы хотите использовать другую локаль (или вы точно не знаете, какая локаль используется в вашей системе), вы можете указать для `initdb`, какую именно локаль использовать, задав параметр `--locale`. Например:

```
initdb --locale=ru_RU
```

Данный пример для Unix-систем задаёт русский язык (`ru`), на котором говорят в России (RU) Другими вариантами могут быть `en_US` (американский английский) и `fr_CA` (канадский французский). Если в языковом окружении может использоваться более одного набора символов, значение может принимать вид *`language_territory.codeset`*. Например, `fr_BE.UTF-8` обозначает французский язык (fr), на котором говорят в Бельгии (BE), с кодировкой UTF-8.

То, какие локали и под какими именами доступны на вашей системе, зависит от того, что было включено в операционную систему производителем и что из этого было установлено. В большинстве Unix-систем команда `locale -a` выведет список доступных локалей. Windows использует более развёрнутые имена локалей, такие как `German_Germany` или `Russian_Russia.1251`, но принципы остаются теми же.

Иногда целесообразно объединить правила из различных локалей, например, использовать английские правила сравнения и испанские сообщения. Для этой цели существует набор категорий локали, каждая из которых управляет только определёнными аспектами правил локализации:

`LC_COLLATE`	Порядок сортировки строк
`LC_CTYPE`	Классификация символов (Что представляет собой буква? Каков её эквивалент в верхнем регистре?)
`LC_MESSAGES`	Язык сообщений
`LC_MONETARY`	Форматирование валютных сумм
`LC_NUMERIC`	Форматирование чисел
`LC_TIME`	Форматирование даты и времени

Эти имена категорий `initdb` принимает в качестве имён соответствующих параметров, позволяющих переопределить выбор локали в определённой категории. Например, чтобы настроить локаль на канадский французский, но при этом использовать американские правила форматирования денежных сумм, используйте `initdb --locale=fr_CA --lc-monetary=en_US`.

Если вы хотите, чтобы система работала без языковой поддержки, используйте специальное имя локали `C` либо эквивалентное ему `POSIX`.

Значения некоторых категорий локали должны быть заданы при создании базы данных. Вы можете использовать различные параметры локали для различных баз данных, но после создания базы вы уже не сможете изменить их для этой базы данных. `LC_COLLATE` и `LC_CTYPE` являются этими категориями. Они влияют на порядок сортировки в индексах, поэтому они должны быть зафиксированы, иначе индексы на текстовых столбцах могут повредиться. (Однако можно смягчить эти ограничения через задание правил сравнения, как это описано в разделе Раздел 22.2.) Значения по умолчанию для этих категорий определяются при запуске `initdb`, и эти значения используются при создании новых баз данных, если другие значения не указаны явно в команде `CREATE DATABASE`.

Прочие категории локали вы можете изменить в любое время, настроив параметры конфигурации сервера, которые имеют такое же имя как и категории локали (подробнее см. Подраздел 18.11.2). Значения, выбранные через `initdb`, фактически записываются лишь в файл конфигурации `postgresql.conf`, чтобы использоваться по умолчанию при запуске сервера. Если вы удалите эти значения из `postgresql.conf`, сервер получит соответствующие значения из своей среды выполнения.

Обратите внимание на то, что поведение локали сервера определяется переменными среды, установленными на стороне сервера, а не средой клиента. Таким образом, необходимо правильно сконфигурировать локаль перед запуском сервера. Если же клиент и сервер работают с разными локалями, то сообщения, возможно, будут появляться на разных языках в зависимости от того, где они возникают.

Примечание

Когда мы говорим о наследовании локали от среды выполнения, это означает следующее для большинства операционных систем: для определённой категории локали, к примеру, для правил сортировки, следующие переменные среды анализируются в приведённом ниже порядке до тех пор, пока одна из них не окажется заданной: `LC_ALL`, `LC_COLLATE` (или переменная, относящаяся к соответствующей категории), `LANG`. Если ни одна из этих переменных среды не задана, значение локали устанавливается по умолчанию в `C`.

Некоторые библиотеки локализации сообщений также обращаются к переменной среды `LANGUAGE`, которая заменяет все прочие параметры локализации при выборе языка сообщений. В случае затруднений, пожалуйста, воспользуйтесь документацией по вашей операционной системе, в частности, справкой по gettext.

Для того чтобы стал возможен перевод сообщений на язык, выбранный пользователем, NLS должен быть выбран на момент сборки (`configure --enable-nls`). В остальном поддержка локализации осуществляется автоматически.

22.1.2. Поведение

Локаль влияет на следующую функциональность SQL:

- Порядок сортировки в запросах с использованием `ORDER BY` или стандартных операторах сравнения текстовых данных
- Функции `upper`, `lower`, и `initcap`
- Операторы поиска по шаблону (`LIKE`, `SIMILAR TO`, и регулярные выражения в стиле POSIX); локаль влияет как на поиск без учёта регистра, так и на классификацию знаков по классам символов регулярных выражений
- Семейство функций `to_char`

- Возможность использовать индексы с предложениями `LIKE`

Недостатком использования отличающихся от `C` или `POSIX` локалей в Postgres Pro является влияние на производительность. Это замедляет обработку символов и мешает `LIKE` использовать обычные индексы. По этой причине используйте локали только в том случае, если они действительно вам нужны.

В качестве обходного решения, которое позволит Postgres Pro пользоваться индексами с предложениями `LIKE` с использованием локали, отличной от C, существует несколько классов пользовательских операторов. Они позволяют создать индекс, который выполняет строгое посимвольное сравнение, игнорируя правила сравнения, соответствующие локали. За дополнительными сведениями обратитесь к Разделу 11.10. Ещё один подход заключается в создании индексов с помощью правил сортировки `C`, как было сказано в Разделе 22.2.

22.1.3. Выбор локалей

Локали могут выбираться на разных уровнях в зависимости от требований. В обзоре выше показано, как выбрать локаль при вызове `initdb`, чтобы на уровне кластера установить для параметров локали значения, которые будут использоваться по умолчанию. В следующем списке показано, где ещё можно выбирать локали. В каждом пункте устанавливаются значения по умолчанию для последующих пунктов, и в каждом последующем пункте можно переопределить заданные выше значения на более низком уровне.

1. Как объяснялось выше, по умолчанию значения языковых параметров в инициализируемом кластере БД определяются переменными среды ОС. Во многих случаях этого достаточно: если операционная система настроена для нужного языка/региона, то Postgres Pro по умолчанию также будет вести себя в соответствии с этой локалью.
2. Как показано выше, значения параметров локали для инициализируемого кластера также можно указать в аргументах командой строки `initdb`. Используйте эту возможность, если вы хотите выбрать для вашей СУБД локаль, отличную от текущей локали операционной системы.
3. Локаль можно выбрать для отдельной базы данных. Предназначенные для этого параметры имеются у SQL-команды `CREATE DATABASE` и соответствующей ей внешней команды `createdb`. Это полезно, например, когда в одном кластере содержатся базы данных, принадлежащие различным клиентам с разными потребностями.
4. Параметры локали можно задать для отдельных столбцов таблицы. Для этого используются объекты SQL, которые называются *правила сортировки* и описаны в Разделе 22.2. Это полезно для сортировки данных на разных языках или для изменения порядка сортировки конкретной таблицы.
5. Наконец, локали можно выбирать на уровне запросов. При этом также используются объекты правил сортировки SQL. Это может быть полезно для динамического изменения порядка сортировки во время выполнения или для экспериментов с разными локалями.

22.1.4. Провайдеры локалей

Провайдер локалей указывает, какая библиотека определяет поведение локали для правил сортировки и классификаций символов.

Все команды и утилиты, позволяющие устанавливать параметры локали, как описано выше, позволяют выбрать и провайдера локали. Выбрать провайдера ICU можно так:

```
initdb --locale-provider=icu --icu-locale=en
```

За подробностями обратитесь к описанию соответствующих команд и программ. Обратите внимание, что можно использовать различных провайдеров локалей на разных уровнях, например, использовать `libc` по умолчанию для кластера, но иметь базу данных, использующую провайдера `icu`, а также иметь в этих базах данных объекты правил сортировки, использующие любых доступных провайдеров.

Независимо от провайдера локали, ОС всё равно будет использоваться для некоторых аспектов локализации, например сообщений (см. lc_messages).

Доступные провайдеры локалей перечислены ниже:

`builtin`

Провайдер локалей `builtin` использует встроенные операции. Для этого провайдера поддерживаются только локали `C` и `C.UTF-8`.

Локаль `C` ведёт себя так же, как локаль `C` в провайдере `libc`. Поведение локали может зависеть от кодировки базы данных.

Локаль `C.UTF-8` может использоваться, только когда кодировка базы данных — `UTF-8`. Поведение локали базируется на Unicode. Для сортировки используются только значения кодов символов. Классы символов регулярных выражений основаны на семантике, соответствующей POSIX, а для преобразования регистра используются «простые» правила.

`icu`

Провайдер локалей `icu` использует внешнюю библиотеку ICU . Postgres Pro должен быть настроен с поддержкой этой библиотеки.

ICU предоставляет функциональность сортировки и классификации символов, которые не зависят от операционной системы и кодировки базы данных. Это удобно, если планируется переход на другие платформы без каких-либо изменений. Параметры `LC_COLLATE` и `LC_CTYPE` можно установить независимо от используемой локали ICU.

Примечание

Для провайдера ICU результаты работы могут зависеть от используемой версии библиотеки ICU, поскольку она обновляется с учётом изменений в естественном языке с течением времени.

`libc`

Провайдер `libc` использует библиотеку C операционной системы. Поведение сортировки и классификации символов управляется параметрами `LC_COLLATE` и `LC_CTYPE`, поэтому их нельзя установить независимо.

Примечание

При использовании провайдера libc одна и та же локаль может вести себя по-разному на разных платформах.

22.1.5. Локали ICU

22.1.5.1. Имена локалей ICU

Формат ICU для имени локали — это языковая метка.

```
CREATE COLLATION mycollation1 (provider = icu, locale = 'ja-JP');
CREATE COLLATION mycollation2 (provider = icu, locale = 'fr');
```

22.1.5.2. Нормализация и проверка локали

Если при определении нового объекта правил сортировки ICU или базы данных с ICU в качестве провайдера имя локали не представляет собой языковую метку, оно преобразуется («нормализуется») в эту форму. Например,

```
CREATE COLLATION mycollation3 (provider = icu, locale = 'en-US-u-kn-true');
NOTICE:  using standard form "en-US-u-kn" for locale "en-US-u-kn-true"
CREATE COLLATION mycollation4 (provider = icu, locale = 'de_DE.utf8');
NOTICE:  using standard form "de-DE" for locale "de_DE.utf8"
```

Получив такое уведомление, проверьте, что `provider` и `locale` соответствуют ожидаемому результату. Для согласованных результатов при использовании провайдера ICU укажите стандартную метку языка вместо того, чтобы полагаться на преобразование.

Локаль без имени языка или со специальным именем языка `root` преобразуется в язык `und` (undefined, неопределённый).

Для более лёгкого перехода на ICU большинство имён локалей libc и из некоторых других форматов можно преобразовывать в языковые метки. Имя локали libc в ICU может вести себя не так, как в libc.

Если есть проблема с интерпретацией имени локали или если имя локали представляет собой язык или регион, которые ICU не распознаёт, выводится следующее предупреждение:

```
CREATE COLLATION nonsense (provider = icu, locale = 'nonsense');
WARNING:  ICU locale "nonsense" has unknown language "nonsense"
HINT:  To disable ICU locale validation, set parameter icu_validation_level to
 DISABLED.
CREATE COLLATION
```

icu_validation_level определяет способ вывода сообщения. Если для этого параметра не задано значение `ERROR`, правило сортировки всё равно будет создано, но поведение может отличаться от ожиданий пользователя.

22.1.5.3. Языковая метка

Языковая метка, определённая в BCP 47, представляет собой стандартизированный идентификатор, используемый для определения языков, регионов и другой информации о локали.

Основные языковые метки — это *`язык-регион`* или даже только *`язык`*. Здесь *`язык`* — это код языка (например, `fr` для французского), а *`регион`* — код региона (например, `CA` для Канады). Примеры: `ja-JP`, `de` или `fr-CA`.

Параметры правил сортировки могут быть включены в языковую метку для настройки поведения этих правил. ICU предоставляет широкие возможности для настройки: чувствительность (или нечувствительность) к диакритическим знакам, регистру и пунктуации, обработка цифр в тексте, и многие другие варианты для различных целей.

Чтобы включить эту дополнительную информацию о правилах сортировки в языковую метку, добавьте параметр `-u`, указывающий на наличие дополнительных параметров правил сортировки, за которым следует одна или несколько пар *`-ключ-значение`*, где *`ключ`* — это ключ для параметра правил сортировки, а *`значение`* — допустимое значение для этого параметра. Для логических параметров *`-ключ`* может быть указан без соответствующего *`-значения`*, что подразумевает значение `true`.

Например, языковая метка `en-US-u-kn-ks-level2` означает локаль с английским языком в регионе США с параметрами правил сортировки `kn` и `ks`, имеющими значения `true` и `level2` соответственно. Эти параметры означают, что правила сортировки нечувствительны к регистру и будут обрабатывать последовательность цифр как одно число:

```
CREATE COLLATION mycollation5 (provider = icu, deterministic = false, locale = 'en-US-
u-kn-ks-level2');
SELECT 'aB' = 'Ab' COLLATE mycollation5 as result;
 result
--------
 t
(1 row)

SELECT 'N-45' < 'N-123' COLLATE mycollation5 as result;
 result
--------
 t
(1 row)
```

Обратитесь к Подразделу 22.2.3 для получения подробной информации и дополнительных примеров использования языковых меток с информацией о пользовательских правилах сортировки для локали.

22.1.6. Проблемы

Если поддержка локализации не работает в соответствии с объяснением, данным выше, проверьте, насколько корректна конфигурация поддержки локализации в вашей операционной системе. Чтобы проверить, какие локали установлены на вашей системе, вы можете использовать команду `locale -a`, если ваша операционная система поддерживает это.

Проверьте, действительно ли Postgres Pro использует локаль, которую вы подразумеваете. Параметры `LC_COLLATE` и `LC_CTYPE` определяются при создании базы данных, и не могут быть изменены, за исключением случаев, когда создаётся новая база данных. Прочие параметры локали, включая `LC_MESSAGES` и `LC_MONETARY` первоначально определены средой, в которой запускается сервер, но могут быть оперативно изменены. Вы можете проверить текущие параметры локали с помощью команды `SHOW`.

Клиентские приложения, которые обрабатывают ошибки сервера, разбирая текст сообщения об ошибке, очевидно, столкнутся с проблемами, когда сообщения сервера будут на другом языке. Авторам таких приложений рекомендуется пользоваться системой кодов ошибок в качестве альтернативы.

22.2. Поддержка правил сортировки

Правила сортировки позволяют устанавливать порядок сортировки и особенности классификации символов в отдельных столбцах или даже при выполнении отдельных операций. Это смягчает последствия того, что параметры базы данных `LC_COLLATE` и `LC_CTYPE` невозможно изменить после её создания.

22.2.1. Основные понятия

Концептуально, каждое выражение с типом данных, к которому применяется сортировка, имеет правила сортировки. (Встроенными сортируемыми типами данных являются `text`, `varchar`, и `char`. Типы, определяемые в базе пользователем, могут также быть отмечены как сортируемые, и, конечно, *домен* на основе сортируемого типа данных является сортируемым.) Если выражение содержит ссылку на столбец, правила сортировки выражения определяются правилами сортировки столбца. Если выражение — константа, правилами сортировки являются стандартные правила для типа данных константы. Правила сортировки более сложных выражений являются производной от правил сортировки входящих в него частей, как описано ниже.

Правилами сортировки выражения могут быть правила сортировки «по умолчанию», что означает использование параметров локали, установленных для базы данных. Также возможно, что правила сортировки выражения могут не определиться. В таких случаях операции упорядочивания и другие операции, для которых необходимы правила сортировки, завершатся с ошибкой.

Когда база данных должна выполнить упорядочивание или классификацию символов, она использует правила сортировки выполняемого выражения. Это происходит, к примеру, с предложениями `ORDER BY` и такими вызовами функций или операторов как `<`. Правила сортировки, которые применяются в предложении `ORDER BY`, это просто правила ключа сортировки. Правила сортировки, применяемые к вызову функции или оператора, определяются их параметрами, как описано ниже. В дополнение к операциям сравнения, правила сортировки учитываются функциями, преобразующими регистр символов, такими как `lower`, `upper`, и `initcap`; операторами поиска по шаблону; и функцией `to_char` и связанными с ней.

При вызове функции или оператора правило сортировки определяется в зависимости от того, какие правила заданы для аргументов во время выполнения данной операции. Если результатом вызова функции или оператора является сортируемый тип данных, правила сортировки также используются во время разбора как определяемые правила сортировки функции или выражения оператора, когда для внешнего выражения требуется знание правил сортировки.

Определение правил сортировки выражения может быть неявным или явным. Это отличие влияет на то, как комбинируются правила сортировки, когда несколько разных правил появляются в выражении. Явное определение правил сортировки возникает, когда используется предложение `COLLATE`; все прочие варианты являются неявными. Когда необходимо объединить несколько правил сортировки, например, в вызове функции, используются следующие правила:

1. Если для одного из выражений-аргументов правило сортировки определено явно, то и для других аргументов явно задаваемое правило должно быть тем же, иначе возникнет ошибка. В случае присутствия явного определения правила сортировки, оно становится результирующим для всей операции.
2. В противном случае все входные выражения должны иметь одни и те же неявно определяемые правила сортировки или правила сортировки по умолчанию. Если присутствуют какие- либо правила сортировки, отличные от заданных по умолчанию, получаем результат комбинации правил сортировки. Иначе результатом станут правила сортировки, заданные по умолчанию.
3. Если среди входных выражений есть конфликтующие неявные правила сортировки, отличные от заданных по умолчанию, тогда комбинация рассматривается как имеющая неопределённые правила сортировки. Это не является условием возникновения ошибки, если вызываемой конкретной функции не требуются данные о правилах сортировки, которые ей следует применить. Если же такие данные требуются, это приведёт к ошибке во время выполнения.

В качестве примера рассмотрим данное определение таблицы:

```
CREATE TABLE test1 (
    a text COLLATE "ru_RU",
    b text COLLATE "es_ES",
    ...
);
```

Затем в

```
SELECT a < 'foo' FROM test1;
```

выполняется оператор сравнения `<` согласно правилам `ru_RU`, так как выражение объединяет неявно определяемые правила сортировки с правилами, заданными по умолчанию. Но в

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

сравнение выполняется с помощью правил `fr_FR`, так как явное определение правил сортировки переопределяет неявное. Кроме того, в запросе

```
SELECT a < b FROM test1;
```

анализатор запросов не может определить, какое правило сортировки использовать, поскольку столбцы `a` и `b` имеют конфликтующие неявные правила сортировки. Так как оператору `<` требуется знать, какое правило использовать, это приведёт к ошибке. Ошибку можно устранить, применив явное указание правил сортировки к любому из двух входных выражений. Например:

```
SELECT a < b COLLATE "ru_RU" FROM test1;
```

либо эквивалентное ему

```
SELECT a COLLATE "ru_RU" < b FROM test1;
```

С другой стороны, следующее выражение схожей структуры

```
SELECT a || b FROM test1;
```

не приводит к ошибке, поскольку для оператора `||` правила сортировки не имеют значения, так как результат не зависит от сортировки.

Правила сортировки, назначенные функции или комбинации входных выражений оператора, также могут быть применены к функции или результату оператора, если функция или оператор возвращают результат сортируемого типа данных. Так, в

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

упорядочение будет происходить согласно правилам `ru_RU`. Но данный запрос:

```
SELECT * FROM test1 ORDER BY a || b;
```

приводит к ошибке, потому что, даже если оператору `||` не нужно знать правила сортировки, предложению `ORDER BY` это требуется. Как было сказано выше, конфликт может быть разрешён при помощи явного указания правил сортировки:

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

22.2.2. Управление правилами сортировки

Правила сортировки представляют собой объект схемы SQL, который сопоставляет SQL-имя с локалью, реализуемой библиотекой, установленной в операционной системе. В определении правила сортировки задаётся *провайдер*, то есть библиотека, реализующая правило сортировки. Стандартный провайдер с именем `libc` использует системную библиотеку C и предоставляет её локали. Именно эти локали используются большинством утилит операционной системы. Также есть провайдер `icu`, который использует внешнюю библиотеку ICU. Локали ICU можно использовать, только если поддержка ICU была включена в конфигурации сборки Postgres Pro.

Правило сортировки, предоставляемое провайдером `libc`, сопоставляется с комбинацией параметров `LC_COLLATE` и `LC_CTYPE`, которую может принять системный вызов `setlocale()`. (Основная цель правила сортировки — настроить параметр `LC_COLLATE`, который управляет упорядочиванием символов. Однако на практике редко требуется иметь значение `LC_CTYPE`, отличное от `LC_COLLATE`, поэтому удобнее объединить их в одну сущность, и не создавать отдельную инфраструктуру для указания `LC_CTYPE` в выражениях.) Правила сортировки `libc` также связаны с кодировкой набора символов (см. Раздел 22.3). Одно и то же имя правила сортировки может существовать для разных кодировок.

Объект правила сортировки, предоставляемой провайдером `icu`, сопоставляется с именованным сортировщиком, реализуемым библиотекой ICU. ICU не поддерживает различные характеристики «collate» и «ctype», так что они всегда совпадают. Кроме того, правила сортировки ICU не зависят от кодировки, так что в базе данных будет всего одно правило сортировки ICU с определённым именем.

22.2.2.1. Стандартные правила сортировки

На всех платформах поддерживаются следующие правила сортировки:

`unicode`

Это правило сортировки использует Алгоритм сортировки Unicode с Таблицей сортировки символов Unicode по умолчанию. Оно доступно во всех кодировках. Для его использования требуется поддержка ICU, и поведение зависит от того, какая версия ICU используется в данной сборке Postgres Pro. (Это правило сортировки работает так же, как и корневая локаль ICU; см. `und-x-icu` («undefined», неопределённая).)

`ucs_basic`

С этим стандартным правилом SQL сортировка выполняется по кодам символов Unicode, а не в порядке, принятом в естественном языке, и только символы ASCII от «`A`» до «`Z`» обрабатываются как буквы. Такое поведение эффективно и стабильно работает во всех версиях. Правило доступно только для кодировки `UTF8`. (Это правило сортировки работает так же, как и правило `C` локали libc в кодировке `UTF8`.)

`pg_c_utf8`

С этим правилом сортировка выполняется по кодам символов Unicode, а не в порядке, принятом в естественном языке. Для функций `lower`, `initcap` и `upper` используется простое преобразование регистра Unicode. Для сопоставления шаблонов (включая регулярные выражения) используется совместимый с POSIX вариант Unicode (*Совместимость*). Такое поведение эффективно и стабильно в основной версии Postgres Pro. Данное правило сортировки доступно только для кодировки `UTF8`.

`C` (равнозначно `POSIX`)

Правила сортировки `C` и `POSIX` основаны на «традиционном» поведении C. С этими правилами сортировка выполняется по байтовым значениям, а не в порядке, принятом в естественном языке, и только символы ASCII от «`A`» до «`Z`» обрабатываются как буквы. Поведение эффективно и стабильно во всех версиях для данной кодировки базы данных, но может отличаться для других кодировок.

`default`

С правилом сортировки `default` выбирается локаль, указанная при создании базы данных.

Поддержка дополнительных правил сортировки зависит от операционной системы, а эффективность и стабильность этих правил — от провайдера, его версии и локали.

22.2.2.2. Предопределённые правила сортировки

Если операционная система поддерживает использование нескольких локалей в одной программе (`newlocale` и связанные функции) или включена поддержка ICU, то при инициализации кластера баз данных программа `initdb` наполняет системный каталог `pg_collation` информацией обо всех локалях, которые обнаруживает в этот момент в операционной системе.

Для просмотра всех имеющихся локалей выполните запрос `SELECT * FROM pg_collation` или команду `\dOS+` в psql.

22.2.2.2.1. Правила сортировки libc

Например, операционная система может предоставлять локаль с именем `ru_RU.utf8` При этом программа `initdb` создаст правило сортировки с именем `ru_RU.utf8` для кодировки `UTF8`, в котором `LC_COLLATE` и `LC_CTYPE` будут равны `ru_RU.utf8`. Также будет создано правило сортировки с именем без метки `.utf8` в окончании. Таким образом, вы можете использовать это правило под именем `ru_RU`, которое будет компактнее и не будет зависеть от кодировки. Заметьте, что изначальный набор имён правил сортировки тем не менее зависит от платформы.

Стандартный набор правил сортировки, предоставляемый провайдером `libc`, сопоставляется непосредственно с локалями, установленными в операционной системе (их можно просмотреть с помощью команды `locale -a`). В случаях, когда у правила сортировки `libc` должны быть различные значения `LC_COLLATE` и `LC_CTYPE`, или когда в операционную систему после инициализации СУБД устанавливаются новые локали, создать новое правило сортировки можно с помощью команды CREATE COLLATION. Новые локали операционной системы можно также импортировать в массовом порядке, воспользовавшись функцией `pg_import_system_collations()`.

В любой базе данных имеют значение только те правила сортировки, которые используют кодировку этой базы данных. Прочие записи в `pg_collation` игнорируются. Таким образом, усечённое имя правил сортировки, такое как `ru_RU`, может считаться уникальным внутри данной базы данных, даже если бы оно не было уникальным глобально. Использование усечённого имени сортировки рекомендуется, так как при переходе на другую кодировку базы данных придётся выполнить на одно изменение меньше. Однако следует помнить, что правила сортировки `default`, `C` и `POSIX` можно использовать независимо от кодировки базы данных.

В Postgres Pro предполагается, что отдельные объекты правил сортировки несовместимы, даже когда они имеют идентичные свойства. Так, например,

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

выведет сообщение об ошибке, несмотря на то, что поведение правил сортировки `C` и `POSIX` идентично. По этой причине смешивать усечённые и полные имена правил сортировки не рекомендуется.

22.2.2.2.2. Правила сортировки ICU

С ICU не представляется разумным перечислять все возможные имена локалей. ICU использует для локалей определённую схему именования, но имён локалей может быть гораздо больше, чем собственно различных локалей. Программа `initdb`, используя API ICU, извлекает список различных локалей и наполняет начальный набор правил в базе данных. Правила сортировки провайдера ICU создаются с именами, включающими метку языка в формате BCP 47 и указание расширения «для частного использования» (`-x-icu`), для отличия от локалей libc.

Например, могут быть созданы такие правила сортировки:

`ru-x-icu`

Русское правило сортировки, стандартный вариант

`ru-UA-x-icu`

Русское правило сортировки для Украины, стандартный вариант

(Например, существует правило `ru-RU-x-icu`, но на момент написания документации оно равнозначно правилу `ru-x-icu`.)

`und-x-icu` («undefined», неопределённая)

«Корневое» правило сортировки ICU. Оно устанавливает разумный языконезависимый порядок сортировки.

Некоторые (редко используемые) кодировки не поддерживаются ICU. Когда база имеет одну из таких кодировок, записи правил сортировки ICU в `pg_collation` игнорируются. При попытке использовать их будет выдана ошибка с сообщением вида «правило сортировки "de-x-icu" для кодировки "WIN874" не существует».

22.2.2.3. Создание новых правил сортировки

Если стандартных и предопределённых правил сортировки недостаточно, пользователи могут создавать собственные правила сортировки, используя команду SQL CREATE COLLATION.

Стандартные и предопределённые правила сортировки находятся в схеме `pg_catalog`, как и все предопределённые объекты. Пользовательские правила сортировки должны создаваться в пользовательских схемах. Помимо прочего, это полезно тем, что их будет выгружать `pg_dump`.

22.2.2.3.1. Правила сортировки libc

Новые правила сортировки libc могут создаваться так:

```
CREATE COLLATION russian (provider = libc, locale = 'ru_RU');
```

Точные значения, которые могут допускаться в предложении `locale` в этой команде, зависят от операционной системы. В Unix-подобных системах их список выдаёт команда `locale -a`.

Так как предопределённый набор правил сортировки libc уже включает все правила сортировки, определённые в операционной системе в момент инициализации базы данных, необходимость создавать новые правила вручную обычно не возникает. Такая потребность может возникнуть, когда нужно сменить систему именования (в этом случае см. Подраздел 22.2.2.3.3) либо когда операционная система была обновлена и в ней появились новые определения локалей (в этом случае см. `pg_import_system_collations()`).

22.2.2.3.2. Правила сортировки ICU

Правила сортировки ICU могут быть созданы следующим образом:

```
CREATE COLLATION german (provider = icu, locale = 'de-DE');
```

Имена локалей ICU указываются как языковые метки BCP 47, но также принимается большинство имён локалей в стиле libc. Если это возможно, имена локалей в стиле libc преобразуются в языковые метки.

Для новых правил сортировки ICU допускается настройка поведения путём указания параметров в языковой метке. За подробным описанием и примерами обратитесь к Подразделу 22.2.3.

22.2.2.3.3. Копирование правил сортировки

Команда CREATE COLLATION может также создать новое правило сортировки из существующего, что может быть полезно для использования имён, независимых от операционных систем, создания имён для совместимости или использования правил сортировки ICU под более понятными именами. Например:

```
CREATE COLLATION russian FROM "ru_RU";
CREATE COLLATION french FROM "fr-x-icu";
```

22.2.2.4. Недетерминированные правила сортировки

Правило сортировки может быть либо *детерминированным*, либо *недетерминированным*. С детерминированными правилами сортировки используются детерминированные сравнения, что означает, что сроки считаются равными, только если они состоят из одинаковой последовательности байтов. Недетерминированное же сравнение может признать равными строки, состоящие и из разных байтов. Обычно это требуется при сравнении без учёта регистра или ударения, а также при сравнении строк в различных нормальных формах Unicode. Как именно будут реализованы подобные сравнения, определяется провайдером правил сортировки; флаг детерминированности только отмечает, будет ли вопрос равенства решаться побайтовым сравнением. Подробнее сопутствующая терминология описывается в *Техническом стандарте Unicode 10*.

Чтобы создать недетерминированное правило сортировки, укажите свойство `deterministic = false` в команде `CREATE COLLATION`, например так:

```
CREATE COLLATION ndcoll (provider = icu, locale = 'und', deterministic = false);
```

В данном примере будет использоваться стандартное правило сортировки Unicode в недетерминированном режиме. В частности, это позволит корректно сравнивать строки в различных нормальных формах. Для более интересных применений задействуются специальные возможности ICU, рассмотренные выше. Например:

```
CREATE COLLATION case_insensitive (provider = icu, locale = 'und-u-ks-level2',
 deterministic = false);
CREATE COLLATION ignore_accents (provider = icu, locale = 'und-u-ks-level1-kc-true',
 deterministic = false);
```

Все стандартные и предопределённые правила сортировки являются детерминированными, и так же детерминированными по умолчанию создаются пользовательские правила. Недетерминированные правила сортировки обеспечивают более «правильное» поведение, особенно в части использования всех возможностей Unicode и обработки множества особых случаев, но они имеют и ряд недостатков. Прежде всего, их применение отрицательно сказывается на производительности. Заметьте в частности, что в B-деревьях с недетерминированным правилами не будет производиться исключение дубликатов. К тому же с недетерминированными правилами сортировки невозможны некоторые операции, например, поиск по шаблону. Поэтому применять их следует только тогда, когда в этом есть явная необходимость.

Подсказка

Для эффективной обработки текста в различных формах нормализации Unicode также можно использовать функции/выражения `normalize` и `is normalized`, позволяющие предварительно обработать или проверить строки, и не прибегать к использованию недетерминированных правил сортировки. Однако каждый подход имеет свои минусы.

22.2.3. Пользовательские правила сортировки ICU

ICU позволяет управлять поведением правил сортировки путём определения новых правил с параметрами, заданными в виде части языковой метки. Эти параметры могут изменить порядок сортировки в соответствии с различными потребностями. Например:

```
-- игнорировать различия в диакритических знаках и регистре
CREATE COLLATION ignore_accent_case (provider = icu, deterministic = false, locale =
 'und-u-ks-level1');
SELECT 'Å' = 'A' COLLATE ignore_accent_case; -- true
SELECT 'z' = 'Z' COLLATE ignore_accent_case; -- true

-- буквы в верхнем регистре идут перед буквами в нижнем
```

```
CREATE COLLATION upper_first (provider = icu, locale = 'und-u-kf-upper');
SELECT 'B' < 'b' COLLATE upper_first; -- true

-- сортировать последовательности цифр по числовым значениям и игнорировать знаки
 пунктуации
CREATE COLLATION num_ignore_punct (provider = icu, deterministic = false, locale =
 'und-u-ka-shifted-kn');
SELECT 'id-45' < 'id-123' COLLATE num_ignore_punct; -- true
SELECT 'w;x*y-z' = 'wxyz' COLLATE num_ignore_punct; -- true
```

Многие доступные параметры описаны в Подразделе 22.2.3.2, а дополнительную информацию можно найти в Подразделе 22.2.3.5.

22.2.3.1. Уровни сравнения ICU

Сравнение двух строк (сортировка) в ICU определяется многоуровневым процессом, в котором текстовые признаки сгруппированы в «уровни». Обработка каждого уровня зависит от параметрами правил сортировки. На более высоких уровнях выполняется более детальное сравнение.

Таблица 22.1 показывает, какие различия в текстовых признаках имеют приоритет при сравнении на данном уровне. Символ Unicode `U+2063` является невидимым разделителем и, как видно из таблицы, игнорируется на всех уровнях сравнения ниже `identic`.

Таблица 22.1. Уровни сортировки ICU

Уровень	**Описание**	`'f' = 'f'`	`'ab' = U&'a\2063b'`	`'x-y' = 'x_y'`	`'g' = 'G'`	`'n' = 'ñ'`	`'y' = 'z'`
level1	Базовый символ	`true`	`true`	`true`	`true`	`true`	`false`
level2	Диакритические знаки	`true`	`true`	`true`	`true`	`false`	`false`
level3	Регистры/Варианты	`true`	`true`	`true`	`false`	`false`	`false`
level4	Пунктуация[a]	`true`	`true`	`false`	`false`	`false`	`false`
identic	Все	`true`	`false`	`false`	`false`	`false`	`false`

[a] только с `ka-shifted`; см. Таблицу 22.2

На каждом уровне, даже при выключенной полной нормализации, выполняется базовая нормализация. Например, `'á'` может состоять из кодов символов `U&'\0061\0301'` или одного кода символа `U&'\00E1'`, и эти последовательности будут считаться равными даже на уровне `identic`. Чтобы учитывались любые различия в представлениях кодов символов, используйте параметры правил сортировки, созданные с параметром `deterministic`, который имеет значение `true`.

22.2.3.1.1. Примеры уровней сортировки

```
CREATE COLLATION level3 (provider = icu, deterministic = false, locale = 'und-u-ka-
shifted-ks-level3');
CREATE COLLATION level4 (provider = icu, deterministic = false, locale = 'und-u-ka-
shifted-ks-level4');
CREATE COLLATION identic (provider = icu, deterministic = false, locale = 'und-u-ka-
shifted-ks-identic');

-- невидимый разделитель, игнорируемый на всех уровнях кроме identic
SELECT 'ab' = U&'a\2063b' COLLATE level4; -- true
SELECT 'ab' = U&'a\2063b' COLLATE identic; -- false

-- знаки пунктуации игнорируются на уровне 3, но не на уровне 4
SELECT 'x-y' = 'x_y' COLLATE level3; -- true
```

```
SELECT 'x-y' = 'x_y' COLLATE level4; -- false
```

22.2.3.2. Параметры правил сортировки для локали ICU

Таблица 22.2 показывает доступные параметры правил сортировки, которые можно использовать как часть языковой метки для настройки правил сортировки.

Таблица 22.2. Параметр правил сортировки ICU

Ключ	Значения	По умолчанию	Описание
`co`	`emoji`, `phonebk`, `standard`, `...`	`standard`	Тип правил сортировки. Обратитесь к Подразделу 22.2.3.5 для получения дополнительной информации.
`ka`	`noignore`, `shifted`	`noignore`	Если установлено значение `shifted`, некоторые символы (например, знаки препинания или пробел) игнорируются при сравнении. Чтобы ключ `ks` применялся, он должен быть установлен на уровне `level3` или ниже. Установите ключ `kv`, чтобы указать игнорируемые классы символов.
`kb`	`true`, `false`	`false`	Обратное сравнение для различий уровня 2. Например, с локалью `und-u-kb` `'àe'` окажется перед `'aé'`.
`kc`	`true`, `false`	`false`	Относит случай к «уровню 2.5», который находится между диакритическими знаками и другими признаками уровня 3. Если установлено значение `true`, а для `ks` установлено значение `level1`, диакритические знаки будут игнорироваться, но будет учитываться регистр.
`kf`	`upper`, `lower`, `false`	`false`	Если установлено значение `upper`, символы в верхнем регистре идут перед символами в нижнем, а если `lower` — наоборот. Если установлено значение `false`, сортировка зависит от правил локали.
`kn`	`true`, `false`	`false`	Если установлено значение `true`, последовательности цифр сортируются по числовым значениям. Например, `'id-45'` окажется перед `'id-123'`.
`kk`	`true`, `false`	`false`	Включает полную нормализацию, может повлиять на производительность. Базовая нормализация выполняется, даже если установлено значение `false`. Локали для языков, требующих полной нормализации, обычно включают её по умолчанию. В некоторых случаях важна полная нормализация, например, когда к одному символу применяется несколько диакритических знаков. Например, последовательности кодов символов `U&'\0065\0323\0302'` и `U&'\0065\0302\0323'` представляют собой `e` с циркумфлексом и точками снизу, применяемыми в разном порядке. При включённой полной нормализации эти последовательности кодов символов рассмат-

Ключ	Значения	По умолчанию	Описание
			риваются как равные, в противном случае они неравны.
`kr`	`space`, `punct`, `symbol`, `currency`, `digit`, *`id_скрипта`*		Принимает одно или несколько допустимых значений или любой *`id_скрипта`* BCP 47, например `latn` («латынь») или `grek` («греческий»). Несколько значений разделяются знаком «-». Переопределяет упорядочение классов символов: символы, принадлежащие более раннему классу в списке, указываются перед символами, принадлежащими более позднему классу в списке. Например, значение `digit-currency-space` (как часть языковой метки, такой как `und-u-kr-digit-currency-space`) сортирует знаки препинания перед цифрами и пробелами.
`ks`	`level1`, `level2`, `level3`, `level4`, `identic`	`level3`	Чувствительность (или «степень») при сравнении, где на уровне на уровне `identic` большинство различий учитывается, а на уровне `level1` — игнорируется. За подробностями обратитесь к Таблице 22.1.
`kv`	`space`, `punct`, `symbol`, `currency`	`punct`	Классы символов игнорируются при сравнении на уровне 3. Установка следующего значения в списке включает все предыдущие, например `symbol` также включает `punct` и `space` в символах, которые следует игнорировать. Чтобы работать, ключ `ka` должен иметь значение `shifted`, а ключ `ks` — значение `level3` или ниже.

Значения по умолчанию могут зависеть от локали. Приведённая выше таблица не является полной. Обратитесь к Подразделу 22.2.3.5 для получения дополнительной информации.

Примечание

В случае со многими параметрами правил сортировки, чтобы конкретный параметр имел желаемый эффект, нужно создавать правила сортировки с параметром `deterministic`, для которого задано значение `false` (см. Подраздел 22.2.2.4). Кроме того, некоторые параметры вступают в силу только тогда, когда для ключа `ka` установлено значение `shifted` (см. Таблицу 22.2).

22.2.3.3. Примеры параметров правил сортировки

```
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale = 'de-u-co-phonebk');
```

Немецкое правило сортировки с порядком, принятым для телефонной книги

```
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale = 'und-u-co-emoji');
```

Корневое правило с сортировкой эмодзи, соответствующее техническому стандарту Unicode №51

```
CREATE COLLATION latn_cyrl (provider = icu, locale = 'ru-RU-u-kr-latn-cyrl');
```

Правило сортировки, с которым латинские буквы идут перед буквами кириллицы. (По умолчанию сначала идут буквы кириллицы.)

```
CREATE COLLATION upperfirst (provider = icu, locale = 'ru-RU-u-kf-upper');
```

Правило сортировки, с которым буквы в верхнем регистре идут перед буквами в нижнем. (По умолчанию сначала идут буквы в нижнем регистре.)

```
CREATE COLLATION special (provider = icu, locale = 'ru-RU-u-kf-upper-kr-latn-cyrl');
```

Правило, в котором сочетаются предыдущие свойства.

22.2.3.4. Правила для сортировки ICU

Если вышеуказанных параметров правил сортировки недостаточно, порядок элементов сортировки можно изменить с помощью особых правил сортировки, синтаксис которых подробно описан в *https://unicode-org.github.io/icu/userguide/collation/customization/*.

В этом простом примере создаётся правило сортировки на основе корневой локали с дополнительным условием сортировки:

```
CREATE COLLATION custom (provider = icu, locale = 'und', rules = '&V << w <<< W');
```

Так буква «W» окажется после «V», но это условие рассматривается как второстепенное отличие, подобное диакритическому знаку. Такие правила содержатся в определениях локалей некоторых языков. (Конечно, если определение локали уже содержит нужные правила, их не нужно снова указывать явно.)

Ниже показан более сложный пример. Следующий оператор задаёт правило сортировки с именем `ebcdic`, при использовании которого символы US-ASCII сортируются в порядке, соответствующем кодировке EBCDIC.

```
CREATE COLLATION ebcdic (provider = icu, locale = 'und',
rules = $$
& ' ' < '.' < '<' < '(' < '+' < \|
< '&' < '!' < '$' < '*' < ')' < ';'
< '-' < '/' < ',' < '%' < '_' < '>' < '?'
< '`' < ':' < '#' < '@' < \' < '=' < '"'
<*a-r < '~' <*s-z < '^' < '[' < ']'
< '{' <*A-I < '}' <*J-R < '\' <*S-Z <*0-9
$$);

SELECT c
FROM (VALUES ('a'), ('b'), ('A'), ('B'), ('1'), ('2'), ('!'), ('^')) AS x(c)
ORDER BY c COLLATE ebcdic;
 c
---
 !
 a
 b
 ^
 A
 B
 1
 2
```

22.2.3.5. Ссылки на внешние ресурсы про ICU

Этот раздел (Подраздел 22.2.3) представляет собой лишь краткий обзор поведения ICU и языковых меток. Технические подробности, дополнительные параметры и новое поведение описаны в следующих документах:

- *Технический стандарт Unicode №35*
- *BCP 47*
- *репозиторий CLDR*
- *https://unicode-org.github.io/icu/userguide/locale/*

- *https://unicode-org.github.io/icu/userguide/collation/*

22.3. Поддержка кодировок

Поддержка кодировок в Postgres Pro позволяет хранить текст в различных кодировках, включая однобайтовые кодировки, такие как входящие в семейство ISO 8859 и многобайтовые кодировки, такие как EUC (Extended Unix Code), UTF-8 и внутренний код Mule. Все поддерживаемые кодировки могут прозрачно использоваться клиентами, но некоторые не поддерживаются сервером (в качестве серверной кодировки). Кодировка по умолчанию выбирается при инициализации кластера базы данных Postgres Pro при помощи `initdb`. Она может быть переопределена при создании базы данных, что позволяет иметь несколько баз данных с разными кодировками.

Важным ограничением, однако, является то, что кодировка каждой базы данных должна быть совместима с параметрами локали базы данных `LC_CTYPE` (классификация символов) и `LC_COLLATE` (порядок сортировки строк). Для локали `C` или `POSIX` подойдёт любой набор символов, но для других локалей, предоставляемых библиотекой libc, есть только один набор символов, который будет работать правильно. (Однако в среде Windows кодировка UTF-8 может использоваться с любой локалью.) Если у вас включена поддержка ICU, локали, предоставляемые библиотекой ICU, можно использовать с большинством (но не всеми) кодировками на стороне сервера.

22.3.1. Поддерживаемые кодировки

Таблица 22.3 показывает кодировки, доступные для использования в Postgres Pro.

Таблица 22.3. Кодировки Postgres Pro

Имя	Описание	Язык	Поддержка на сервере	ICU?	Байт на символ	Псевдонимы
`BIG5`	Big Five	Традиционные китайские иероглифы	Нет	Нет	1–2	`WIN950`, `Windows950`
`EUC_CN`	Extended UNIX Code-CN	Упрощённые китайские иероглифы	Да	Да	1–3	
`EUC_JP`	Extended UNIX Code-JP	Японский	Да	Да	1–3	
`EUC_JIS_2004`	Extended UNIX Code-JP, JIS X 0213	Японский	Да	Нет	1–3	
`EUC_KR`	Extended UNIX Code-KR	Корейский	Да	Да	1–3	
`EUC_TW`	Extended UNIX Code-TW	Традиционные китайские иероглифы, тайваньский	Да	Да	1–4	
`GB18030`	Национальный стандарт	Китайский	Нет	Нет	1–4	
`GBK`	Расширенный национальный стандарт	Упрощённые китайские иероглифы	Нет	Нет	1–2	`WIN936`, `Windows936`
`ISO_8859_5`	ISO 8859-5, ECMA 113	Латинский/Кириллица	Да	Да	1	
`ISO_8859_6`	ISO 8859-6, ECMA 114	Латинский/Арабский	Да	Да	1	

Имя	**Описание**	**Язык**	**Поддержка на сервере**	**ICU?**	**Байт на символ**	**Псевдонимы**
`ISO_8859_7`	ISO 8859-7, ECMA 118	Латинский/Греческий	Да	Да	1	
`ISO_8859_8`	ISO 8859-8, ECMA 121	Латинский/Иврит	Да	Да	1	
`JOHAB`	JOHAB	Корейский (Хангыль)	Нет	Нет	1–3	
`KOI8R`	KOI8-R	Кириллица (Русский)	Да	Да	1	`KOI8`
`KOI8U`	KOI8-U	Кириллица (Украинский)	Да	Да	1	
`LATIN1`	ISO 8859-1, ECMA 94	Западноевропейские	Да	Да	1	`ISO88591`
`LATIN2`	ISO 8859-2, ECMA 94	Центральноевропейские	Да	Да	1	`ISO88592`
`LATIN3`	ISO 8859-3, ECMA 94	Южноевропейские	Да	Да	1	`ISO88593`
`LATIN4`	ISO 8859-4, ECMA 94	Североевропейские	Да	Да	1	`ISO88594`
`LATIN5`	ISO 8859-9, ECMA 128	Турецкий	Да	Да	1	`ISO88599`
`LATIN6`	ISO 8859-10, ECMA 144	Скандинавские	Да	Да	1	`ISO885910`
`LATIN7`	ISO 8859-13	Балтийские	Да	Да	1	`ISO885913`
`LATIN8`	ISO 8859-14	Кельтские	Да	Да	1	`ISO885914`
`LATIN9`	ISO 8859-15	LATIN1 со знаком евро и диакритическими знаками	Да	Да	1	`ISO885915`
`LATIN10`	ISO 8859-16, ASRO SR 14111	Румынский	Да	Нет	1	`ISO885916`
`MULE_INTERNAL`	Внутренний код Mule	Мультиязычный редактор Emacs	Да	Нет	1–4	
`SJIS`	Shift JIS	Японский	Нет	Нет	1–2	`Mskanji`, `ShiftJIS`, `WIN932`, `Windows932`
`SHIFT_JIS_2004`	Shift JIS, JIS X 0213	Японский	Нет	Нет	1–2	
`SQL_ASCII`	не указан (см. текст)	*any*	Да	Нет	1	
`UHC`	Унифицированный код Хангыль	Корейский	Нет	Нет	1–2	`WIN949`, `Windows949`

Имя	**Описание**	**Язык**	**Поддержка на сервере**	**ICU?**	**Байт на символ**	**Псевдонимы**
`UTF8`	Unicode, 8-bit	*все*	Да	Да	1–4	`Unicode`
`WIN866`	Windows CP866	Кириллица	Да	Да	1	`ALT`
`WIN874`	Windows CP874	Тайский	Да	Нет	1	
`WIN1250`	Windows CP1250	Центральноевропейские	Да	Да	1	
`WIN1251`	Windows CP1251	Кириллица	Да	Да	1	`WIN`
`WIN1252`	Windows CP1252	Западноевропейские	Да	Да	1	
`WIN1253`	Windows CP1253	Греческий	Да	Да	1	
`WIN1254`	Windows CP1254	Турецкий	Да	Да	1	
`WIN1255`	Windows CP1255	Иврит	Да	Да	1	
`WIN1256`	Windows CP1256	Арабский	Да	Да	1	
`WIN1257`	Windows CP1257	Балтийские	Да	Да	1	
`WIN1258`	Windows CP1258	Вьетнамский	Да	Да	1	`ABC`, `TCVN`, `TCVN5712`, `VSCII`

Не все клиентские API поддерживают все перечисленные кодировки. Например, драйвер интерфейса JDBC Postgres Pro не поддерживает `MULE_INTERNAL`, `LATIN6`, `LATIN8` и `LATIN10`.

Поведение кодировки `SQL_ASCII` существенно отличается от других. Когда набором символов сервера является `SQL_ASCII`, сервер интерпретирует байтовые значения 0–127 согласно кодировке ASCII, тогда как значения 128–255 воспринимаются как незначимые. Перекодировка не будет выполнена при выборе `SQL_ASCII`. Таким образом, этот вариант является не столько объявлением того, что используется определённая кодировка, сколько объявлением того, что кодировка игнорируется. В большинстве случаев, если вы работаете с любыми данными, отличными от ASCII, не стоит использовать `SQL_ASCII`, так как Postgres Pro не сможет преобразовать или проверить символы, отличные от ASCII.

22.3.2. Настройка кодировки

`initdb` определяет кодировку по умолчанию для кластера Postgres Pro. Например,

```
initdb -E EUC_JP
```

настраивает кодировку по умолчанию на `EUC_JP` (Расширенная система кодирования для японского языка). Можно использовать `--encoding` вместо `-E` в случае предпочтения более длинных имён параметров. Если параметр `-E` или `--encoding` не задан, `initdb` пытается определить подходящую кодировку в зависимости от указанной или заданной по умолчанию локали.

При создании базы данных можно указать кодировку, отличную от заданной по умолчанию, если эта кодировка совместима с выбранной локалью:

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr korean
```

Это создаст базу данных с именем `korean`, которая использует кодировку `EUC_KR` и локаль `ko_KR`. Также, получить желаемый результат можно с помощью данной SQL-команды:

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr'
 LC_CTYPE='ko_KR.euckr' TEMPLATE=template0;
```

Заметьте, что приведённые выше команды задают копирование базы данных `template0`. При копировании любой другой базы данных, параметры локали и кодировку исходной базы изменить нельзя, так как это может привести к искажению данных. Более подробное описание приведено в Разделе 21.3.

Кодировка базы данных хранится в системном каталоге `pg_database`. Её можно увидеть при помощи параметра `psql -l` или команды `\l`.

```
$ psql -l
                                         List of databases
   Name     |  Owner    | Encoding  |  Collation   |    Ctype     |          Access
 Privileges
-----------+----------+-----------+-------------+-------------
+-------------------------------------
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/
hlinnaka,hlinnaka=CTc/hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/
hlinnaka,hlinnaka=CTc/hlinnaka}
(7 rows)
```

Важно

На большинстве современных операционных систем Postgres Pro может определить, какая кодировка подразумевается параметром `LC_CTYPE`, что обеспечит использование только соответствующей кодировки базы данных. На более старых системах необходимо самостоятельно следить за тем, чтобы использовалась кодировка, соответствующая выбранной языковой среде. Ошибка в этой области, скорее всего, приведёт к странному поведению зависимых от локали операций, таких как сортировка.

Postgres Pro позволит суперпользователям создавать базы данных с кодировкой `SQL_ASCII`, даже когда значение `LC_CTYPE` не установлено в `C` или `POSIX`. Как было сказано выше, `SQL_ASCII` не гарантирует, что данные, хранящиеся в базе, имеют определённую кодировку, и таким образом, этот выбор чреват сбоями, связанными с локалью. Использование данной комбинации устарело и, возможно, будет полностью запрещено.

22.3.3. Автоматическая перекодировка между сервером и клиентом

Postgres Pro поддерживает автоматическое перекодирование символов между сервером и клиентов для многих сочетаний кодировок (они перечисляются в Подразделе 22.3.4).

Чтобы включить автоматическую перекодировку символов, необходимо сообщить Postgres Pro кодировку, которую вы хотели бы использовать на стороне клиента. Это можно выполнить несколькими способами:

- Использование команды `\encoding` в psql. `\encoding` позволяет оперативно изменять клиентскую кодировку. Например, чтобы изменить кодировку на `SJIS`, введите:

  ```
  \encoding SJIS
  ```

- libpq (Раздел 32.11) имеет функции, для управления клиентской кодировкой.
- Использование `SET client_encoding TO`. Клиентская кодировка устанавливается следующей SQL-командой:

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  Также, для этой цели можно использовать стандартный синтаксис SQL `SET NAMES`:

  ```
  SET NAMES 'value';
  ```

  Получить текущую клиентскую кодировку:

  ```
  SHOW client_encoding;
  ```

  Вернуть кодировку по умолчанию:

  ```
  RESET client_encoding;
  ```

- Использование `PGCLIENTENCODING`. Если установлена переменная окружения `PGCLIENTENCODING`, то эта клиентская кодировка выбирается автоматически при подключении к серверу. (В дальнейшем это может быть переопределено при помощи любого из методов, указанных выше.)
- Использование переменной конфигурации client_encoding. Если задана переменная `client_encoding`, указанная клиентская кодировка выбирается автоматически при подключении к серверу. (В дальнейшем это может быть переопределено при помощи любого из методов, указанных выше.)

Если перекодировка определённого символа невозможна (предположим, выбраны `EUC_JP` для сервера и `LATIN1` для клиента, и передаются некоторые японские иероглифы, не представленные в `LATIN1`), возникает ошибка.

Если клиентская кодировка определена как `SQL_ASCII`, перекодировка отключается вне зависимости от кодировки сервера. (Однако если серверная кодировка отлична от `SQL_ASCII`, сервер будет тем не менее проверять, что входящие данные являются допустимыми для его кодировки; поэтому итоговый результат будет тем же, что и при совпадении клиентской кодировки с серверной.) На сервере же использовать кодировку `SQL_ASCII` неразумно, кроме случаев, когда все ваши данные полностью вписываются в ASCII.

22.3.4. Возможные перекодировки наборов символов

Postgres Pro поддерживает перекодирование между любыми двумя наборами символов, для которых в системном каталоге `pg_conversion` присутствует функция перекодирования. Postgres Pro включает несколько предопределённых перекодировок, сведённых в Таблице 22.4 и описанных подробнее в Таблице 22.5. Кроме этого, есть возможность создать новую перекодировку, используя SQL-команду CREATE CONVERSION. (Чтобы она использовалась для автоматического перекодирования текста между сервером и клиентом, она должна быть помечена как перекодировка «по умолчанию» для своей пары кодировок.)

Таблица 22.4. Встроенные клиент-серверные перекодировки наборов символов

Серверная кодировка	Доступные клиентские кодировки
`BIG5`	*не поддерживается как серверная кодировка*
`EUC_CN`	*EUC_CN*, `MULE_INTERNAL` , `UTF8`
`EUC_JP`	*EUC_JP*, `MULE_INTERNAL` , `SJIS`, `UTF8`
`EUC_JIS_2004`	*EUC_JIS_2004*, `SHIFT_JIS_2004` , `UTF8`
`EUC_KR`	*EUC_KR*, `MULE_INTERNAL` , `UTF8`
`EUC_TW`	*EUC_TW*, `BIG5`, `MULE_INTERNAL` , `UTF8`
`GB18030`	*не поддерживается как серверная кодировка*

Серверная кодировка	**Доступные клиентские кодировки**
GBK	*не поддерживается как серверная кодировка*
ISO_8859_5	*ISO_8859_5*, KOI8R, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	*ISO_8859_6*, UTF8
ISO_8859_7	*ISO_8859_7*, UTF8
ISO_8859_8	*ISO_8859_8*, UTF8
JOHAB	*не поддерживается как серверная кодировка*
KOI8R	*KOI8R*, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
KOI8U	*KOI8U*, UTF8
LATIN1	*LATIN1*, MULE_INTERNAL, UTF8
LATIN2	*LATIN2*, MULE_INTERNAL, UTF8, WIN1250
LATIN3	*LATIN3*, MULE_INTERNAL, UTF8
LATIN4	*LATIN4*, MULE_INTERNAL, UTF8
LATIN5	*LATIN5*, UTF8
LATIN6	*LATIN6*, UTF8
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	*не поддерживается как серверная кодировка*
SHIFT_JIS_2004	*не поддерживается как серверная кодировка*
SQL_ASCII	*любая (перекодировка не будет выполнена)*
UHC	*не поддерживается как серверная кодировка*
UTF8	*все поддерживаемые кодировки*
WIN866	*WIN866*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2, MULE_INTERNAL, UTF8
WIN1251	*WIN1251*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

Таблица 22.5. Все встроенные перекодировки наборов символов

Имя преобразования [a]	**Исходная кодировка**	**Целевая кодировка**
big5_to_euc_tw	BIG5	EUC_TW

Имя преобразования [a]	**Исходная кодировка**	**Целевая кодировка**
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL

Имя преобразования [a]	**Исходная кодировка**	**Целевая кодировка**
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4

Имя преобразования [a]	**Исходная кодировка**	**Целевая кодировка**
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8

Имя преобразования [a]	**Исходная кодировка**	**Целевая кодировка**
`utf8_to_shift_jis_2004`	`UTF8`	`SHIFT_JIS_2004`
`euc_jis_2004_to_shift_jis_2004`	`EUC_JIS_2004`	`SHIFT_JIS_2004`
`shift_jis_2004_to_euc_jis_2004`	`SHIFT_JIS_2004`	`EUC_JIS_2004`

[a] Имена преобразований следуют стандартной схеме именования. К официальному названию исходной кодировки, в котором все не алфавитно-цифровые символы заменяются подчёркиваниями, добавляется `_to_`, а за ним аналогично подготовленное имя целевой кодировки. Таким образом, эти имена иногда не совпадают буквально с общепринятыми названиями, показанными в Таблице 22.3.

22.3.5. Дополнительные источники информации

Рекомендуемые источники для начала изучения различных видов систем кодирования.

Обработка информации на китайском, японском, корейском & вьетнамском языках.

Содержит подробные объяснения по `EUC_JP`, `EUC_CN`, `EUC_KR`, `EUC_TW`.

https://www.unicode.org/

Сайт Unicode Consortium.

RFC 3629

UTF-8 (формат преобразования 8-битного UCS/Unicode) определён здесь.

Глава 23. Регламентные задачи обслуживания базы данных

Как и в любой СУБД, в Postgres Pro для достижения оптимальной производительности нужно регулярно выполнять определённые процедуры. Задачи, которые рассматриваются в этой главе, являются *обязательными*, но они по природе своей повторяющиеся и легко поддаются автоматизации с использованием стандартных средств, таких как задания cron или Планировщика задач в Windows. Создание соответствующих заданий и контроль над их успешным выполнением входят в обязанности администратора базы данных.

Одной из очевидных задач обслуживания СУБД является регулярное создание резервных копий данных. При отсутствии свежей резервной копии у вас не будет шанса восстановить систему после катастрофы (сбой диска, пожар, удаление важной таблицы по ошибке и т. д.). Механизмы резервного копирования и восстановления в Postgres Pro детально рассматриваются в Главе 24.

Другое важное направление обслуживания СУБД — периодическая «очистка» базы данных. Эта операция рассматривается в Разделе 23.1. С ней тесно связано обновление статистики, которая будет использоваться планировщиком запросов; оно рассматривается в Подразделе 23.1.3.

Ещё одной задачей, требующей периодического выполнения, является управление файлами журнала. Она рассматривается в Разделе 23.3.

Для контроля состояния базы данных и для отслеживания нестандартных ситуаций можно использовать *check_postgres*. Скрипт check_postgres можно интегрировать с Nagios и MRTG, однако он может работать и самостоятельно.

По сравнению с некоторыми другими СУБД Postgres Pro неприхотлив в обслуживании. Тем не менее должное внимание к вышеперечисленным задачам будет значительно способствовать комфортной и производительной работе с СУБД.

23.1. Регламентная очистка

Базы данных Postgres Pro требуют периодического проведения процедуры обслуживания, которая называется *очисткой*. Во многих случаях очистку достаточно выполнять с помощью *демона автоочистки*, который описан в Подразделе 23.1.6. Возможно, в вашей ситуации для получения оптимальных результатов потребуется настроить описанные там же параметры автоочистки. Некоторые администраторы СУБД могут дополнить или заменить действие этого демона командами `VACUUM` (обычно они выполняются по расписанию в заданиях cron или Планировщика задач). Чтобы правильно организовать очистку вручную, необходимо понимать темы, которые будут рассмотрены в следующих подразделах. Администраторы, которые полагаются на автоочистку, возможно, всё же захотят просмотреть этот материал, чтобы лучше понимать и настраивать эту процедуру.

23.1.1. Основные принципы очистки

Команды `VACUUM` в Postgres Pro должны обрабатывать каждую таблицу по следующим причинам:

1. Для высвобождения или повторного использования дискового пространства, занятого изменёнными или удалёнными строками.
2. Для обновления статистики по данным, используемой планировщиком запросов Postgres Pro.
3. Для обновления карты видимости, которая ускоряет сканирование только индекса.
4. Для предотвращения потери очень старых данных из-за зацикливания идентификаторов *транзакций* или *мультитранзакций*.

Разные причины диктуют выполнение действий `VACUUM` с разной частотой и в разном объёме, как рассматривается в следующих подразделах.

Существует два варианта `VACUUM`: обычный `VACUUM` и `VACUUM FULL`. Команда `VACUUM FULL` может высвободить больше дискового пространства, однако работает медленнее. Кроме того, обычная команда `VACUUM` может выполняться параллельно с использованием производственной базы данных.

(При этом такие команды как `SELECT`, `INSERT`, `UPDATE` и `DELETE` будут выполняться нормально, хотя нельзя будет изменить определение таблицы командами типа `ALTER TABLE`.) Команда `VACUUM FULL` требует блокировки обрабатываемой таблицы в режиме `ACCESS EXCLUSIVE` и поэтому не может выполняться параллельно с другими операциями с этой таблицей. По этой причине администраторы, как правило, должны стараться использовать обычную команду `VACUUM` и избегать `VACUUM FULL`.

Команда `VACUUM` порождает существенный объём трафика ввода-вывода, который может стать причиной низкой производительности в других активных сеансах. Это влияние фоновой очистки можно регулировать, настраивая параметры конфигурации (см. Подраздел 18.4.4).

23.1.2. Высвобождение дискового пространства

В Postgres Pro команды `UPDATE` или `DELETE` не вызывают немедленного удаления старой версии изменяемых строк. Этот подход необходим для реализации эффективного многоверсионного управления конкурентным доступом (MVCC, см. Главу 13): версия строки не должна удаляться до тех пор, пока она остаётся потенциально видимой для других транзакций. Однако в конце концов устаревшая или удалённая версия строки оказывается не нужна ни одной из транзакций. После этого занимаемое ей место должно быть освобождено и может быть отдано новым строкам, во избежание неограниченного роста потребности в дисковом пространстве. Это происходит при выполнении команды `VACUUM`.

Обычная форма `VACUUM` удаляет неиспользуемые версии строк в таблицах и индексах и помечает пространство свободным для дальнейшего использования. Однако это дисковое пространство не возвращается операционной системе, кроме особого случая, когда полностью освобождаются одна или несколько страниц в конце таблицы и можно легко получить исключительную блокировку таблицы. Команда `VACUUM FULL`, напротив, кардинально сжимает таблицы, записывая абсолютно новую версию файла таблицы без неиспользуемого пространства. Это минимизирует размер таблицы, однако может занять много времени. Кроме того, для этого требуется больше места на диске для записи новой копии таблицы до завершения операции.

Обычно цель регулярной очистки — выполнять простую очистку (`VACUUM`) достаточно часто, чтобы не возникала необходимость в `VACUUM FULL`. Демон автоочистки пытается работать в этом режиме, и на самом деле он сам никогда не выполняет `VACUUM FULL`. Основная идея такого подхода не в том, чтобы минимизировать размер таблиц, а в том, чтобы поддерживать использование дискового пространства на стабильном уровне: каждая таблица занимает объём, равный её минимальному размеру, плюс объём, который был занят между процедурами очистки. Хотя с помощью `VACUUM FULL` можно сжать таблицу до минимума и возвратить дисковое пространство операционной системе, большого смысла в этом нет, если в будущем таблица так же вырастет снова. Следовательно, для активно изменяемых таблиц лучше с умеренной частотой выполнять `VACUUM`, чем очень редко выполнять `VACUUM FULL`.

Некоторые администраторы предпочитают планировать очистку БД самостоятельно, например, проводя все работы ночью в период низкой загрузки. Однако очистка только по фиксированному расписанию плоха тем, что при резком скачке интенсивности изменений раздувание таблицы может привести к тому, что для высвобождения пространства действительно понадобится выполнить `VACUUM FULL`. Использование демона автоочистки снимает эту проблему, поскольку он планирует очистку динамически, отслеживая интенсивность изменений. Полностью отключать этот демон может иметь смысл, только если вы имеете дело с предельно предсказуемой загрузкой. Возможен и компромиссный вариант — настроить параметры демона автоочистки так, чтобы он реагировал только на необычайно высокую интенсивность изменений и мог удержать ситуацию под контролем, в то время как команды `VACUUM`, запускаемые по расписанию, будут выполнять основную работу в периоды нормальной загрузки.

Если же автоочистка не применяется, обычно планируется выполнение `VACUUM` для всей базы данных раз в сутки в период низкой активности, и в случае необходимости оно дополняется более частой очисткой интенсивно изменяемых таблиц. (В некоторых ситуациях, когда изменения производятся крайне интенсивно, самые востребованные таблицы могут очищаться раз в несколько минут.) Если в вашем кластере несколько баз данных, не забывайте выполнять `VACUUM` для каждой из них; при этом может быть полезна программа vacuumdb.

Подсказка

Результат обычного `VACUUM` может быть неудовлетворительным, когда вследствие массового изменения или удаления в таблице оказывается много мёртвых версий строк. Если у вас есть такая таблица и вам нужно освободить лишнее пространство, которое она занимает, используйте команду `VACUUM FULL` или, в качестве альтернативы, `CLUSTER` или один из вариантов `ALTER TABLE`, выполняющий перезапись таблицы. Эти команды записывают абсолютно новую копию таблицы и строят для неё индексы. Все эти варианты требуют блокировки в режиме `ACCESS EXCLUSIVE`. Заметьте, что они также на время требуют дополнительного пространства на диске в объёме, приблизительно равном размеру таблицы, поскольку старые копии таблицы и индексов нельзя удалить до завершения создания новых копий.

Подсказка

Если у вас есть таблица, всё содержимое которой периодически нужно удалять, имеет смысл делать это, выполняя только `TRUNCATE`, а не `DELETE` и затем `VACUUM`. `TRUNCATE` немедленно удаляет всё содержимое таблицы, не требуя последующей очистки (`VACUUM` или `VACUUM FULL`) для высвобождения неиспользуемого дискового пространства. Недостатком такого подхода является нарушение строгой семантики MVCC.

23.1.3. Обновление статистики планировщика

Планировщик запросов в Postgres Pro, выбирая эффективные планы запросов, полагается на статистическую информацию о содержимом таблиц. Эта статистика собирается командой `ANALYZE`, которая может вызываться сама по себе или как дополнительное действие команды `VACUUM`. Статистика должна быть достаточно точной, так как в противном случае неудачно выбранные планы запросов могут снизить производительность базы данных.

Демон автоочистки, если он включён, будет автоматически выполнять `ANALYZE` после существенных изменений содержимого таблицы. Однако администраторы могут предпочесть выполнение `ANALYZE` вручную, в частности, если известно, что производимые в таблице изменения не повлияют на статистику по «интересным» столбцам. Демон же планирует выполнение `ANALYZE` в зависимости только от количества вставленных или изменённых строк; он не знает, приведут ли они к значимым изменениям статистики.

Изменение кортежей в дочерних таблицах и секциях не считается поводом для запуска анализа родительской таблицы. Если родительская таблица пуста или изменяется редко, процесс автоочистки может никогда не затронуть её и статистика иерархии наследования собираться не будет. Поэтому необходимо выполнять `ANALYZE` для родительской таблицы вручную, чтобы поддерживать статистику в актуальном состоянии.

Как и процедура очистки для высвобождения пространства, частое обновление статистики полезнее для интенсивно изменяемых таблиц, нежели для тех таблиц, которые изменяются редко. Однако даже в случае часто изменяемой таблицы обновление статистики может не требоваться, если статистическое распределение данных меняется слабо. Как правило, достаточно оценить, насколько меняются максимальное и минимальное значения в столбцах таблицы. Например, максимальное значение в столбце `timestamp`, хранящем время изменения строки, будет постоянно увеличиваться по мере добавления и изменения строк; для такого столбца может потребоваться более частое обновление статистики, чем, к примеру, для столбца, содержащего адреса страниц (URL), которые запрашивались с сайта. Столбец с URL-адресами может меняться столь же часто, однако статистическое распределение его значений, вероятно, будет изменяться относительно медленно.

Команду `ANALYZE` можно выполнять для отдельных таблиц и даже просто для отдельных столбцов таблицы, поэтому, если того требует приложение, одни статистические данные можно обновлять чаще, чем другие. Однако на практике обычно лучше просто анализировать всю базу данных, поскольку это быстрая операция, так как `ANALYZE` читает не каждую отдельную строку, а статистически случайную выборку строк таблицы.

Подсказка

Хотя индивидуальная настройка частоты `ANALYZE` для отдельных столбцов может быть не очень полезной, смысл может иметь настройка детализации статистики, собираемой командой `ANALYZE`. Для столбцов, которые часто используются в предложениях `WHERE`, и имеют очень неравномерное распределение данных, может потребоваться более детальная, по сравнению с другими столбцами, гистограмма данных. В таких случаях можно воспользоваться командой `ALTER TABLE SET STATISTICS` или изменить значение по умолчанию параметра уровня БД default_statistics_target.

Кроме того, по умолчанию информация об избирательности функций ограничена. Однако если вы создаёте объект статистики или индекс по выражению с вызовом функции, об этой функции будет собрана полезная статистическая информация, которая может значительно улучшить планы запросов, задействующие данный индекс.

Подсказка

Демон автоочистки не выполняет команды `ANALYZE` для сторонних таблиц, поскольку он не знает, как часто это следует делать. Если для получения качественных планов вашим запросам необходима статистика по сторонним таблицам, будет хорошей идеей дополнительно запускать `ANALYZE` для них по подходящему расписанию.

Подсказка

Демон автоочистки не выполняет команды `ANALYZE` для секционированных таблиц. В иерархии наследования анализ родительской таблицы проводится, только если меняются данные в ней самой — изменения в дочерней таблице не считаются поводом для автоанализа родительской таблицы. Если для получения качественных планов вашим запросам необходима статистика по родительским таблицам, нужно периодически выполнять `ANALYZE` вручную, чтобы их статистика была актуальной.

23.1.4. Обновление карты видимости

Процедура очистки поддерживает карты видимости для каждой таблицы, позволяющие определить, в каких страницах есть только записи, заведомо видимые для всех активных транзакций (и всех будущих транзакций, пока страница не будет изменена). Это имеет два применения. Во-первых, сам процесс очистки может пропускать такие страницы при следующем запуске, поскольку на этих страницах вычищать нечего.

Во-вторых, с такими картами Postgres Pro может выдавать результаты некоторых запросов, используя только индекс, не обращаясь к данным таблицы. Так как индексы Postgres Pro не содержат информацию о видимости записей, при обычном сканировании по индексу необходимо извлечь соответствующую запись из таблицы и проверить её видимость для текущей транзакции. Поэтому при *сканировании только индекса*, наоборот, сначала проверяется карта видимости. Если известно, что все записи на странице видимы, то выборку из таблицы можно пропустить. Это наиболее полезно с большими наборами данных, когда благодаря карте видимости можно оптимизировать чтение с диска. Карта видимости значительно меньше таблицы, поэтому она легко помещается в кеш, даже когда объём самих страниц очень велик.

23.1.5. Предотвращение ошибок из-за зацикливания счётчика транзакций

В Postgres Pro семантика транзакций MVCC зависит от возможности сравнения номеров идентификаторов транзакций (XID): версия строки, у которой XID добавившей её транзакции больше, чем XID текущей транзакции, относится «к будущему» и не должна быть видна в текущей транзакции. Однако поскольку идентификаторы транзакций имеют ограниченный размер (32 бита), кластер, работающий долгое время (более 4 миллиардов транзакций) столкнётся с *зацикливанием идентификаторов транзакций*: счётчик XID прокрутится до нуля, и внезапно транзакции, которые относились к прошлому, окажутся в будущем — это означает, что их результаты станут невидимыми. Одним словом, это катастрофическая потеря данных. (На самом деле данные никуда не пропадают, однако если вы не можете их получить, то это слабое утешение.) Для того чтобы этого избежать, необходимо выполнять очистку для каждой таблицы в каждой базе данных как минимум единожды на два миллиарда транзакций.

Периодическое выполнение очистки решает эту проблему, потому что процедура `VACUUM` помечает строки как *замороженные*, указывая, что они были вставлены транзакцией, зафиксированной достаточно давно, так что эффект добавляющей транзакции с точки зрения MVCC определённо будет виден во всех текущих и будущих транзакциях. Обычные значения XID сравниваются по модулю 2^{32}. Это означает, что для каждого обычного XID существуют два миллиарда значений XID, которые «старше» него, и два миллиарда значений, которые «младше» него; другими словами, пространство значений XID циклично и не имеет конечной точки. Следовательно, как только создаётся версия строки с обычным XID, для следующих двух миллиардов транзакций эта версия строки оказывается «в прошлом», неважно о каком значении обычного XID идет речь. Если после двух миллиардов транзакций эта версия строки всё ещё существует, она внезапно окажется в будущем. Для того чтобы это предотвратить, в какой-то момент значение XID для старых версий строк должно быть заменено на `FrozenTransactionId` (заморожено) до того, как будет достигнута граница в два миллиарда транзакций. После получения этого особенного XID для всех обычных транзакций эти версии строк будут относиться «к прошлому», независимо от зацикливания, и, таким образом, эти версии строк будут действительны до момента их удаления, когда бы это ни произошло.

Примечание

В версиях PostgreSQL до 9.4 замораживание было реализовано как замена XID добавления строки специальным идентификатором `FrozenTransactionId`, который можно было увидеть в системной колонке `xmin` данной строки. В новых версиях просто устанавливается битовый флаг, а исходный `xmin` строки сохраняется для возможного расследования в будущем. Однако строки с `xmin`, равным `FrozenTransactionId` (2), можно по-прежнему встретить в базах данных, обновлённых (с применением pg_upgrade) с версий до 9.4.

Также системные каталоги могут содержать строки со значением `xmin`, равным `BootstrapTransactionId` (1), показывающим, что они были вставлены на первом этапе initdb. Как и `FrozenTransactionId`, этот специальный XID считается более старым, чем любой обычный XID.

Параметр vacuum_freeze_min_age определяет, насколько старым должен стать XID, чтобы строки с таким XID были заморожены. Увеличение его значения помогает избежать ненужной работы, если строки, которые могли бы быть заморожены в ближайшее время, будут изменены ещё раз, а уменьшение приводит к увеличению количества транзакций, которые могут выполниться, прежде чем потребуется очередная очистка таблицы.

`VACUUM` определяет, какие страницы таблицы нужно сканировать, анализируя карту видимости. Обычно при этой операции пропускаются страницы, в которых нет мёртвых версий строк, даже если в них могут быть версии строк со старыми XID. Таким образом, обычная команда `VACUUM` не всегда замораживает все старые версии строк, имеющиеся в таблице. Когда это имеет место, операция `VACUUM` в конце концов должна провести *агрессивную очистку* и заморозить все подходящие незамороженные значения XID и MXID, в том числе на полностью видимых, но не полностью замороженных страницах. На практике периодическая агрессивная очистка требуется для большинства таблиц. Когда `VACUUM` будет делать это — зависит от параметра vacuum_freeze_table_age: полностью видимые, но не полностью замороженные страницы будут сканироваться, если число транзакций, прошедших со времени последнего такого сканирования, оказывается больше чем `vacuum_freeze_table_age` минус `vacuum_freeze_min_age`. Если `vacuum_freeze_table_age` равно 0, `VACUUM` будет применять агрессивную стратегию всегда.

Максимальное время, в течение которого таблица может обходиться без очистки, составляет два миллиарда транзакций минус значение `vacuum_freeze_min_age` с момента последней агрессивной очистки. Если бы таблица не подвергалась очистке дольше, была бы возможна потеря данных. Чтобы гарантировать, что это не произойдёт, для любой таблицы, которая может содержать значения XID старше, чем возраст, указанный в конфигурационном параметре autovacuum_freeze_max_age, вызывается автоочистка. (Это случится, даже если автоочистка отключена.)

Это означает, что если очистка таблицы не вызывается другим способом, то автоочистка для неё будет вызываться приблизительно через каждые `autovacuum_freeze_max_age` минус `vacuum_freeze_min_age` транзакций. Для таблиц, очищаемых регулярно для высвобождения пространства, это неактуально. В то же время статичные таблицы (включая таблицы, в которых данные вставляются, но не изменяются и не удаляются) не нуждаются в очистке для высвобождения пространства, поэтому для очень больших статичных таблиц имеет смысл увеличить интервал между вынужденными запусками автоочистки. Очевидно, это можно сделать, либо увеличив `autovacuum_freeze_max_age`, либо уменьшив `vacuum_freeze_min_age`.

Фактический максимум для `vacuum_freeze_table_age` составляет 0.95 * `autovacuum_freeze_max_age`; большее значение будет ограничено этим пределом. Значение, превышающее `autovacuum_freeze_max_age`, не имело бы смысла, поскольку по достижении этого значения в любом случае вызывалась бы автоочистка для предотвращения зацикливания, а коэффициент 0.95 оставляет немного времени для того, чтобы запустить команду `VACUUM` вручную до того, как это произойдёт. Как правило, установленное значение `vacuum_freeze_table_age` должно быть несколько меньше `autovacuum_freeze_max_age`, чтобы оставленный промежуток был достаточен для выполнения в этом окне `VACUUM` по расписанию или автоочистки, управляемой обычной активностью операций удаления и изменения. Если это значение будет слишком близким к максимуму, автоочистка для предотвращения зацикливания будет выполняться, даже если таблица только что была очищена для высвобождения пространства, в то же время при небольшом значении будет чаще производиться агрессивная очистка.

Единственный минус увеличения `autovacuum_freeze_max_age` (и `vacuum_freeze_table_age` с ним) заключается в том, что подкаталоги `pg_xact` и `pg_commit_ts` в кластере баз данных будут занимать больше места, поскольку в них нужно будет хранить статус и (при включённом `track_commit_timestamp`) время фиксации всех транзакций вплоть до горизонта `autovacuum_freeze_max_age`. Для статуса фиксации используется по два бита на транзакцию, поэтому если в `autovacuum_freeze_max_age` установлено максимально допустимое значение в два миллиарда, то размер `pg_xact` может составить примерно половину гигабайта, а `pg_commit_ts` примерно 20 ГБ. Если по сравнению с объёмом вашей базы данных этот объём незначителен, тогда рекомендуется установить для `autovacuum_freeze_max_age` максимально допустимое значение. В противном случае установите значение этого параметра в зависимости от объёма, который вы готовы выделить для `pg_xact` и `pg_commit_ts`. (Значению по умолчанию, 200 миллионам транзакций, соответствует приблизительно 50 МБ в `pg_xact` и около 2 ГБ в `pg_commit_ts`.)

Уменьшение значения `vacuum_freeze_min_age`, с другой стороны, чревато тем, что команда `VACUUM` может выполнять бесполезную работу: замораживание версии строки — пустая трата времени, если эта строка будет вскоре изменена (и в результате получит новый XID). Поэтому значение этого параметра должно быть достаточно большим для того, чтобы строки не замораживались, пока их последующее изменение не станет маловероятным.

Для отслеживания возраста самых старых незамороженных значений XID в базе данных команда `VACUUM` сохраняет статистику по XID в системных таблицах `pg_class` и `pg_database`. В частности, столбец `relfrozenxid` в записи для определённой таблицы в `pg_class` содержит самое старое оставшееся незамороженное значение XID на конец последней операции `VACUUM`, которая увеличила значение `relfrozenxid` (обычно это последняя операция агрессивной очистки). Аналогично столбец `datfrozenxid` в записи для базы данных в `pg_database` представляет нижнюю границу обычных значений XID, встречающихся в этой базе, — он просто хранит минимальное из всех значений `relfrozenxid` для таблиц этой базы. Эту информацию удобно получать с помощью таких запросов:

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

Столбец `age` показывает количество транзакций от граничного значения XID до XID текущей транзакции.

Подсказка

Если для команды `VACUUM` указан параметр `VERBOSE`, `VACUUM` выводит различную статистику по таблице, в том числе информацию об увеличении значений `relfrozenxid` и `relminmxid`, а также о количестве новых замороженных страниц. Те же сведения вносятся в журнал сервера, когда в него записываются сообщения об операции `VACUUM`, выполненной процессом автоочистки (этим управляет параметр log_autovacuum_min_duration).

Обычно `VACUUM` сканирует только те страницы, которые изменялись после последней очистки, однако `relfrozenxid` может увеличиться только при сканировании всех страниц таблицы, включая те, что могут содержать незамороженные XID. Это происходит, когда возраст `relfrozenxid` превышает `vacuum_freeze_table_age` транзакций, когда `VACUUM` вызывается с указанием `FREEZE`, или когда оказывается, что очистку для удаления мёртвых версий строк нужно провести во всех ещё не замороженных страницах. Когда `VACUUM` сканирует в таблице каждую ещё не полностью замороженную страницу, значение `age(relfrozenxid)` в результате должно стать немного больше, чем установленное значение `vacuum_freeze_min_age` (больше на число транзакций, начатых с момента запуска `VACUUM`). `VACUUM` установит для `relfrozenxid` значение, равное самому старому XID, остающемуся в таблице, поэтому возможно, что окончательное значение будет новее, чем требовалось. Если по достижении `autovacuum_freeze_max_age` для таблицы ни разу не будет увеличен `relfrozenxid`, в скором времени для неё будет принудительно запущена автоочистка.

Если по какой-либо причине автоочистка не может вычистить старые значения XID из таблицы, система начинает выдавать предупреждающие сообщения, подобные приведённому ниже, когда самое старое значение XID в базе данных оказывается в сорока миллионах транзакций от точки зацикливания:

```
ПРЕДУПРЕЖДЕНИЕ:  база данных "mydb" должна быть очищена (предельное число транзакций:
 39985967)
ПОДСКАЗКА:  Во избежание ошибок назначения XID выполните очистку (VACUUM) всей базы.
```

(Проблему можно решить, как предлагает подсказка, запустив `VACUUM` вручную; однако учтите, что выполнять `VACUUM` должен суперпользователь, в противном случае эта процедура не сможет обработать системные каталоги и, следовательно, не сможет увеличить значение `datfrozenxid` для базы данных.) Если эти предупреждения игнорировать, система отключится и не будет начинать никаких транзакций, как только до точки зацикливания останется менее трёх миллионов транзакций:

```
ОШИБКА:  база данных не принимает команды, назначающие новые XID, во избежание потери
 данных из-за зацикливания в БД "mydb"
ПОДСКАЗКА:  Выполните очистку (VACUUM) для этой базы данных.
```

В этом состоянии любые уже выполняемые транзакции могут продолжаться, но могут быть запущены лишь транзакции только для чтения. Операции, изменяющие записи базы данных или удаляющие отношения, завершатся ошибкой. Команду `VACUUM` по-прежнему можно запускать в обычном режиме. Обратите внимание, что вопреки некоторым рекомендациям в прошлых релизах, нет необходимости или даже нежелательно останавливать управляющий процесс postmaster или входить в однопользовательский режим, чтобы восстановить нормальную работу. Вместо этого выполните следующие действия:

1. Разберите старые подготовленные транзакции. Их можно найти, проверив pg_prepared_xacts на наличие строк с большим значением `age(transactionid)`. Такие транзакции следует фиксировать или отменять.
2. Завершите длительные открытые транзакции. Их можно найти, проверив pg_stat_activity на наличие строк с большим значением `age(backend_xid)` или `age(backend_xmin)`. Такие транзакции следует фиксировать или отменять, либо можно завершить сеанс с помощью `pg_terminate_backend`.
3. Удалите все старые слоты репликации. Используйте pg_stat_replication, чтобы найти слоты с большим значением `age(xmin)` или `age(catalog_xmin)`. Во многих случаях такие слоты создавались для репликации на серверы, которых уже нет или которые давно не работают. Если удалить слот для сервера, который всё ещё существует и может по-прежнему пытаться подключиться к этому слоту, возможно, эту реплику придётся пересоздать.
4. Выполните `VACUUM` в целевой базе данных. Проще всего использовать `VACUUM` для всей базы данных. Чтобы сократить время выполнения, также можно вручную выполнить команды `VACUUM` для таблиц с самым старым `relminxid`. Не используйте `VACUUM FULL` в этом сценарии, поскольку для него требуется XID и, следовательно, произойдёт сбой, за исключением режима суперпользователя, где напротив будет обрабатываться XID и, таким образом, увеличится риск зацикливания счётчика идентификатора транзакции. Не используйте `VACUUM FREEZE`, поскольку при этом выполнится объём работы, который будет больше минимально необходимого для восстановления нормального функционирования.
5. После восстановления нормальной работы проверьте, что автоочистка правильно настроена в целевой базе данных, чтобы избежать проблем в будущем.

Примечание

В более ранних версиях иногда было необходимо останавливать процесс postmaster и выполнять `VACUUM` для базы данных в однопользовательском режиме. В типичных сценариях в этом больше нет необходимости, поэтому по возможности следует избегать этих действий, поскольку они могут привести к сбою системы и повышают риски, так как отключают защиту от зацикливания идентификатора транзакции, предназначенную для предотвращения потери данных. Использовать однопользовательский режим в этом сценарии следует лишь при желании выполнить `TRUNCATE` или `DROP` ненужных таблиц, чтобы избежать необходимости выполнять для них `VACUUM`. Резерв в три миллиона транзакций позволяет администратору это сделать. За подробной информацией об использовании однопользовательского режима обратитесь к странице справки по postgres.

23.1.5.1. Мультитранзакции и зацикливание

Идентификаторы мультитранзакций используются для поддержки блокировки строк несколькими транзакциями одновременно. Поскольку в заголовке строки есть только ограниченное пространство для хранения информации о блокировках, в нём указывается «идентификатор множественной транзакции», или идентификатор мультитранзакции для краткости, когда строку блокируют одновременно несколько транзакций. Информация о том, какие именно идентификаторы транзакций относятся к определённой мультитранзакции, хранится отдельно в подкаталоге `pg_multixact`, а в поле `xmax` в заголовке строки сохраняется только идентификатор мультитранзакции. Как и идентификаторы транзакций, идентификаторы мультитранзакций исполнены в виде 32-разрядного счётчика и хранятся аналогично, что требует аккуратного управления их возрастом, очисткой хранилища и предотвращением зацикливаний. Существует отдельная область, в которой содержится список членов каждой мультитранзакции, где счётчики также 32-битные и требуют должного контроля. Чтобы узнать идентификаторы транзакции, связанные с идентификатором мультитранзакции, можно воспользоваться системной функцией `pg_get_multixact_members()`, описанной в Таблице 9.82.

Когда `VACUUM` сканирует какую-либо часть таблицы, каждый идентификатор мультитранзакции старее чем vacuum_multixact_freeze_min_age заменяется другим значением, которое может быть нулевым, идентификатором одиночной транзакции или новым идентификатором мультитранзакции. Для каждой таблицы в `pg_class.relminmxid` хранится самый старый возможный идентификатор мультитранзакции, всё ещё задействованный в какой-либо строке этой таблицы. Если это значение оказывается старее vacuum_multixact_freeze_table_age, выполняется агрессивная очистка. Как рассказывалось в предыдущем разделе, при агрессивной очистке будут пропускаться только те страницы, которые считаются полностью замороженными. Узнать возраст `pg_class.relminmxid` можно с помощью функции `mxid_age()`.

Благодаря агрессивным операциям `VACUUM`, вне зависимости от их причины, *гарантируется* увеличение значения `relminmxid` для таблицы. В конце концов, по мере сканирования всех таблиц во всех базах данных и увеличения их старейших значений мультитранзакций, информация о старых мультитранзакциях может быть удалена с диска.

В качестве меры защиты, агрессивное сканирование с целью очистки будет происходить для любой таблицы, возраст мультитранзакций которой больше, чем autovacuum_multixact_freeze_max_age. Агрессивное сканирование также будет выполняться чаще со всеми таблицами, начиная с имеющих старейшие мультитранзакции, если объём занятой области членов мультитранзакций превышает примерно 10 ГБ. Эти два варианта агрессивного сканирования осуществляются, даже если процесс автоочистки отключён. Область хранения членов мультитранзакций может увеличиться до примерно 20 ГБ, прежде чем начнётся зацикливание.

Как и в случае с XID, если при автоочистке не удаётся удалить старые MXID из таблицы, система начнет выдавать сообщения с предупреждением, когда самые старые MXID базы данных достигнут сорока миллионов транзакций от точки зацикливания. И снова, как и в случае с XID, если эти предупреждения игнорировать, система перестанет генерировать новые MXID, если до точки зацикливания останется менее трёх миллионов транзакций.

Когда заканчиваются MXID, нормальную работу можно восстановить практически так же, как когда заканчиваются XID. Выполните те же действия, что и в предыдущем разделе, но со следующими отличиями:

1. Запущенные транзакции и подготовленные транзакции можно игнорировать, если они точно не могут появиться в мультитранзакции.
2. Информация о MXID не отображается напрямую в системных представлениях, таких как `pg_stat_activity`, однако поиск старых XID по-прежнему позволяет эффективно определить, какие транзакции вызывают проблемы с зацикливанием MXID.
3. Когда заканчиваются XID, блокируются все пишущие транзакции, а когда заканчиваются MXID, блокируется только подмножество пишущих транзакций, а именно те, которые включают блокировки строк, требующие MXID.

23.1.6. Демон автоочистки

В Postgres Pro имеется не обязательная, но настоятельно рекомендуемая к использованию функция, называемая *автоочисткой*, предназначение которой — автоматизировать выполнение команд `VACUUM` и `ANALYZE`. Когда автоочистка включена, она проверяет, в каких таблицах было вставлено, изменено или удалено много строк. При этих проверках используются средства сбора статистики; поэтому автоочистка будет работать, только если параметр track_counts имеет значение `true`. В конфигурации по умолчанию автоочистка включена и соответствующие параметры имеют подходящие значения.

«Демон автоочистки» на самом деле состоит из нескольких процессов. Существует постоянный фоновый процесс, называемый *процессом запуска автоочистки*, который отвечает за запуск *рабочих процессов автоочистки* для всех баз данных. Этот контролирующий процесс распределяет работу по времени, стараясь запускать рабочий процесс для каждой базы данных каждые autovacuum_naptime секунд. (Следовательно, если всего имеется *N* баз данных, новый рабочий процесс будет запускаться каждые `autovacuum_naptime`/*N* секунд.) Одновременно могут выполняться до autovacuum_max_workers рабочих процессов. Если число баз данных, требующих обработки, превышает `autovacuum_max_workers`, обработка следующей базы начинается сразу по завершении первого рабочего процесса. Каждый рабочий процесс проверяет все таблицы в своей базе данных и в случае необходимости выполняет `VACUUM` и/или `ANALYZE`. Для отслеживания действий рабочих процессов можно установить параметр log_autovacuum_min_duration.

Если в течение короткого промежутка времени потребность в очистке возникает для нескольких больших таблиц, все рабочие процессы автоочистки могут продолжительное время заниматься очисткой только этих таблиц. В результате другие таблицы и базы данных будут ожидать очистки, пока не появится свободный рабочий процесс. Число рабочих процессов для одной базы не ограничивается, при этом каждый процесс старается не повторять работу, только что выполненную другими. Заметьте, что в ограничениях max_connections или superuser_reserved_connections число выполняющихся рабочих процессов не учитывается.

Для таблиц с `relfrozenxid`, устаревшим более чем на autovacuum_freeze_max_age транзакций, очистка выполняется всегда (это также применимо к таблицам, для которых максимальный порог заморозки был изменён через параметры хранения; см. ниже). В противном случае очистка таблицы производится, если количество кортежей, устаревших с момента последнего выполнения `VACUUM`, превышает «пороговое значение очистки». Пороговое значение очистки определяется как:

```
порог очистки = базовый порог очистки + коэффициент доли для очистки * количество
 кортежей
```

где базовый порог очистки — значение autovacuum_vacuum_threshold, коэффициент доли — autovacuum_vacuum_scale_factor, а количество кортежей — `pg_class.reltuples`.

Таблица также очищается, если число кортежей, добавленных после предыдущей очистки, превышает установленный порог очистки при добавлении, который определяется так:

```
порог очистки при добавлении = базовый порог очистки при добавлении + коэффициент доли
 для очистки при добавлении * количество кортежей
```

Базовый порог очистки при добавлении и коэффициент доли для очистки при добавлении определяются параметрами autovacuum_vacuum_insert_threshold и autovacuum_vacuum_insert_scale_factor, соответственно. При такой очистке часть страниц таблицы могут быть помечены как *полностью видимые* и могут быть также заморожены кортежи, что уменьшит объём работы, которую нужно будет проделать при следующей очистке. Для таблиц, в которых выполняются в основном операции `INSERT` и практически не выполняются `UPDATE`/`DELETE`, может иметь смысл уменьшить параметр таблицы autovacuum_freeze_min_age, так как это позволит замораживать кортежи раньше. Количество устаревших и добавленных кортежей получается от системы накопительной статистики; оно представляет собой согласованное в конечном счёте число, обновляемое после каждой операции `UPDATE`, `DELETE` и `INSERT`. Если значение `relfrozenxid` для таблицы старее `vacuum_freeze_table_age` транзакций, производится агрессивная очистка с целью заморозить старые версии строк и увеличить значение `relfrozenxid`; в противном случае сканируются только страницы, изменённые после последней очистки.

Для выполнения сбора статистики используется аналогичное условие: пороговое значение, определяемое как:

```
порог анализа = базовый порог анализа + коэффициент доли для анализа * количество
 кортежей
```

сравнивается с общим количеством кортежей добавленных, изменённых или удалённых после последнего выполнения `ANALYZE`.

В секционированных таблицах кортежи не хранятся напрямую и, следовательно, не обрабатываются автоочисткой. (Автоочистка обрабатывает секции таблицы так же, как и другие таблицы.) К сожалению, это означает, что автоочистка не запускает `ANALYZE` для секционированных таблиц, в результате чего создаются неоптимальные планы для запросов, ссылающихся на статистику секционированных таблиц. Эту проблему можно обойти, вручную запуская `ANALYZE` для секционированных

ванных таблиц при их первом заполнении, а также всякий раз, когда распределение данных в их секциях существенно меняется.

Автоочистка не обрабатывает временные таблицы. Поэтому очистку и сбор статистики в них нужно производить с помощью SQL-команд в обычном сеансе.

Используемые по умолчанию пороговые значения и коэффициенты берутся из `postgresql.conf`, однако их (и многие другие параметры, управляющие автоочисткой) можно переопределить для каждой таблицы; за подробностями обратитесь к разделу Параметры хранения. Если какие-либо значения определены через параметры хранения таблицы, при обработке этой таблицы действуют они, а в противном случае — глобальные параметры. За более подробной информацией о глобальных параметрах обратитесь к Разделу 18.10.

Когда выполняются несколько рабочих процессов, параметры задержки автоочистки по стоимости (см. Подраздел 18.4.4) «распределяются» между всеми этими процессами, так что общее воздействие на систему остаётся неизменным, независимо от их числа. Однако этот алгоритм распределения нагрузки не учитывает процессы, обрабатывающие таблицы с индивидуальными значениями параметров хранения `autovacuum_vacuum_cost_delay` и `autovacuum_vacuum_cost_limit`.

Рабочие процессы автоочистки обычно не мешают выполнению других команд. Если какой-либо процесс попытается получить блокировку, конфликтующую с блокировкой `SHARE UPDATE EXCLUSIVE`, которая удерживается в ходе автоочистки, автоочистка прервётся и процесс получит нужную ему блокировку. Конфликтующие режимы блокировок отмечены в Таблице 13.2. Однако если автоочистка выполняется для предотвращения зацикливания идентификаторов транзакций (т. е. описание запроса автоочистки в представлении `pg_stat_activity` заканчивается на `(to prevent wraparound)`), автоочистка не прерывается без ручного вмешательства.

Предупреждение

При частом выполнении таких команд, как ANALYZE, которые затребуют блокировки, конфликтующие с `SHARE UPDATE EXCLUSIVE`, может получиться так, что автоочистка не будет успевать завершаться в принципе.

23.2. Регулярная переиндексация

В некоторых ситуациях стоит периодически перестраивать индексы, выполняя команду REINDEX или последовательность отдельных шагов по восстановлению индексов.

Страницы индексов на основе B-деревьев, которые стали абсолютно пустыми, могут быть использованы повторно. Однако возможность неэффективного использования пространства всё же остаётся: если со страницы были удалены почти все, но не все ключи индекса, страница всё равно остаётся занятой. Следовательно, шаблон использования, при котором со временем удаляются многие, но не все ключи в каждом диапазоне, приведёт к неэффективному расходованию пространства. В таких случаях рекомендуется периодически проводить переиндексацию.

Возможность раздувания в индексах на основе не B-деревьев глубоко не исследовалась. Поэтому имеет смысл периодически отслеживать физический размер индекса, когда применяется индекс такого типа.

Кроме того, с B-деревьями доступ по недавно построенному индексу осуществляется немного быстрее, нежели доступ по индексу, который неоднократно изменялся, поскольку в недавно построенном индексе страницы, близкие логически, обычно расположены так же близко и физически. (Это соображение неприменимо к индексам, которые основаны не на B-деревьях.) Поэтому периодически проводить переиндексацию стоит хотя бы для того, чтобы увеличить скорость доступа.

Команда REINDEX проста и безопасна для использования в любых случаях. Эта команда по умолчанию затребует блокировку `ACCESS EXCLUSIVE`, поэтому её обычно лучше выполнять с указанием `CONCURRENTLY`, с которым затребуется только `SHARE UPDATE EXCLUSIVE`.

23.3. Обслуживание журнала

Журнал сервера базы данных желательно сохранять где-либо, а не просто сбрасывать его в `/dev/null`. Этот журнал бесценен при диагностике проблем.

> **Примечание**
>
> Журнал сервера может содержать конфиденциальную информацию и должен быть защищён, где бы он ни хранился и куда бы ни передавался. Например, операторы DDL могут содержать незашифрованные пароли или другие данные аутентификации. В операторах, выводимых на уровне `ERROR`, может отображаться исходный код SQL для приложений, а также могут содержаться фрагменты строк данных. Так как журнал собственно предназначен для записи данных, событий и связанной с ними информации, это не является уязвимостью или дефектом. Вам следует позаботиться о том, чтобы к журналам сервера получали доступ только лица с соответствующими полномочиями.

Журнал сервера может быть очень объёмным (особенно при высоких уровнях отладки), так что хранить его неограниченно долго вы вряд ли захотите. Поэтому необходимо организовать *ротацию* журнальных файлов так, чтобы новые файлы создавались, а старые удалялись через разумный промежуток времени.

Если просто направить stderr команды `postgres` в файл, вы получите в нём журнал сообщений, но очистить этот файл можно будет, только если остановить и перезапустить сервер. Это может быть допустимо при использовании Postgres Pro в среде разработки, но вряд ли такой вариант будет приемлемым в производственной среде.

Лучшим подходом будет перенаправление вывода сервера stderr в какую-либо программу ротации журнальных файлов. Существует и встроенное средство ротации журнальных файлов, которое можно использовать, установив для параметра `logging_collector` значение `true` в `postgresql.conf`. Параметры, управляющие этой программой, описаны в Подразделе 18.8.1. Этот подход также можно использовать для получения содержимого журнала в формате CSV (значения, разделённые запятыми).

Вы также можете использовать внешнюю программу для ротации журнальных файлов, если уже применяете такое приложение для других серверных приложений. Например, утилиту rotatelogs, включённую в дистрибутив Apache, можно использовать и с Postgres Pro. Один из вариантов — направить вывод stderr сервера в желаемую программу. Если вы запускаете сервер, используя `pg_ctl`, то stderr уже будет перенаправлен в stdout, так что будет достаточно просто применить конвейер, например:

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

Вы можете скомбинировать эти подходы, настроив программу logrotate так, чтобы она собирала файлы журналов, которые записывает встроенный в Postgres Pro сборщик сообщений. В этом случае имена и расположение файлов журналов определяет сборщик сообщений, а logrotate периодически архивирует эти файлы. Когда logrotate производит ротацию журналов, важно, чтобы приложение выводило дальнейшие сообщения в новый файл. Обычно это делает скрипт `postrotate`, передающий сигнал `SIGHUP` приложению, которое в свою очередь заново открывает файл журнала. В Postgres Pro вместо этого вы можете выполнить `pg_ctl` с указанием `logrotate`. Когда выполняется эта команда, сервер либо переключается на новый файл журнала, либо заново открывает существующий, в зависимости от конфигурации журналирования (см. Подраздел 18.8.1).

> **Примечание**
>
> Когда используются статические имена файлов журналов, сервер может столкнуться с ошибкой при открытии файла, если будет достигнуто ограничение на максимальное число открытых файлов или переполнится таблица файлов. В этом случае сообщения будут продолжать

> выводиться в старый журнал пока не произойдёт успешное переключение. Если программа logrotate сжимает файл журнала и затем удаляет его, сообщения сервера, выводимые в этом интервале времени, могут быть потеряны. Чтобы избежать этого, можно сделать так, чтобы сборщик сообщений выбирал динамические имена файлов, а скрипт `prerotate` игнорировал открытые файлы журналов.

Ещё одно решение промышленного уровня заключается в передаче журнала в syslog, чтобы ротацией файлов занималась уже служба syslog. Для этого присвойте параметру конфигурации `log_destination` значение `syslog` (для вывода журнала только в syslog) в `postgresql.conf`. Затем вы сможете посылать сигнал `SIGHUP` службе syslog, когда захотите принудительно начать запись нового журнального файла. Если вы хотите автоматизировать ротацию журнальных файлов, программу logrotate можно настроить и для работы с журнальными файлами, которые формирует syslog.

Однако во многих системах, а особенно с большими сообщениями, syslog работает не очень надёжно; он может обрезать или терять сообщения как раз тогда, когда они вам нужны. Кроме того, в Linux, syslog> сбрасывает каждое сообщение на диск, от чего страдает производительность. (Для отключения этой синхронной записи можно добавить «`-`» перед именем файла в файле конфигурации syslog.)

Обратите внимание, что все описанные выше решения обеспечивают создание новых журнальных файлов через задаваемые промежутки времени, но не удаление старых, ставших бесполезными файлов журналов. Возможно, вы захотите создать задание для периодического удаления старых файлов. Кроме того, вы можете настроить программу ротации файлов так, чтобы старые файлы журналов циклически перезаписывались.

Инструмент pgbadger выполняет сложный анализ файлов журнала. Кроме того, *check_postgres* может посылать уведомления в Nagios, когда в журнале появляются важные сообщения, а также при обнаружении других нестандартных ситуаций.

Глава 24. Резервное копирование и восстановление

Как и всё, что содержит важные данные, базы данных Postgres Pro следует регулярно сохранять в резервной копии. Хотя эта процедура по существу проста, важно чётко понимать лежащие в её основе приёмы и положения.

Существует три фундаментально разных подхода к резервному копированию данных в Postgres Pro:

- Выгрузка в SQL
- Копирование на уровне файлов
- Непрерывное архивирование

Каждый из них имеет свои сильные и слабые стороны; все они обсуждаются в следующих разделах.

24.1. Выгрузка в SQL

Идея, стоящая за этим методом, заключается в генерации текстового файла с командами SQL, которые при выполнении на сервере пересоздадут базу данных в том же самом состоянии, в котором она была на момент выгрузки. Postgres Pro предоставляет для этой цели вспомогательную программу pg_dump. Простейшее применение этой программы выглядит так:

```
pg_dump имя_базы > файл_дампа
```

Как видите, pg_dump записывает результаты своей работы в устройство стандартного вывода. Далее будет рассмотрено, чем это может быть полезно. В то время как вышеупомянутая команда создаёт текстовый файл, pg_dump может создать файлы и в других форматах, которые допускают параллельную обработку и более гибкое управление восстановлением объектов.

Программа pg_dump является для Postgres Pro обычным клиентским приложением (хотя и весьма умным). Это означает, что вы можете выполнять процедуру резервного копирования с любого удалённого компьютера, если имеете доступ к нужной базе данных. Но помните, что pg_dump не использует для своей работы какие-то специальные привилегии. В частности, ей обычно требуется доступ на чтение всех таблиц, которые вы хотите выгрузить, так что для копирования всей базы данных практически всегда её нужно запускать с правами суперпользователя СУБД. (Если у вас нет достаточных прав для резервного копирования всей базы данных, вы тем не менее можете сделать резервную копию той части базы, доступ к которой у вас есть, используя такие параметры, как `-n` *`схема`* или `-t` *`таблица`*.)

Указать, к какому серверу должна подключаться программа pg_dump, можно с помощью аргументов командной строки `-h` *`сервер`* и `-p` *`порт`*. По умолчанию в качестве сервера выбирается localhost или значение, указанное в переменной окружения `PGHOST`. Подобным образом, по умолчанию используется порт, заданный в переменной окружения `PGPORT`, а если она не задана, то порт, указанный по умолчанию при компиляции. (Для удобства при компиляции сервера обычно устанавливается то же значение по умолчанию.)

Как и любое другое клиентское приложение Postgres Pro, pg_dump по умолчанию будет подключаться к базе данных с именем пользователя, совпадающим с именем текущего пользователя операционной системы. Чтобы переопределить имя, либо добавьте параметр `-U`, либо установите переменную окружения `PGUSER`. Помните, что pg_dump подключается к серверу через обычные механизмы проверки подлинности клиента (которые описываются в Главе 19).

Важное преимущество pg_dump в сравнении с другими методами резервного копирования, описанными далее, состоит в том, что вывод pg_dump обычно можно загрузить в более новые версии Postgres Pro, в то время как резервная копия на уровне файловой системы и непрерывное архивирование жёстко зависят от версии сервера. Также, только метод с применением pg_dump будет работать при переносе базы данных на другую машинную архитектуру, например, при переносе с 32-битной на 64-битную версию сервера.

Дампы, создаваемые pg_dump, являются внутренне согласованными, то есть, дамп представляет собой снимок базы данных на момент начала запуска pg_dump. pg_dump не блокирует другие операции с базой данных во время своей работы. (Исключение составляют операции, которым нужна исключительная блокировка, как например, большинство форм команды `ALTER TABLE`.)

24.1.1. Восстановление дампа

Текстовые файлы, созданные pg_dump, предназначаются для последующего чтения программой psql с использованием параметров по умолчанию. Общий вид команды для восстановления дампа в текстовом формате:

```
psql -X имя_базы < файл_дампа
```

где *`файл_дампа`* — это файл, содержащий вывод команды pg_dump. База данных, заданная параметром *`имя_базы`*, не будет создана данной командой, так что вы должны создать её сами из базы `template0` перед запуском psql (например, с помощью команды `createdb -T template0 имя_базы`). Убедиться, что psql работает с параметрами по умолчанию, можно с помощью параметра `-X` (`--no-psqlrc`). Программа psql принимает параметры, указывающие сервер, к которому осуществляется подключение, и имя пользователя, подобно pg_dump. За дополнительными сведениями обратитесь к справке по psql.

Дампы, выгруженные не в текстовом формате, восстанавливаются утилитой pg_restore.

Перед восстановлением SQL-дампа все пользователи, которые владели объектами или имели права на объекты в выгруженной базе данных, должны уже существовать. Если их нет, при восстановлении будут ошибки пересоздания объектов с изначальными владельцами и/или правами. (Иногда это желаемый результат, но обычно нет).

По умолчанию, если происходит ошибка SQL, программа psql продолжает выполнение. Если же запустить psql с установленной переменной `ON_ERROR_STOP`, это поведение поменяется и psql завершится с кодом 3 в случае возникновения ошибки SQL:

```
psql -X --set ON_ERROR_STOP=on имя_базы < файл_дампа
```

В любом случае вы получите только частично восстановленную базу данных. В качестве альтернативы можно указать, что весь дамп должен быть восстановлен в одной транзакции, так что восстановление либо полностью выполнится, либо полностью отменится. Включить данный режим можно, передав psql аргумент `-1` или `--single-transaction`. Выбирая этот режим, учтите, что даже незначительная ошибка может привести к откату восстановления, которое могло продолжаться несколько часов. Однако это всё же может быть предпочтительней, чем вручную вычищать сложную базу данных после частично восстановленного дампа.

Благодаря способности pg_dump и psql писать и читать каналы ввода-вывода, можно скопировать базу данных непосредственно с одного сервера на другой, например:

```
pg_dump -h host1 имя_базы | psql -X -h host2 имя_базы
```

Важно

Дампы, которые выдаёт pg_dump, содержат определения относительно `template0`. Это означает, что любые языки, процедуры и т. п., добавленные в базу через `template1`, pg_dump также выгрузит в дамп. Как следствие, если при восстановлении вы используете модифицированный `template1`, вы должны создать пустую базу данных из `template0`, как показано в примере выше.

После восстановления резервной копии имеет смысл запустить `ANALYZE` для каждой базы данных, чтобы оптимизатор запросов получил полезную статистику; за подробностями обратитесь к Подразделу 23.1.3 и Подразделу 23.1.6. Другие советы по эффективной загрузке больших объёмов данных в Postgres Pro вы можете найти в Разделе 14.4.

24.1.2. Использование pg_dumpall

Программа pg_dump выгружает только одну базу данных в один момент времени и не включает в дамп информацию о ролях и табличных пространствах (так как это информация уровня кластера, а не самой базы данных). Для удобства создания дампа всего содержимого кластера баз данных предоставляется программа pg_dumpall, которая делает резервную копию всех баз данных кластера, а также сохраняет данные уровня кластера, такие как роли и определения табличных пространств. Простое использование этой команды:

```
pg_dumpall > файл_дампа
```

Полученную копию можно восстановить с помощью psql:

```
psql -X -f файл_дампа postgres
```

(В принципе, здесь в качестве начальной базы данных можно указать имя любой существующей базы, но если вы загружаете дамп в пустой кластер, обычно нужно использовать `postgres`). Восстанавливать дамп, который выдала pg_dumpall, всегда необходимо с правами суперпользователя, так как они требуются для восстановления информации о ролях и табличных пространствах. Если вы используете табличные пространства, убедитесь, что пути к табличным пространствам в дампе соответствуют новой среде.

pg_dumpall выдаёт команды, которые заново создают роли, табличные пространства и пустые базы данных, а затем вызывает для каждой базы pg_dump. Таким образом, хотя каждая база данных будет внутренне согласованной, состояние разных баз не будет синхронным.

Только глобальные данные кластера можно выгрузить, передав pg_dumpall ключ `--globals-only`. Это необходимо, чтобы полностью скопировать кластер, когда pg_dump выполняется для отдельных баз данных.

24.1.3. Управление большими базами данных

Некоторые операционные системы накладывают ограничение на максимальный размер файла, что приводит к проблемам при создании больших файлов с помощью pg_dump. К счастью, pg_dump может писать в стандартный вывод, так что вы можете использовать стандартные инструменты Unix для того, чтобы избежать потенциальных проблем. Вот несколько возможных методов:

Используйте сжатые дампы. Вы можете использовать предпочитаемую программу сжатия, например gzip:

```
pg_dump имя_базы | gzip > имя_файла.gz
```

Затем загрузить сжатый дамп можно командой:

```
gunzip -c имя_файла.gz | psql имя_базы
```

или:

```
cat имя_файла.gz | gunzip | psql имя_базы
```

Используйте `split`. Команда `split` может разбивать выводимые данные на небольшие файлы, размер которых удовлетворяет ограничению нижележащей файловой системы. Например, чтобы получить части по 2 гигабайта:

```
pg_dump имя_базы | split -b 2G - имя_файла
```

Восстановить их можно так:

```
cat имя_файла* | psql имя_базы
```

Использовать GNU split можно вместе с gzip:

```
pg_dump имя_базы | split -b 2G --filter='gzip > $FILE.gz'
```

Восстановить данные после такого разбиения можно с помощью команды `zcat`.

Используйте специальный формат дампа pg_dump. Если при сборке Postgres Pro была подключена библиотека zlib, дамп в специальном формате будет записываться в файл в сжатом виде.

В таком формате размер файла дампа будет близок к размеру, полученному с применением `gzip`, но он лучше тем, что позволяет восстанавливать таблицы выборочно. Следующая команда выгружает базу данных в специальном формате:

```
pg_dump -Fc имя_базы > имя_файла
```

Дамп в специальном формате не является скриптом для psql и должен восстанавливаться с помощью команды pg_restore, например:

```
pg_restore -d имя_базы имя_файла
```

За подробностями обратитесь к справке по командам pg_dump и pg_restore.

Для очень больших баз данных может понадобиться сочетать `split` с одним из двух других методов.

Используйте возможность параллельной выгрузки в pg_dump. Чтобы ускорить выгрузку большой БД, вы можете использовать режим параллельной выгрузки в pg_dump. При этом одновременно будут выгружаться несколько таблиц. Управлять числом параллельных заданий позволяет параметр `-j`. Параллельная выгрузка поддерживается только для формата архива в каталоге.

```
pg_dump -j число -F d -f выходной_каталог имя_базы
```

Вы также можете восстановить копию в параллельном режиме с помощью `pg_restore -j`. Это поддерживается для любого архива в формате каталога или специальном формате, даже если архив создавался не командой `pg_dump -j`.

24.2. Резервное копирование на уровне файлов

Альтернативной стратегией резервного копирования является непосредственное копирование файлов, в которых Postgres Pro хранит содержимое базы данных; в Разделе 17.2 рассказывается, где находятся эти файлы. Вы можете использовать любой способ копирования файлов по желанию, например:

```
tar -cf backup.tar /usr/local/pgsql/data
```

Однако существуют два ограничения, которые делают этот метод непрактичным или как минимум менее предпочтительным по сравнению с pg_dump:

1. Чтобы полученная резервная копия была годной, сервер баз данных *должен* быть остановлен. Такие полумеры, как запрещение всех подключений к серверу, работать *не* будут (отчасти потому что `tar` и подобные средства не получают мгновенный снимок состояния файловой системы, но ещё и потому, что в сервере есть внутренние буферы). Узнать о том, как остановить сервер, можно в Разделе 17.5. Необходимо отметить, что сервер нужно будет остановить и перед восстановлением данных.

2. Если вы ознакомились с внутренней организацией базы данных в файловой системе, у вас может возникнуть соблазн скопировать или восстановить только отдельные таблицы или базы данных в соответствующих файлах или каталогах. Это *не* будет работать, потому что информацию, содержащуюся в этих файлах, нельзя использовать без файлов журналов транзакций, `pg_xact/*`, которые содержат состояние всех транзакций. Без этих данных файлы таблиц непригодны к использованию. Разумеется также невозможно восстановить только одну таблицу и соответствующие данные `pg_xact`, потому что в результате нерабочими станут все другие таблицы в кластере баз данных. Таким образом, копирование на уровне файловой системы будет работать, только если выполняется полное копирование и восстановление всего кластера баз данных.

Ещё один подход к резервному копированию файловой системы заключается в создании «целостного снимка» каталога с данными, если это поддерживает файловая система (и вы склонны считать, что эта функциональность реализована корректно). Типичная процедура включает создание «замороженного снимка» тома, содержащего базу данных, затем копирование всего каталога с данными (а не его избранных частей, см. выше) из этого снимка на устройство резервного копирования, и наконец освобождение замороженного снимка. При этом сервер базы данных может не прекращать свою работу. Однако резервная копия, созданная таким способом, содержит файлы базы данных в таком состоянии, как если бы сервер баз данных не был остановлен штатным образом; таким образом, когда вы запустите сервер баз данных с сохранёнными данными, он будет считать, что до этого процесс сервера был прерван аварийно, и будет накатывать журнал WAL. Это не проблема, просто имейте это в виду (и обязательно включите файлы WAL в резервную копию). Чтобы сократить время восстановления, можно выполнить команду `CHECKPOINT` перед созданием снимка.

Если ваша база данных размещена в нескольких файловых системах, получить в точности одновременно замороженные снимки всех томов может быть невозможно. Например, если файлы данных и журналы WAL находятся на разных дисках или табличные пространства расположены в разных файловых системах, резервное копирование со снимками может быть неприменимо, потому что снимки *должны* быть одновременными. В таких ситуациях очень внимательно изучите документацию по вашей файловой системе, прежде чем довериться технологии согласованных снимков.

Если одновременные снимки невозможны, остаётся вариант с остановкой сервера баз данных на время, достаточное для получения всех замороженных снимков. Другое возможное решение — получить базовую копию путём непрерывного архивирования (см. Подраздел 24.3.2), такие резервные копии не могут пострадать от изменений файловой системы в процессе резервного копирования. Для этого требуется включить непрерывное архивирование только на время резервного копирования; для восстановления применяется процедура восстановления из непрерывного архива (Подраздел 24.3.5).

Ещё один вариант — копировать содержимое файловой системы с помощью rsync. Для этого rsync запускается сначала во время работы сервера баз данных, а затем сервер останавливается на время, достаточное для запуска `rsync --checksum`. (Ключ `--checksum` необходим, потому что `rsync` различает время только с точностью до секунд.) Во второй раз rsync отработает быстрее, чем в первый, потому что скопировать надо будет относительно немного данных; и в итоге будет получен согласованный результат, так как сервер был остановлен. Данный метод позволяет получить копию на уровне файловой системы с минимальным временем простоя.

Обратите внимание, что размер копии на уровне файлов обычно больше, чем дампа SQL. (Программе pg_dump не нужно, например, записывать содержимое индексов, достаточно команд для их пересоздания). Однако копирование на уровне файлов может выполняться быстрее.

24.3. Непрерывное архивирование и восстановление на момент времени (Point-in-Time Recovery, PITR)

Всё время в процессе работы Postgres Pro ведёт *журнал предзаписи* (WAL), который расположен в подкаталоге `pg_wal/` каталога с данными кластера баз данных. В этот журнал записываются все изменения, вносимые в файлы данных. Прежде всего, журнал существует для безопасного восстановления после краха сервера: если происходит крах, целостность СУБД может быть восстановлена в результате «воспроизведения» записей, зафиксированных после последней контрольной точки. Однако наличие журнала делает возможным использование третьей стратегии копирования баз данных: можно сочетать резервное копирование на уровне файловой системы с копированием файлов WAL. Если потребуется восстановить данные, мы можем восстановить копию файлов, а затем воспроизвести журнал из скопированных файлов WAL, и таким образом привести систему в нужное состояние. Такой подход более сложен для администрирования, чем любой из описанных выше, но он имеет значительные преимущества:

- В качестве начальной точки для восстановления необязательно иметь полностью согласованную копию на уровне файлов. Внутренняя несогласованность копии будет исправлена при воспроизведении журнала (практически то же самое происходит при восстановлении после краха). Таким образом, согласованный снимок файловой системы не требуется, вполне можно использовать tar или похожие средства архивации.
- Поскольку при воспроизведении можно обрабатывать неограниченную последовательность файлов WAL, непрерывную резервную копию можно получить, просто продолжая архивировать файлы WAL. Это особенно ценно для больших баз данных, полные резервные копии которых делать как минимум неудобно.

- Воспроизводить все записи WAL до самого конца нет необходимости. Воспроизведение можно остановить в любой точке и получить целостный снимок базы данных на этот момент времени. Таким образом, данная технология поддерживает *восстановление на момент времени*: можно восстановить состояние базы данных на любое время с момента создания резервной копии.
- Если непрерывно передавать последовательность файлов WAL другому серверу, получившему данные из базовой копии того же кластера, получается система *тёплого резерва*: в любой момент мы можем запустить второй сервер и он будет иметь практически текущую копию баз данных.

> **Примечание**
>
> Программы pg_dump и pg_dumpall не создают копии на уровне файловой системы и не могут применяться как часть решения по непрерывной архивации. Создаваемые ими копии являются логическими и не содержат информации, необходимой для воспроизведения WAL.

Как и обычное резервное копирование файловой системы, этот метод позволяет восстанавливать только весь кластер баз данных целиком, но не его части. Кроме того, для архивов требуется большое хранилище: базовая резервная копия может быть объёмной, а нагруженные системы будут генерировать многие мегабайты трафика WAL, который необходимо архивировать. Тем не менее этот метод резервного копирования предпочитается во многих ситуациях, где необходима высокая надёжность.

Для успешного восстановления с применением непрерывного архивирования (также называемого «оперативным резервным копированием» многими разработчиками СУБД), вам необходима непрерывная последовательность заархивированных файлов WAL, начинающаяся не позже, чем с момента начала копирования. Так что для начала вы должны настроить и протестировать процедуру архивирования файлов WAL *до того*, как получите первую базовую копию. Соответственно, сначала мы обсудим механику архивирования файлов WAL.

24.3.1. Настройка архивирования WAL

В абстрактном смысле, запущенная СУБД Postgres Pro производит неограниченно длинную последовательность записей WAL. СУБД физически делит эту последовательность на *файлы сегментов* WAL, которые обычно имеют размер 16 МиБ (хотя размер сегмента может быть изменён при initdb). Файлы сегментов получают цифровые имена, которые отражают их позицию в абстрактной последовательности WAL. Когда архивирование WAL не применяется, система обычно создаёт только несколько файлов сегментов и затем «перерабатывает» их, меняя номер в имени ставшего ненужным файла на больший. Предполагается, что файлы сегментов, содержимое которых предшествует последней контрольной точке, уже не представляют интереса и могут быть переработаны.

При архивировании данных WAL необходимо считывать содержимое каждого файла-сегмента, как только он заполняется, и сохранять эти данные куда-то, прежде чем файл-сегмент будет переработан и использован повторно. В зависимости от применения и доступного аппаратного обеспечения, возможны разные способы «сохранить данные куда-то»: можно скопировать файлы-сегменты в смонтированный по NFS каталог на другую машину, записать их на ленту (убедившись, что у вас есть способ идентифицировать исходное имя каждого файла) или собрать их в пакет и записать на CD, либо какие-то совсем другие варианты. Чтобы у администратора баз данных была гибкость в этом плане, Postgres Pro пытается не делать каких-либо предположений о том, как будет выполняться архивация. Вместо этого, Postgres Pro позволяет администратору указать команду оболочки или библиотеку архивирования, которая будет запускаться для копирования файла завершённого сегмента в нужное место. Её действие может заключаться в выполнении простых команд оболочки, включая `cp`, или в вызове нетривиальной функции на языке C — это решать вам.

Чтобы включить архивирование WAL, установите в параметре конфигурации wal_level уровень `replica` или выше, в archive_mode — значение `on` и задайте желаемую команду оболочки в параметре archive_command или библиотеку архивирования в параметре archive_library. На практике эти параметры всегда задаются в файле `postgresql.conf`.

В `archive_command` символы `%p` заменяются полным путём к файлу, подлежащему архивации, а `%f` заменяются только именем файла. (Путь задаётся относительно текущего рабочего каталога, т. е. каталога данных кластера). Если в команду нужно включить сам символ `%`, запишите `%%`. Простейшая команда, которая может быть полезна:

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/
%f'  # Unix
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  # Windows
```

Она будет копировать архивируемые сегменты WAL в каталог /mnt/server/archivedir. (Команда дана как пример, а не как рекомендация, и может работать не на всех платформах.) После замены параметров `%p` и `%f` фактически запускаемая команда может выглядеть так:

```
test ! -f /mnt/server/archivedir/00000001000000A900000065 && cp
 pg_wal/00000001000000A900000065 /mnt/server/archivedir/00000001000000A900000065
```

Подобная команда будет генерироваться для каждого следующего архивируемого файла.

Команда архивирования будет запущена от имени того же пользователя, от имени которого работает сервер Postgres Pro. Поскольку архивируемые последовательности файлов WAL фактически содержат всё, что есть в вашей базе данных, вам нужно будет защитить архивируемые данные от посторонних глаз; например, сохраните архив в каталог, чтение которого запрещено для группы и остальных пользователей.

Важно, чтобы команда архивирования возвращала нулевой код завершения, если и только если она завершилась успешно. Получив нулевой результат, Postgres Pro будет полагать, что файл успешно заархивирован и удалит его или переработает. Однако ненулевой код состояния скажет Postgres Pro, что файл не заархивирован; попытки заархивировать его будут периодически повторяться, пока это не удастся.

Другой способ архивирования — использовать пользовательский модуль архивирования в качестве `archive_library`. Поскольку такие модули пишутся на языке `C`, создать собственный модуль гораздо сложнее, чем написать команду оболочки. Однако модули архивирования могут быть эффективнее, чем вызываемые команды оболочки, им доступны многие полезные ресурсы сервера. Дополнительная информация о модулях архивирования находится в Главе 49.

Когда команда архивирования завершается сигналом (отличным от SIGTERM, получаемого при штатном отключении сервера) или при возникновении ошибки оболочки (например, если команда не найдена) либо если функция архивирования выдаёт ошибку уровня `ERROR` или `FATAL`, процесс архиватора прерывается и перезапускается управляющим процессом postmaster. В таких случаях в pg_stat_archiver не сообщается об ошибке.

Команды и библиотеки архивирования обычно разрабатываются так, чтобы не допускать перезаписи любых существующих архивных файлов. Это важная мера безопасности, позволяющая сохранить целостность архива в случае ошибки администратора (например, если архивируемые данные двух разных серверов будут сохраняться в одном каталоге). Рекомендуется протестировать предлагаемую библиотеку архивирования, чтобы убедиться, что она не перезаписывает существующий файл.

В редких случаях Postgres Pro может попытаться повторно архивировать ранее заархивированный файл WAL. Например, если система выходит из строя до того, как сервер делает надёжную запись об успешном архивировании, он пытается снова заархивировать файл после перезапуска (при условии, что архивирование всё ещё включено). Когда команда или библиотека архивирования обнаруживает существовавший файл, первая должна возвращать нулевой статус, а вторая — `true`, если содержимое файла WAL полностью совпадает с содержимым существующего архива, который находится в хранилище. Если содержимое существующего файла отличается от содержимого архивируемого файла WAL, команда или библиотека архивирования *должны* возвращать ненулевой статус или `false` соответственно.

Рекомендуется протестировать команду архивирования, чтобы убедиться, что она действительно не перезаписывает существующие файлы. В показанной выше команде для Unix для этого добавлен отдельный шаг `test`. На некоторых платформах Unix у `cp` есть ключ `-i`, который позволяет сделать то же самое менее явно. Но не проверив, какой код состояния при этом возвращается, полагаться на этот ключ не следует. (В частности, GNU `cp` возвратит нулевой код состояния, если используется ключ `-i` и целевой файл существует, а это *не то*, что нужно.)

Разрабатывая схему архивирования, подумайте, что произойдёт, если команда или библиотека архивирования начнёт постоянно выдавать ошибку, потому что требуется вмешательство оператора или для архивирования не хватает места. Например, это может произойти, если вы записываете архивы на ленточное устройство без механизма автозамены; когда лента заполняется полностью, больше ничего архивироваться не будет, пока вы не замените кассету. Вы должны убедиться, что любые возникающие ошибки или обращения к человеку (оператору) обрабатываются так, чтобы проблема решалась достаточно быстро. Пока она не разрешится, каталог `pg_wal/` продолжит наполняться файлами-сегментами WAL. (Если файловая система, в которой находится каталог `pg_wal/` заполнится до конца, Postgres Pro завершит свою работу аварийно. Зафиксированные транзакции не потеряются, но база данных не будет работать, пока вы не освободите место.)

Не важно, с какой скоростью работает команда или библиотека архивирования, если только она не ниже средней скорости, с которой сервер генерирует записи WAL. Обычно работа продолжается, даже если процесс архивирования немного отстаёт. Если же архивирование отстаёт значительно, это приводит к увеличению объёма данных, которые могут быть потеряны в случае аварии. При этом каталог `pg_wal/` будет содержать большое количество ещё не заархивированных файлов-сегментов, которые в конце концов могут занять всё доступное дисковое пространство. Поэтому рекомендуется контролировать процесс архивации и следить за тем, чтобы он выполнялся как задумано.

При написании команды или библиотеки архивирования вы должны иметь в виду, что имена файлов для архивирования могут иметь длину до 64 символов и содержать любые комбинации из цифр, точек и букв ASCII. Сохранять исходный относительный путь (`%p`) необязательно, но необходимо сохранять имя файла (`%f`).

Обратите внимание, что хотя архивирование WAL позволяет сохранить любые изменения данных, произведённые в базе данных Postgres Pro, оно не затрагивает изменения, внесённые в конфигурационные файлы (такие как `postgresql.conf`, `pg_hba.conf` и `pg_ident.conf`), поскольку эти изменения выполняются вручную, а не через SQL. Поэтому имеет смысл разместить конфигурационные файлы там, где они будут заархивированы обычными процедурами копирования файлов. Как перемещать конфигурационные файлы, рассказывается в Разделе 18.2.

Команда или функция архивирования вызывается, только когда сегмент WAL заполнен до конца. Таким образом, если сервер постоянно генерирует небольшой трафик WAL (или есть продолжительные периоды, когда это происходит), между завершением транзакций и их безопасным сохранением в архиве может образоваться большая задержка. Чтобы ограничить время жизни неархивированных данных, можно установить archive_timeout, чтобы сервер переключался на новый файл сегмента WAL как минимум с заданной частотой. Заметьте, что неполные файлы, архивируемые досрочно из-за принудительного переключения по тайм-ауту, будут иметь тот же размер, что и заполненные файлы. Таким образом, устанавливать очень маленький `archive_timeout` неразумно — это приведёт к замусориванию архивного хранилища. Обычно подходящее значение `archive_timeout` — минута или около того.

Также вы можете принудительно переключить сегмент WAL вручную с помощью `pg_switch_wal`, если хотите, чтобы только что завершённая транзакция заархивировалась как можно скорее. Другие полезные функции, относящиеся к управлению WAL, перечисляются в Таблице 9.95.

Когда `wal_level` имеет значение `minimal`, некоторые команды SQL выполняются в обход журнала WAL, как описывается в Подразделе 14.4.7. Если архивирование или потоковая репликация были включены во время выполнения таких операторов, WAL не будет содержать информацию, необходимую для восстановления. (На восстановление после краха это не распространяется). Поэтому `wal_level` можно изменить только при запуске сервера. Однако для изменения параметра `archive_command` или `archive_library` достаточно перезагрузить файл конфигурации. Если вы используете для архивирования команду оболочки и хотите на время остановить архивирование, это можно сделать, например, задав в качестве значения `archive_command` пустую строку (''). В результате файлы WAL будут накапливаться в каталоге `pg_wal/`, пока не будет восстановлена действующая команда `archive_command`.

24.3.2. Создание базовой резервной копии

Проще всего получить базовую резервную копию, используя программу pg_basebackup. Эта программа сохраняет базовую копию в виде обычных файлов или в архиве tar. Если гибкости pg_basebackup не хватает, вы также можете получить базовую резервную копию, используя низкоуровневый API (см. Подраздел 24.3.4).

Продолжительность создания резервной копии обычно не имеет большого значения. Однако если вы эксплуатируете сервер с отключённым режимом `full_page_writes`, вы можете заметить падение производительности в процессе резервного копирования, так как режим `full_page_writes` включается принудительно на время резервного копирования.

Чтобы резервной копией можно было пользоваться, нужно сохранить все файлы сегментов WAL, сгенерированные во время и после копирования файлов. Для облегчения этой задачи, процесс создания базовой резервной копии записывает *файл истории резервного копирования*, который немедленно сохраняется в области архивации WAL. Данный файл получает имя по имени файла первого сегмента WAL, который потребуется для восстановления скопированных файлов. Например, если начальный файл WAL назывался `0000000100001234000055CD`, файл истории резервного копирования получит имя `0000000100001234000055CD.007C9330.backup`. (Вторая часть имени файла обозначает точную позицию внутри файла WAL и обычно может быть проигнорирована.) Как только вы заархивировали копии файлов данных и файлов сегментов WAL, полученных в процессе копирования (по сведениям в файле истории резервного копирования), все заархивированные сегменты WAL с именами, меньшими по номеру, становятся ненужными для восстановления файловой копии и могут быть удалены. Но всё же рассмотрите возможность хранения нескольких наборов резервных копий, чтобы быть абсолютно уверенными, что вы сможете восстановить ваши данные.

Файл истории резервного копирования — это просто небольшой текстовый файл. В него записывается метка, которая была передана pg_basebackup, а также время и текущие сегменты WAL в момент начала и завершения резервной копии. Если вы связали с данной меткой соответствующий файл дампа, то заархивированного файла истории достаточно, чтобы найти файл дампа, нужный для восстановления.

Поскольку необходимо хранить все заархивированные файлы WAL с момента последней базовой резервной копии, интервал базового резервного копирования обычно выбирается в зависимости от того, сколько места может быть выделено для архива файлов WAL. Также стоит отталкиваться от того, сколько вы готовы ожидать восстановления, если оно понадобится — системе придётся воспроизвести все эти сегменты WAL, а этот процесс может быть долгим, если с момента последней базовой копии прошло много времени.

24.3.3. Создание инкрементальной резервной копии

Для создания инкрементальной резервной копии можно использовать pg_basebackup с параметром `--incremental`. В качестве аргумента для `--incremental` необходимо указать манифест более ранней резервной копии с того же сервера. Файлы, не относящиеся к отношениям, будут включены в резервную копию полностью, но некоторые файлы отношений могут заменяться инкрементальными файлами меньшего размера, содержащими только блоки, которые были изменены с момента предыдущего резервного копирования, и достаточный объём метаданных для восстановления текущей версии файла.

Чтобы определить, какие блоки должны попасть в резервную копию, сервер использует сводки WAL, которые хранятся в каталоге данных внутри каталога `pg_wal/summaries`. Если необходимые файлы сводки отсутствуют, попытка создать инкрементальную резервную копию завершится ошибкой. Сводки, находящиеся в этом каталоге, должны охватывать все номера LSN от начального LSN предыдущей резервной копии до начального LSN текущей. Поскольку сервер ищет сводки

WAL сразу после определения начального LSN текущей копии, необходимые файлы сводки могут временно отсутствовать на диске, но сервер будет ждать их появления, что полезно, если процесс создания сводок WAL отстаёт. Однако, если необходимые файлы уже удалены или процесс создания сводок WAL не реагирует достаточно быстро, инкрементальное резервное копирование завершится ошибкой.

При восстановлении из инкрементальной резервной копии необходима не только она сама, но и все более ранние копии для добавления блоков, пропущенных в инкрементальной резервной копии. За дополнительной информацией об этом требовании обратитесь к pg_combinebackup. Обратите внимание, что существуют ограничения на использование `pg_combinebackup`, если состояние контрольной суммы кластера изменилось; см. ограничения pg_combinebackup.

Кроме того, обратите внимание, что все требования к использованию полной резервной копии также применимы и к инкрементальной резервной копии. Например, по-прежнему нужны все файлы сегментов WAL, созданные во время и после резервного копирования файловой системы, а также все соответствующие файлы истории WAL. И всё равно нужно создать `recovery.signal` (или `standby.signal`) и выполнить восстановление, как описано в Подразделе 24.3.5. Дополнительное требование именно инкрементального резервного копирования — наличие более ранних резервных копий во время восстановления и использовать `pg_combinebackup`. Имейте в виду, что Postgres Pro не имеет встроенного механизма определения резервных копий, которые всё ещё необходимы для восстановления последующих инкрементальных резервных копий. Пользователь должен самостоятельно отслеживать взаимосвязи между полными и инкрементальными резервными копиями и не удалять более ранние копии, если они могут понадобиться для дальнейшего восстановления.

Инкрементальное резервное копирование лучше подходит для относительно больших баз данных, где значительная часть данных не меняется или изменяется медленно. Для небольшой базы данных можно игнорировать существование инкрементальных резервных копий и создавать полные резервные копии, которыми проще управлять. Для большой базы данных, которая сильно меняется, размер инкрементальных резервных копий будет сопоставим с размером полных.

Инкрементальное резервное копирование возможно только в том случае, если воспроизведение начнётся с более поздней контрольной точки, чем для предыдущей копии, от которой она зависит. При выполнении инкрементального копирования на основном сервере указанное условие всегда выполняется, поскольку каждое резервное копирование создаёт новую контрольную точку. На резервном сервере воспроизведение начинается с последней точки перезапуска. Поэтому инкрементальное копирование на резервном сервере может завершиться ошибкой, если с момента предыдущего копирования было выполнено мало операций и новая точка перезапуска не была создана.

24.3.4. Создание базовой резервной копии через низкоуровневый API

Вместо создания полной или инкрементальной базовой резервной копии с помощью pg_basebackup можно создать базовую копию с использованием низкоуровневого API. Такая процедура содержит чуть больше шагов, чем метод pg_basebackup, но всё же относительно проста. Очень важно, чтобы эти шаги выполнялись по порядку, и следующий шаг выполнялся, только в случае успешного завершения предыдущего.

Несколько процессов резервного копирования могут быть запущены параллельно (используя данный API или pg_basebackup).

1. Убедитесь, что архивирование WAL включено и работает.
2. Подключитесь к серверу (к любой базе данных) как пользователь с правами на выполнение `pg_backup_start` (суперпользователь или пользователь, которому дано право `EXECUTE` для этой функции) и выполните команду:

   ```
   SELECT pg_backup_start(label => 'label', fast => false);
   ```

   где `label` — любая метка, по которой можно однозначно идентифицировать данную операцию резервного копирования. Соединение, через которое вызывается `pg_backup_start`, должно поддерживаться до окончания резервного копирования, иначе этот процесс будет автоматически прерван.

Резервное копирование всегда запускается в начале контрольной точки. По умолчанию `pg_backup_start` будет ждать следующей запланированной контрольной точки, что может занять длительное время (см. конфигурационные параметры checkpoint_timeout и checkpoint_completion_target). Обычно это вполне приемлемо, так как при этом минимизируется влияние на выполнение других запросов. Если же вы хотите начать резервное копирование максимально быстро, передайте во втором параметре `pg_backup_start` значение `true`. В этом случае контрольная точка будет выполнена немедленно без ограничения объёма ввода-вывода.

3. Скопируйте файлы, используя любое удобное средство резервного копирования, например, tar или cpio (не pg_dump или pg_dumpall). В процессе копирования останавливать работу базы данных не требуется, это ничего не даёт. В Подразделе 24.3.4.1 описано, что следует учитывать в процессе копирования.

4. Через то же подключение, что и раньше, выполните команду:

```
SELECT * FROM pg_backup_stop(wait_for_archive => true);
```

При этом сервер выйдет из режима резервного копирования. Ведущий сервер вместе с этим автоматически переключится на следующий сегмент WAL. На ведомом автоматическое переключение сегментов WAL невозможно, поэтому вы можете выполнить `pg_switch_wal` на ведущем, чтобы произвести переключение вручную. Такое переключение позволяет получить готовый к архивированию последний сегмент WAL, записанный в процессе резервного копирования.

Функция `pg_backup_stop` возвратит одну строку с тремя значениями. Второе из них нужно записать в файл `backup_label` в корневой каталог резервной копии. Третье значение, если оно не пустое, должно быть записано в файл `tablespace_map`. Эти файлы крайне важны для восстановления копии и должны записываться байт за байтом без изменений, для чего может потребоваться открыть файл в двоичном редакторе.

5. Как только файлы сегментов WAL, активные во время создания резервной копии, будут заархивированы, процедура резервного копирования будет завершена. Функция `pg_backup_stop` в первом значении результата указывает, какой последний сегмент требуется для формирования полного набора файлов резервной копии. На ведущем сервере, если включён режим архивации (параметр `archive_mode`) и аргумент `wait_for_archive` равен `true`, функция `pg_backup_stop` не завершится, пока не будет заархивирован последний сегмент. На ведомом параметр `archive_mode` должен иметь значение `always`, чтобы функция `pg_backup_stop` ожидала архивации. Эти файлы будут заархивированы автоматически, так как вы ранее задали значение `archive_library` или `archive_command`. Чаще всего это происходит быстро, но мы советуем наблюдать за системой архивации и проверять, не возникают ли задержки. Если архивирование остановится из-за ошибок команды или библиотеки архивации, попытки архивации будут продолжаться до успешного завершения, и только тогда резервное копирование окончится. Если вы хотите ограничить время выполнения `pg_backup_stop`, установите соответствующее значение в `statement_timeout`, но заметьте, что в случае прерывания `pg_backup_stop` по времени резервная копия может оказаться негодной.

Если в процедуре резервного копирования предусмотрено отслеживание и архивация всех файлов сегментов WAL, необходимых для резервной копии, то в аргументе `wait_for_archive` (по умолчанию равном true) можно передать false, чтобы функция `pg_backup_stop` завершилась сразу, как только в WAL будет помещена запись о завершении копирования. По умолчанию `pg_backup_stop` будет ждать окончания архивации всех файлов WAL, что может занять некоторое время. Использовать этот параметр следует с осторожностью: если архивация WAL не контролируется, в резервной копии могут оказаться не все необходимые файлы WAL и её нельзя будет восстановить.

24.3.4.1. Копирование каталога данных

Некоторые средства резервного копирования файлов выдают предупреждения или ошибки, если файлы, которые они пытаются скопировать, изменяются в процессе копирования. При получении базовой резервной копии активной базы данных это вполне нормально и не является ошибкой. Однако вам нужно знать, как отличить ошибки такого рода от реальных ошибок. Например, некоторые версии rsync возвращают отдельный код завершения для ситуации «исчезнувшие исходные файлы», и вы можете написать управляющий скрипт, который примет этот код как не ошибочный. Также некоторые версии GNU tar возвращают код завершения, неотличимый от кода критической ошибки, если файл был усечён, когда tar копировал его. К счастью, GNU tar версий 1.16 и более поздних завершается с кодом 1, если файл был изменён во время копирования, и 2 в случае других ошибок. С GNU tar версии 1.23 и более поздними, вы можете использовать следующие ключи `--warning=no-file-changed --warning=no-file-removed`, чтобы скрыть соответствующие предупреждения.

Убедитесь, что ваша резервная копия включает все файлы из каталога кластера баз данных (например, `/usr/local/pgsql/data`). Если вы используете табличные пространства, которые находятся не внутри этого каталога, не забудьте включить и их в резервную копию (также важно, чтобы при создании резервной копии символьные ссылки сохранялись как ссылки, иначе табличные пространства будут повреждены при восстановлении).

Однако следует исключить из резервной копии файлы в подкаталоге данных кластера `pg_wal/`. Эту небольшую корректировку стоит внести для снижения риска ошибок при восстановлении. Это легко организовать, если `pg_wal/` — символическая ссылка на каталог за пределами каталога данных (так часто делают из соображений производительности). Также имеет смысл исключить файлы `postmaster.pid` и `postmaster.opts`, содержащие информацию о работающем процессе postmaster (а не о том процессе postmaster, который будет восстанавливать эту копию). (Эти файлы могут ввести pg_ctl в заблуждение.)

Часто также стоит исключать из резервной копии каталог `pg_replslot/` кластера, чтобы слоты репликации, существующие на главном сервере, не попадали в копию. В противном случае при последующем восстановлении копии на резервном сервере может получиться так, что он будет неограниченно долго сохранять файлы WAL, а главный не будет очищаться, если он следит за горячим резервом, так как клиенты этих слотов репликации будут продолжать подключаться и изменять состояние слотов на главном, а не резервном сервере. Даже если резервная копия предназначена только для создания нового главного сервера, копирование слотов репликации вряд ли принесёт пользу, так как к моменту включения в работу этого нового сервера содержимое этих слотов станет абсолютно неактуальным.

Содержимое каталогов `pg_dynshmem/`, `pg_notify/`, `pg_serial/`, `pg_snapshots/`, `pg_stat_tmp/` и `pg_subtrans/` (но не сами эти каталоги) можно исключить из резервной копии, так как оно будет инициализировано при запуске главного процесса.

Из резервной копии можно исключить и файлы и подкаталоги с именами, начинающимся с `pgsql_tmp`. Эти файлы удаляются при запуске главного процесса, а каталоги создаются по мере необходимости.

Из резервной копии могут быть исключены файлы `pg_internal.init`. Такие файлы содержат кешируемые данные отношения и всегда перестраиваются при восстановлении.

В файл метки резервной копии записывается строка метки, заданная при вызове `pg_backup_start`, время запуска функции `pg_backup_start` и имя начального файла WAL. Таким образом, в случае сомнений можно заглянуть внутрь архива резервной копии и точно определить, в каком сеансе резервного копирования он был создан. Файл карты табличных пространств содержит имена символических ссылок, как они существуют в каталоге `pg_tblspc/`, и полный путь каждой символической ссылки. Эти файлы не только к вашему сведению; их существование и содержание важны для правильного проведения процесса восстановления системы.

Вы также можете создать резервную копию, когда сервер остановлен. В этом случае, вы, очевидно, не сможете вызвать `pg_backup_start` или `pg_backup_stop`, и следовательно, вам надо будет самостоятельно как-то идентифицировать резервные копии и понимать, какие файлы WAL должны быть заархивированы. Поэтому обычно всё-таки лучше следовать вышеописанной процедуре непрерывного архивирования.

24.3.5. Восстановление непрерывной архивной копии

Допустим, худшее случилось, и вам необходимо восстановить базу данных из резервной копии. Порядок действий таков:

1. Остановите сервер баз данных, если он запущен.
2. Если у вас есть место для этого, скопируйте весь текущий каталог кластера баз данных и все табличные пространства во временный каталог на случай, если они вам понадобятся. Учтите, что эта мера предосторожности требует, чтобы свободного места на диске было достаточно для размещения двух копий существующих данных. Если места недостаточно, необходимо сохранить как минимум содержимое подкаталога `pg_wal` каталога кластера, так как он может содержать файлы WAL, не попавшие в архив перед остановкой системы.
3. Удалите все существующие файлы и подкаталоги из каталога кластера и из корневых каталогов используемых табличных пространств.
4. Из полной резервной копии можно восстановить файлы БД непосредственно в целевые каталоги. Важно, чтобы у восстановленных файлов были правильные разрешения и правильный владелец (пользователь, запускающий сервер, а не `root`!). Если вы используете табличные пространства, проверьте также, что символьные ссылки в `pg_tblspc/` восстановились корректно.
5. При восстановлении из инкрементальной резервной копии нужно восстановить как саму копию, так и все более ранние копии, от которых она напрямую или косвенно зависит, на той же машине, где выполняется восстановление. Эти резервные копии следует размещать в отдельных каталогах, а не в целевых каталогах БД. После этого используйте pg_combinebackup, чтобы извлечь данные из полной и всех последующих инкрементальных копий и создать синтетическую полную резервную копию в целевых каталогах. Проверьте правильность разрешений и ссылок на табличные пространства, как описано выше.
6. Удалите все файлы из `pg_wal/`; они восстановились из резервной копии файлов и поэтому, скорее всего, будут старее текущих. Если вы вовсе не архивировали `pg_wal/`, создайте этот каталог с правильными правами доступа, но если это была символьная ссылка, восстановите её.
7. Если на шаге 2 вы сохранили незаархивированные файлы с сегментами WAL, скопируйте их в `pg_wal/`. (Лучше всего именно копировать, а не перемещать их, чтобы у вас остались неизменённые файлы на случай, если возникнет проблема и всё придётся начинать сначала.)
8. Установите параметры восстановления в `postgresql.conf` (см. Подраздел 18.5.5) и создайте файл `recovery.signal` в каталоге данных кластера. Вы можете также временно изменить `pg_hba.conf`, чтобы обычные пользователи не могли подключиться, пока вы не будете уверены, что восстановление завершилось успешно.
9. Запустите сервер. Сервер запустится в режиме восстановления и начнёт считывать необходимые ему архивные файлы WAL. Если восстановление будет прервано из-за внешней ошибки, сервер можно просто перезапустить и он продолжит восстановление. По завершении процесса восстановления сервер удалит файл `recovery.signal` (чтобы предотвратить повторный запуск режима восстановления), а затем перейдёт к обычной работе с базой данных.
10. Просмотрите содержимое базы данных, чтобы убедиться, что вы вернули её к желаемому состоянию. Если это не так, вернитесь к шагу 1. Если всё хорошо, разрешите пользователям подключаться к серверу, восстановив обычный файл `pg_hba.conf`.

Ключевой момент этой процедуры заключается в создании конфигурации восстановления, описывающей, как будет выполняться восстановление и до какой точки. Единственное, что совершенно необходимо задать — это команду `restore_command`, которая говорит Postgres Pro, как получать из архива файл-сегменты WAL. Как и `archive_command`, это командная строка для оболочки. Она может содержать символы `%f`, которые заменятся именем требующегося файла WAL, и `%p`, которые заменятся целевым путём для копирования этого файла. (Путь задаётся относительно текущего рабочего каталога, т. е. каталога кластера данных.) Если вам нужно включить в команду сам символ `%`, напишите `%%`. Простейшая команда, которая может быть полезна, такая:

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

Эта команда копирует заархивированные ранее сегменты WAL из каталога `/mnt/server/archivedir`. Разумеется, вы можете использовать что-то более сложное, возможно, даже скрипт оболочки, который укажет оператору установить соответствующую ленту.

Важно, чтобы данная команда возвращала ненулевой код возврата в случае ошибки. Эта команда *будет* вызываться и с запросом файлов, отсутствующих в архиве; в этом случае она должна вернуть ненулевое значение и это считается штатной ситуацией. В исключительной ситуации, когда команда была прервана сигналом (кроме SIGTERM, который применяется в процессе остановки сервера базы данных) или произошла ошибка оболочки (например, команда не найдена), восстановление будет прервано и сервер не запустится.

Не все запрашиваемые файлы будут сегментами WAL; следует также ожидать запросов файлов с суффиксом `.history`. Также учтите, что базовое имя пути `%p` будет отличаться от `%f`; не думайте, что они взаимозаменяемы.

Сегменты WAL, которые не найдутся в архиве, система будет искать в `pg_wal/`; благодаря этому можно использовать последние незаархивированные сегменты. Однако файлы в `pg_wal/` будут менее предпочтительными, если такие сегменты окажутся в архиве.

Обычно при восстановлении обрабатываются все доступные сегменты WAL и, таким образом, база данных восстанавливается до последнего момента времени (или максимально близкого к нему, в зависимости от наличия сегментов WAL). Таким образом, восстановление обычно завершается с сообщением «файл не найден»; точный текст сообщения об ошибке зависит от того, что делает `restore_command`. Вы также можете увидеть сообщение об ошибке в начале восстановления для файла с именем типа `00000001.history`. Это также нормально и обычно не говорит о какой-либо проблеме при восстановлении в простых ситуациях; подробнее об этом рассказывается в Подразделе 24.3.6.

Если вы хотите восстановить базу на какой-то момент времени (скажем, до момента, когда неопытный администратор базы данных удалил основную таблицу транзакций), просто укажите требуемую точку остановки. Вы можете задать эту точку, иначе называемую «целью восстановления», по дате/времени, именованной точке восстановления или определённому идентификатору транзакции. На момент написания этой документации полезными могут быть только указания даты/времени или имени точки восстановления, пока нет никаких средств, позволяющих точно определить, какой идентификатор транзакции нужно выбрать.

Примечание

Точка останова должна указывать на момент после окончания базового копирования, т. е. после времени завершения `pg_backup_stop`. Использовать базовую резервную копию для восстановления на момент времени, когда она ещё только создавалась, нельзя. (Чтобы восстановить данные на этот момент времени, придётся вернуться к предыдущей базовой резервной копии и накатывать изменения с этой позиции.)

Если при восстановлении обнаруживаются повреждённые данные WAL, восстановление прерывается в этом месте и сервер не запускается. В этом случае процесс восстановления можно перезапустить с начала, указав «цель восстановления» до точки повреждения, чтобы восстановление могло завершиться нормально. Если восстановление завершается ошибкой из-за внешней причины, например, из-за краха системы или недоступности архива WAL, его можно просто перезапустить, и оно продолжится с того места, где было прервано. Перезапуск восстановления реализован по тому же принципу, что и контрольные точки при обычной работе: сервер периодически сохраняет всё текущее состояние на диске и отражает это в файле `pg_control`, чтобы уже обработанные данные WAL не приходилось сканировать снова.

24.3.6. Линии времени

Возможность восстановить базу данных на некий предыдущий момент времени создаёт некоторые сложности, сродни научно-фантастическим историям о путешествиях во времени и параллельных мирах. Например, предположим, что в начальной истории базы данных вы удалили важную таблицу в 17:15 во вторник, но осознали эту ошибку только в среду в полдень. Вы можете спокойно взять резервную копию, восстановить данные на 17:14 во вторник и запустить сервер. В *этой* истории мира базы данных вы никогда не удаляли вышеупомянутую таблицу. Но предположим, что позже вы заметили, что это была не такая уж хорошая идея и захотели вернуться к утру среды в первоначальной истории базы данных. Вы не сможете сделать это, если в процессе работы базы данных она успеет перезаписать какие-либо файлы-сегменты WAL, приводящие к моменту времени, к которому вы хотите вернуться теперь. Таким образом, для получения желаемого результата необходимо как-то отличать последовательности записей WAL, добавленные после восстановления на какой-то момент времени от тех, что существовали в начальной истории базы данных.

Для решения этой проблемы в Postgres Pro есть такое понятие, как *линия времени*. Всякий раз, когда завершается восстановление из архива, создаётся новая линия времени, позволяющая идентифицировать последовательность записей WAL, добавленных после этого восстановления. Номер линии времени включается в имя файлов-сегментов WAL, так что файлы новой линии времени не перезаписывают данные WAL, сгенерированные предыдущими линиями времени. Например, `00000001` в имени файла WAL `0000000100001234000055CD` — это идентификатор линии времени в шестнадцатеричном формате. (Обратите внимание, что в других контекстах, например в сообщениях журнала сервера, идентификаторы линии времени обычно выводятся в десятичном формате.)

Фактически можно архивировать много различных линий времени. Хотя это может показаться бесполезной возможностью, на самом деле она часто бывает спасительной. Представьте, что вы не определились, какую точку времени выбрать для восстановления, и таким образом должны проводить восстановление методом проб и ошибок, пока не найдёте лучший момент для ответвления от старой истории. Без линий времени этот процесс быстро стал бы очень запутанным. А благодаря линиям времени можно вернуться к *любому* предыдущему состоянию, включая состояния в ветках линий времени, покинутых ранее.

Каждый раз, когда образуется новая линия времени, Postgres Pro создаёт файл «истории линии времени», показывающий, от какой линии времени ответвилась данная и когда. Эти файлы истории нужны, чтобы система могла выбрать правильные файлы-сегменты WAL при восстановлении из архива, содержащего несколько линий времени. Таким образом, они помещаются в область архивов WAL так же, как и файлы сегментов WAL. Файлы истории представляют собой небольшие текстовые файлы, так что они не занимают много места и их вполне можно сохранять неограниченно долго (в отличие от файлов сегментов, имеющих большой размер). Если хотите, вы можете добавлять в файл истории комментарии, свои собственные заметки о том, как и почему была создана эта конкретная линия времени. Такие комментарии будут особенно ценны, если в результате экспериментов у вас образуется хитросплетение разных линий времени.

По умолчанию восстановление осуществляется до самой последней линии времени, найденной в архиве. Если вы хотите восстановить состояние на линии времени, которая была текущей, когда создавалась копия, либо на какой-либо дочерней линии времени (то есть хотите вернуться к некоторому состоянию, которое тоже было получено в результате попытки восстановления), вам необходимо указать `current` или идентификатор целевой линии времени в recovery_target_timeline. Восстановить состояние на линии времени, ответвившейся раньше, чем была сделана базовая резервная копия, нельзя.

24.3.7. Советы и примеры

Ниже мы дадим несколько советов по настройке непрерывного архивирования.

24.3.7.1. Обособленные горячие резервные копии

Средства резервного копирования Postgres Pro можно применять для создания обособленных горячих копий. Эти копии нельзя использовать для восстановления на момент времени, но создаются и восстанавливаются они обычно гораздо быстрее, чем дампы pg_dump. (Они также намного больше, чем дампы pg_dump, так что в некоторых случаях выигрыш в скорости может быть потерян.)

Как и базовые резервные копии, обособленную горячую копию проще всего получить, используя программу pg_basebackup. Если вы вызовете эту программу с параметром `-X`, в эту копию автоматически будет включён весь журнал предзаписи, необходимый для её использования, так что никакие особые действия для восстановления не потребуются.

24.3.7.2. Сжатие журналов в архиве

Если размер архива имеет большое значение, можно воспользоваться gzip и сжимать архивные файлы:

```
archive_command = 'gzip < %p > /mnt/server/archivedir/%f.gz'
```

При этом для восстановления придётся использовать gunzip:

```
restore_command = 'gunzip < /mnt/server/archivedir/%f.gz > %p'
```

24.3.7.3. Скрипты `archive_command`

Многие в качестве команды `archive_command` используют скрипты, так что запись в `postgresql.conf` оказывается очень простой:

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

Применять отдельный файла скрипта целесообразно всегда, когда вы хотите использовать в процедуре архивирования несколько команд. Это позволяет управлять сложностью этой процедуры в рамках одного скрипта, который можно написать на любом популярном языке скриптов, например на bash или perl.

В частности, с помощью скриптов можно решить такие задачи:

- Копирование данных в безопасное внешнее хранилище
- Пакетная обработка файлов WAL, чтобы они передавались каждые три часа, а не по одному
- Взаимодействие с другими приложениями резервного копирования и восстановления
- Взаимодействие со средствами мониторинга, регистрация ошибок

Подсказка

Когда в `archive_command` используется скрипт, желательно включить logging_collector. Тогда все сообщения, которые скрипт выведет в stderr, будут записываться в журнал сервера баз данных, что позволит легко диагностировать ошибки в сложных конфигурациях.

24.3.8. Ограничения

На момент написания документации методика непрерывного архивирования имеет несколько ограничений. Они могут быть ликвидированы в будущих версиях:

- Если во время создания базовой резервной копии выполняется команда `CREATE DATABASE`, а затем база-шаблон, задействованная в `CREATE DATABASE`, изменяется, пока продолжается копирование, возможно, что при восстановлении эти изменения распространятся также и на созданную базу данных. Конечно, это нежелательно. Во избежание подобных рисков, лучше всего не изменять никакие базы-шаблоны во время получения базовой резервной копии.
- Команды `CREATE TABLESPACE` записываются в WAL с абсолютным путём и, таким образом, при воспроизведении WAL будут выполнены с тем же абсолютным путём. Это может быть нежелательно, если WAL воспроизводится на другой машине. Но опасность есть, даже если WAL воспроизводится на той же машине, но в другом каталоге данных: при воспроизведении будет так же перезаписано содержимое исходных табличных пространств. Во избежание потенциальных проблем такого рода лучше всего делать новую базовую резервную копию после создания или удаления табличных пространств.

Также следует заметить, что стандартный формат WAL не очень компактный, так как включает много снимков дисковых страниц. Эти снимки страниц предназначены для поддержки восстановления после сбоя, на случай, если понадобится исправить страницы, записанные на диск частично. В зависимости от аппаратного и программного обеспечения вашей системы, риск частичной записи может быть достаточно мал, так что его можно игнорировать, и в этом случае можно существенно уменьшить общий объём архивируемых файлов WAL, выключив снимки страниц с помощью параметра full_page_writes. (Прежде чем делать это, прочтите замечания и предупреждения в Главе 27.) Выключение снимков страниц не препятствует использованию WAL для восстановления PITR. Одним из направлений разработки в будущем является сжатие архивируемых данных WAL путём удаления ненужных копий страниц даже при включённом режиме `full_page_writes`. Тем временем администраторы могут сократить количество снимков страниц, включаемых в WAL, увеличив параметры интервала контрольных точек в разумных пределах.

Глава 25. Отказоустойчивость, балансировка нагрузки и репликация

Серверы базы данных могут работать совместно для обеспечения возможности быстрого переключения на другой сервер в случае отказа первого (отказоустойчивость) или для обеспечения возможности нескольким серверам БД обрабатывать один набор данных (балансировка нагрузки). В идеале, серверы БД могут работать вместе прозрачно для клиента. Веб-серверы, обрабатывающие статические страницы, можно совместить достаточно легко посредством простого распределения запросов на несколько машин. Фактически серверы баз данных только для чтения тоже могут быть совмещены достаточно легко. К сожалению, большинство серверов баз данных получают смешанные запросы на чтение/запись, а серверы с доступом на чтение/запись совместить гораздо сложнее. Это объясняется тем, что данные только для чтения достаточно единожды разместить на каждом сервере, а запись на любой из серверов должна распространиться на все остальные серверы, чтобы будущие запросы на чтение возвращали согласованные результаты.

Проблема синхронизации является главным препятствием для совместной работы серверов. Так как единственного решения, устраняющего проблему синхронизации во всех случаях, не существует, предлагается несколько решений. Разные решения подходят к проблеме по-разному и минимизируют её влияние в разных рабочих условиях.

Некоторые решения применяют синхронизацию, позволяя только одному серверу изменять данные. Сервер, который может изменять данные, называется сервером чтения/записи, *ведущим* или *главным* сервером. Сервер, который отслеживает изменения на ведущем, называется *ведомым* или *резервным* сервером. Резервный сервер, к которому нельзя подключаться до тех пор, пока он не будет повышен до главного, называется сервером *тёплого резерва*, а тот, который может принимать соединения и обрабатывать запросы только на чтение, называется сервером *горячего резерва*.

Некоторые решения являются синхронными, при которых транзакция, модифицирующая данные, не считается подтверждённой, пока все серверы не подтвердят транзакцию. Это гарантирует, что при отработке отказа не произойдёт потеря данных и что все балансирующие серверы возвращают целостные данные вне зависимости от того, к какому серверу был запрос. Асинхронное решение, напротив, допускает некоторую задержку между временем подтверждения транзакции и её передачей на другие серверы, допуская возможность, что некоторые транзакции могут быть потеряны в момент переключения на резервный сервер и что балансирующие серверы могут вернуть слегка устаревшие данные. Асинхронная передача используется, когда синхронная будет слишком медленной.

Решения могут так же разделяться по степени детализации. Некоторые решения работают только на уровне всего сервера БД целиком, в то время как другие позволяют работать на уровне таблиц или уровне БД.

В любом случае необходимо принимать во внимание быстродействие. Обычно выбирается компромисс между функциональностью и производительностью. Например, полностью синхронное решение в медленной сети может снизить производительность больше чем наполовину, в то время как асинхронное решение будет оказывать минимальное воздействие.

В продолжении этого раздела рассматриваются различные решения по организации отказоустойчивости, репликации и балансировки нагрузки.

25.1. Сравнение различных решений

Отказоустойчивость на разделяемых дисках

> Отказоустойчивость на разделяемых дисках позволяет избежать избыточности синхронизации путём задействования только одной копии базы данных. Она использует единственный дисковый массив, который разделяется между несколькими серверами. Если основной сервер БД откажет, резервный сервер может подключиться и запустить базу данных, что позволит восстановить БД после аварии. Это обеспечивает быстрое переключение без потери данных.

Функциональность разделяемого оборудования обычно реализована в сетевых устройствах хранения. Так же возможно применение сетевой файловой системы; особое внимание следует уделить тому, чтобы поведение системы полностью соответствовало POSIX (см. Подраздел 17.2.2.1). Существенное ограничение этого метода состоит в том, что в случае отказа или порчи разделяемого дискового массива оба сервера: главный и резервный — станут нерабочими. Другая особенность — резервный сервер никогда не получает доступ к разделяемым дискам во время работы главного.

Репликация на уровне файловой системы (блочного устройства)

Видоизменённая версия функциональности разделяемого оборудования представлена в виде репликации на уровне файловой системы, когда все изменения в файловой системе отражаются в файловой системе другого компьютера. Единственное ограничение: синхронизация должна выполняться методом, гарантирующим целостность копии файловой системы на резервном сервере — в частности, запись на резервном сервере должна происходить в том же порядке, что и на главном. DRBD является популярным решением на основе репликации файловой системы для Linux.

Трансляция журнала предзаписи

Серверы тёплого и горячего резерва могут так же поддерживаться актуальными путём чтения потока записей из журнала изменений (WAL). Если основной сервер отказывает, резервный содержит почти все данные с него и может быть быстро преобразован в новый главный сервер БД. Это можно сделать синхронно или асинхронно, но может быть выполнено только на уровне сервера БД целиком.

Резервный сервер может быть реализован с применением трансляции файлов журналов (см. Раздел 25.2), или потоковой репликации (см. Подраздел 25.2.5), или их комбинацией. За информацией о горячем резерве обратитесь к Разделу 25.4.

Логическая репликация

В схеме с логической репликацией сервер баз данных может передавать поток изменений данных на другой сервер. Механизм логической репликации в Postgres Pro формирует поток логических изменений данных, обрабатывая WAL. Логическая репликация позволяет переносить изменения данных на уровне таблиц. Для логической репликации не требуется, чтобы за определённым сервером закреплялась роль главного или реплицирующего; напротив, данные могут передаваться в разных направлениях. За дополнительными сведениями о логической репликации обратитесь к Главе 28. Используя интерфейс логического декодирования (Глава 47), подобную функциональность могут предоставлять и сторонние расширения.

Репликация главный-резервный на основе триггеров

Обычно в схеме репликации на основе триггеров запросы, изменяющие данные, передаются выбранному главному серверу. А главный сервер, обрабатывая такие запросы в разрезе таблиц, асинхронно (как правило) передаёт информацию об изменениях резервным. Резервные серверы могут выполнять запросы параллельно с главным и допускать некоторые изменения локальных данных или операции записи. Применение этого вида репликации часто позволяет снять с главного сервера нагрузку сложных аналитических запросов и запросов к хранилищу данных с главного сервера.

Slony-I является примером подобного типа репликации, действующей на уровне таблиц, и поддерживает множество резервных серверов. Так как обновления на резервных серверах происходят асинхронно (в пакетах), возможна потеря данных во время отказа.

Репликация SQL в среднем слое

В схеме с репликацией SQL в среднем слое, средний слой перехватывает каждый SQL-запрос и пересылает его на один или все серверы. Каждый сервер работает независимо. Модифицирующие запросы должны быть направлены всем серверам, чтобы каждый из них получал все изменения. Но читающие запросы могут посылаться только одному серверу, что позволяет перераспределить читающую нагрузку между всеми серверами.

Если запросы просто перенаправлять без изменений, функции подобные `random()`, `CURRENT_TIMESTAMP` и последовательности могут получить различные значения на разных серверах. Это происходит потому что каждый сервер работает независимо, а эти запросы неизбирательные и действительно не изменяют данные. Если такая ситуация недопустима, или средний слой, или приложение должны запросить подобные значения с одного сервера, затем использовать его в других пишущих запросах. Следует иметь в виду, что все транзакции фиксируются или прерываются на всех серверах, возможно с применением двухфазной фиксации (см. PREPARE TRANSACTION и COMMIT PREPARED). Репликацию такого типа реализуют, например, Pgpool-II и Continuent Tungsten.

Асинхронная репликация с несколькими ведущими

Если серверы не находятся постоянно в единой сети или связаны низкоскоростным каналом, как например, ноутбуки или удалённые серверы, обеспечение согласованности данных между ними представляет проблему. Когда используется асинхронная репликация с несколькими ведущими серверами, каждый из них работает независимо и периодически связывается с другими серверами для определения конфликтующих транзакций. Конфликты могут урегулироваться пользователем или по правилам их разрешения. Примером такого типа репликации является Bucardo.

Синхронная репликация с несколькими ведущими

При синхронной репликации с несколькими ведущими серверами каждый сервер может принимать запросы на запись, а изменённые данные передаются с получившего их сервера всем остальным, прежде чем транзакция будет подтверждена. Если запись производится интенсивно, это может провоцировать избыточные блокировки и задержки при фиксации, что приводит к снижению производительности. Запросы на чтение могут быть обработаны любым сервером. В некоторых конфигурациях для более эффективного взаимодействия серверов применяются разделяемые диски. Синхронная репликация с несколькими ведущими лучше всего работает, когда преобладают операции чтения, хотя её большой плюс в том, что любой сервер может принимать запросы на запись — нет необходимости искусственно разделять нагрузку между главным и резервными серверами, а так как изменения передаются от одного сервера другим, не возникает проблем с недетерминированными функциями вроде `random()`.

Postgres Pro Enterprise включает расширение `multimaster`, реализующее этот вид репликации. Используя `multimaster`, вы можете организовать синхронный кластер без разделения ресурсов, который обеспечивает расширяемость OLTP для читающих транзакций, а также высокую степень доступности с автоматическим восстановлением после сбоев.

Возможности представленных выше решений сравниваются в Таблице 25.1.

Таблица 25.1. Таблица свойств отказоустойчивости, балансировки нагрузки и репликации

Тип	Разделяемый диск	Репл. файловой системы	Трансляция журнала предзаписи	Логическая репл.	Триггерная репл.	Репл. SQL в среднем слое	Асинхр. репл. с н. в.	Синхр. репл. с н. в.
Известные примеры	NAS	DRBD	встроенная потоковая репл.	встроенная логическая репл., pglogical	Londiste, Slony	pgpool-II	Bucardo	расширение `multimaster` в составе Postgres Pro Enterprise
Метод взаим.	разделяемые диски	дисковые блоки	WAL	логическое декодирование	Строки таблицы	SQL	Строки таблицы	Строки таблицы и бло-

Тип	**Разделяемый диск**	**Репл. файловой системы**	**Трансляция журнала предзаписи**	**Логическая репл.**	**Триггерная репл.**	**Репл. SQL в среднем слое**	**Асинхр. репл. с н. в.**	**Синхр. репл. с н. в.**
								кировки строк
Не требуется специального оборудования		•	•	•	•	•	•	•
Допускается несколько ведущих серверов				•		•	•	•
Нет доп. нагрузки на ведущем	•		•	•		•		
Нет задержки при нескольких серверах	•		без синхр.	без синхр.	•		•	
Отказ ведущего сервера не может привести к потере данных	•	•	с синхр.	с синхр.		•		•
Сервер реплики принимает читающие запросы			с горячим резервом	•	•	•	•	•
Репликация на уровне таблиц				•	•		•	•
Не требуется разрешение конфликтов	•	•	•		•	•		•

Несколько решений, которые не подпадают под указанные выше категории:

Секционирование данных

При секционировании таблицы расщепляются на наборы данных. Каждый из наборов может быть изменён только на одном сервере. Например, данные могут быть секционированы по офисам, например, Лондон и Париж, с сервером в каждом офисе. В случае необходимости обращения одновременно к данным Лондона и Парижа, приложение может запросить оба сервера, или может быть применена репликация главный-резервный для предоставления копии только для чтения в другом офисе для каждого из серверов.

Выполнение параллельных запросов на нескольких серверах

Многие из описанных выше решений позволяют обрабатывать несколько запросов на нескольких серверах, но ни одно из них не поддерживает выполнение одного запроса на нескольких серверах, что позволило бы его ускорить. Данное же решение позволяет нескольким серверам обрабатывать один запрос одновременно. Для этого обычно данные разделяются между серверами, серверы выполняют свои части запросов, выдают результаты центральному серверу, а он, в свою очередь, объединяет полученные данные и выдаёт итоговый результат пользователю. Это решение может быть реализовано с применением набора средств PL/Proxy.

25.2. Трансляция журналов на резервные серверы

Постоянная архивация может использоваться для создания кластерной конфигурации *высокой степени доступности* (HA) с одним или несколькими *резервными серверами*, способными заменить ведущий сервер в случае выхода его из строя. Такую реализацию отказоустойчивости часто называют *тёплый резерв* или *трансляция журналов*.

Ведущий и резервный серверы работают совместно для обеспечения этой возможности, при этом они связаны опосредованно. Ведущий сервер работает в режиме постоянной архивации изменений, в то время как каждый резервный сервер работает в режиме постоянного приёма архивных изменений, получая файлы WAL от ведущего. Для обеспечения этой возможности не требуется вносить изменения в таблицы БД, поэтому администрировать данное решение репликации проще, чем ряд других. Так же такая конфигурация относительно слабо влияет на производительность ведущего сервера.

Непосредственную передачу записей WAL с одного сервера БД на другой обычно называют трансляцией журналов (или доставкой журналов). Postgres Pro реализует трансляцию журналов на уровне файлов, передавая записи WAL по одному файлу (сегменту WAL) единовременно. Файлы WAL (размером 16 МБ) можно легко и эффективно передать на любое расстояние, будь то соседний сервер, другая система в местной сети или сервер на другом краю света. Требуемая пропускная способность при таком подходе определяется скоростью записи транзакций на ведущем сервере. Трансляция журналов на уровне записей более фрагментарная операция, при которой изменения WAL передаются последовательно через сетевое соединение (см. Подраздел 25.2.5).

Следует отметить, что трансляция журналов асинхронна, то есть записи WAL доставляются после завершения транзакции. В результате образуется окно, когда возможна потеря данных при отказе сервера: будут утеряны ещё не переданные транзакции. Размер этого окна при трансляции файлов журналов может быть ограничен параметром `archive_timeout`, который может принимать значение меньше нескольких секунд. Тем не менее подобные заниженные значения могут потребовать существенного увеличения пропускной способности, необходимой для трансляции файлов. При потоковой репликации (см. Подраздел 25.2.5) окно возможности потери данных гораздо меньше.

Скорость восстановления достаточно высока, обычно резервный сервер становится полностью доступным через мгновение после активации. В результате такое решение называется тёплым резервом, что обеспечивает отличную отказоустойчивость. Восстановление сервера из архивной копии базы и применение изменений обычно происходит существенно дольше. Поэтому такие действия обычно требуются при восстановлении после аварии, не для отказоустойчивости. Так же резервный сервер может обрабатывать читающие запросы. В этом случае он называется сервером *горячего резерва*. См. Раздел 25.4 для подробной информации.

25.2.1. Планирование

Обычно разумно подбирать ведущий и резервный серверы так, чтобы они были максимально похожи, как минимум с точки зрения базы данных. Тогда в частности, пути, связанные с табличными пространствами, могут передаваться без изменений. Таким образом, как на ведущем, так и на резервных серверах должны быть одинаковые пути монтирования для табличных пространств при использовании этой возможности БД. Учитывайте, что если CREATE TABLESPACE выполнена на ведущем сервере, новая точка монтирования для этой команды уже должна существовать на резервных серверах до её выполнения. Аппаратная часть не должна быть в точности одинаковой, но опыт показывает, что сопровождать идентичные системы легче, чем две различные на протяжении жизненного цикла приложения и системы. В любом случае архитектура оборудования должна быть одинаковой — например, трансляция журналов с 32-битной на 64-битную систему не будет работать.

В общем случае трансляция журналов между серверами с различными основными версиями Postgres Pro невозможна. Политика главной группы разработки Postgres Pro состоит в том, чтобы не вносить изменения в дисковые форматы при обновлениях корректирующей версии, таким образом, ведущий и резервный серверы, имеющие разные корректирующие версии, могут работать успешно. Тем не менее формально такая возможность не поддерживается, поэтому рекомендуется поддерживать одинаковую версию ведущего и резервных серверов, насколько это возможно. При обновлении корректирующей версии безопаснее будет в первую очередь обновить резервные серверы — новая корректирующая версия с большей вероятностью прочитает файл WAL предыдущей корректирующей версии, чем наоборот.

25.2.2. Работа резервного сервера

Сервер переходит в режим ожидания, если в каталоге данных при запуске сервера есть файл `standby.signal`.

Сервер, работающий в режиме резервного, последовательно применяет файлы WAL, полученные от главного. Резервный сервер может читать файлы WAL из архива WAL (см. restore_command) или напрямую с главного сервера по соединению TCP (потоковая репликация). Резервный сервер также будет пытаться восстановить любой файл WAL, найденный в кластере резервного в каталоге `pg_wal`. Это обычно происходит после перезапуска сервера, когда он применяет заново файлы WAL, полученные от главного сервера перед перезапуском. Но можно и вручную скопировать файлы в каталог `pg_wal`, чтобы применить их в любой момент времени.

В момент запуска резервный сервер начинает восстанавливать все доступные файлы WAL, размещённые в архивном каталоге, указанном в команде `restore_command`. По достижении конца доступных файлов WAL или при сбое команды `restore_command` сервер пытается восстановить все файлы WAL, доступные в каталоге `pg_wal`. Если это не удаётся и потоковая репликация настроена, резервный сервер пытается присоединиться к ведущему и начать закачивать поток WAL с последней подтверждённой записи, найденной в архиве или `pg_wal`. Если это действие закончилось неудачей, или потоковая репликация не настроена, или соединение позднее разорвалось, резервный сервер возвращается к шагу 1 и пытается восстановить файлы из архива вновь. Цикл обращения за файлами WAL к архиву, `pg_wal`, и через потоковую репликацию продолжается до остановки сервера или его повышения до ведущего.

Режим резерва завершается и сервер переключается в обычный рабочий режим при получении команды `pg_ctl promote` или в результате вызова `pg_promote()`. Перед переключением сервер восстановит все файлы WAL, непосредственно доступные из архива или `pg_wal`, но пытаться подключиться к главному серверу он больше не будет.

25.2.3. Подготовка главного сервера для работы с резервными

Настройка постоянного архивирования на ведущем сервере в архивный каталог, доступный с резервного, описана в Разделе 24.3. Расположение архива должно быть доступно с резервного сервера даже при отключении главного, то есть его следует разместить на резервном или другом доверенном, но не на главном сервере.

При использовании потоковой репликации следует настроить режим аутентификации на ведущем сервере, чтобы разрешить соединения с резервных. Для этого создать роль и обеспечить подходящую запись в файле `pg_hba.conf` в разделе доступа к БД `replication`. Так же следует убедиться, что для параметра `max_wal_senders` задаётся достаточно большое значение в конфигурационном файле ведущего сервера. При использовании слотов для репликации также достаточно большое значение нужно задать для `max_replication_slots`.

Создание базовой резервной копии, необходимой для запуска резервного сервера, описано в Подразделе 24.3.2.

25.2.4. Настройка резервного сервера

Для запуска резервного сервера нужно восстановить резервную копию, снятую с ведущего (см. Подраздел 24.3.5). Затем нужно создать файл `standby.signal`в каталоге данных кластера резервного сервера. Задайте в restore_command обычную команду копирования файлов из архива WAL. Если планируется несколько резервных серверов в целях отказоустойчивости, значением параметра `recovery_target_timeline` должно быть `latest` (значение по умолчанию), чтобы резервный сервер переходил на новую линию времени, образуемую при обработке отказа и переключении на другой сервер.

Примечание

Команда restore_command должна немедленно завершиться при отсутствии файла; сервер повторит эту команду при необходимости.

При необходимости потоковой репликации задайте в primary_conninfo параметры строки соединения для libpq, включая имя (или IP-адрес) сервера и всё, что требуется для подключения к ведущему серверу. Если ведущий требует пароль для аутентификации, пароль также должен быть указан в primary_conninfo.

Если резервный сервер настраивается в целях отказоустойчивости, на нём следует настроить архивацию WAL, соединения и аутентификацию, как на ведущем сервере, потому что резервный сервер станет ведущим после отработки отказа.

При использовании архива WAL его размер может быть уменьшен с помощью команды, задаваемой в archive_cleanup_command, если она будет удалять файлы, ставшие ненужными для резервного сервера. Утилита pg_archivecleanup разработана специально для использования в `archive_cleanup_command` при типичной конфигурации с одним резервным сервером (см. pg_archivecleanup). Заметьте, что если архив используется в целях резервирования, необходимо сохранять все файлы, требующиеся для восстановления как минимум с последней базовой резервной копии, даже если они не нужны для резервного сервера.

Простой пример конфигурации:

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass options=''-c
 wal_sender_timeout=5000'''
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

Можно поддерживать любое количество резервных серверов, но при применении потоковой репликации необходимо убедиться, что значение `max_wal_senders` на ведущем достаточно большое, чтобы все они могли подключиться одновременно.

25.2.5. Потоковая репликация

При потоковой репликации резервный сервер может работать с меньшей задержкой, чем при трансляции файлов. Резервный сервер подключается к ведущему, который передаёт поток записей WAL резервному в момент их добавления, не дожидаясь окончания заполнения файла WAL.

Потоковая репликация асинхронна по умолчанию (см. Подраздел 25.2.8), то есть имеется небольшая задержка между подтверждением транзакции на ведущем сервере и появлением этих изменений на резервном. Тем не менее эта задержка гораздо меньше, чем при трансляции файлов журналов, обычно в пределах одной секунды, если резервный сервер достаточно мощный и справляется с нагрузкой. При потоковой репликации настраивать `archive_timeout` для уменьшения окна потенциальной потери данных не требуется.

При потоковой репликации без постоянной архивации на уровне файлов, сервер может избавиться от старых сегментов WAL до того, как резервный получит их. В этом случае резервный сервер потребует повторной инициализации из новой базовой резервной копии. Этого можно избежать, установив для `wal_keep_size` достаточно большое значение, при котором сегменты WAL будут защищены от ранней очистки, либо настроив слот репликации для резервного сервера. Если с резервного сервера доступен архив WAL, этого не требуется, так как резервный может всегда обратиться к архиву для восполнения пропущенных сегментов.

Чтобы настроить потоковую репликацию, сначала настройте резервный сервер в режиме передачи журналов в виде файлов, как описано в Разделе 25.2. Затем переключите его в режим потоковой репликации, установив в `primary_conninfo` строку подключения, указывающую на ведущий. Настройте listen_addresses и параметры аутентификации (см. `pg_hba.conf`) на ведущем сервере таким образом, чтобы резервный смог подключиться к псевдобазе `replication` ведущего (см. Подраздел 25.2.5.1).

В системах, поддерживающих параметр сокета keepalive, подходящие значения tcp_keepalives_idle, tcp_keepalives_interval и tcp_keepalives_count помогут ведущему вовремя заметить разрыв соединения.

Установите максимальное количество одновременных соединений с резервных серверов (см. описание max_wal_senders).

При запуске резервного сервера с правильно установленным `primary_conninfo` резервный подключится к ведущему после воспроизведения всех файлов WAL, доступных из архива. При успешном установлении соединения можно увидеть `walreceiver` на резервном сервере и соответствующий процесс `walsender` на ведущем.

25.2.5.1. Аутентификация

Право использования репликации очень важно ограничить так, чтобы только доверенные пользователи могли читать поток WAL, так как из него можно извлечь конфиденциальную информацию. Резервный сервер должен аутентифицироваться на главном от имени пользователя с правом `REPLICATION` или от имени суперпользователя. Настоятельно рекомендуется создавать выделенного пользователя с правами `REPLICATION` и `LOGIN` специально для репликации. Хотя право `REPLICATION` даёт очень широкие полномочия, оно не позволяет модифицировать данные в ведущей системе, тогда как с правом `SUPERUSER` это можно делать.

Список аутентификации клиентов для репликации содержится в `pg_hba.conf` в записях с установленным значением `replication` в поле *`database`*. Например, если резервный сервер запущен на компьютере с IP-адресом `192.168.1.100` и учётная запись для репликации `foo`, администратор может добавить следующую строку в файл `pg_hba.conf` ведущего:

```
# Разрешить пользователю "foo" с компьютера 192.168.1.100 подключаться к этому
# серверу в качестве партнёра репликации, если был передан правильный пароль.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

Имя компьютера и номер порта для ведущего, имя подключающегося пользователя и пароль указываются в primary_conninfo. Пароль также может быть задан в файле `~/.pgpass` на резервном сервере (в поле *`database`* нужно указать `replication`). Например, если ведущий принимает подключения по IP-адресу `192.168.1.50`, через порт `5432`, пользователя для репликации `foo` с паролем `foopass`, администратор может добавить следующую строку в файл `postgresql.conf` на резервном сервере:

```
# Резервный сервер подключается к ведущему, работающему на компьютере 192.168.1.50
```

```
# (порт 5432), от имени пользователя "foo" с паролем "foopass".
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

25.2.5.2. Мониторинг

Важным индикатором стабильности работы потоковой репликации является количество записей WAL, созданных на ведущем, но ещё не применённых на резервном сервере. Задержку можно подсчитать, сравнив текущую позиции записи WAL на ведущем с последней позицией WAL, полученной на резервном сервере. Эти позиции можно узнать, воспользовавшись функциями `pg_current_wal_lsn` на ведущем и `pg_last_wal_receive_lsn` на резервном, соответственно (за подробностями обратитесь к Таблице 9.95 и Таблице 9.96). Последняя полученная позиция WAL на резервном сервере также выводится в состоянии процесса-приёмника WAL, которое показывает команда `ps` (подробнее об этом в Разделе 26.1).

Список процессов-передатчиков WAL можно получить через представление `pg_stat_replication`. Большая разница между `pg_current_wal_lsn` и полем `sent_lsn` этого представления может указывать на то, что главный сервер работает с большой нагрузкой, тогда как разница между `sent_lsn` и `pg_last_wal_receive_lsn` на резервном может быть признаком задержек в сети или большой нагрузки резервного сервера.

На сервере горячего резерва состояние процесса-приёмника WAL можно получить через представление `pg_stat_wal_receiver`. Большая разница между `pg_last_wal_replay_lsn` и полем `flushed_lsn` свидетельствует о том, что WAL поступает быстрее, чем удаётся его воспроизвести.

25.2.6. Слоты репликации

Слоты репликации автоматически обеспечивают механизм сохранения сегментов WAL, пока они не будут получены всеми резервными и ведущий сервер не будет удалять строки, находящиеся в статусе recovery conflict, даже при отключении резервного.

Вместо использования слотов репликации для предотвращения удаления старых сегментов WAL можно применять wal_keep_size или сохранять сегменты в архиве с помощью команды archive_command или archive_library. Однако в отличие от слотов репликации при использовании этих методов может храниться больше сегментов WAL, чем необходимо.

Подобным образом параметр hot_standby_feedback сам по себе, без использования слота репликации, позволяет защитить востребованные строки от удаления при очистке, но не защищает в тот промежуток времени, когда резервный сервер не подключён.

Внимание

Следует помнить, что использование слотов репликации может привести к хранению такого количества сегментов WAL на сервере, что они заполнят всё пространство, выделенное для `pg_wal`. Чтобы ограничить размер файлов WAL, хранимых слотами репликации, используйте max_slot_wal_keep_size.

25.2.6.1. Запросы и действия слотов репликации

Каждый слот репликации обладает именем, состоящим из строчных букв, цифр и символов подчёркивания.

Имеющиеся слоты репликации и их статус можно просмотреть в представлении `pg_replication_slots`.

Слоты могут быть созданы и удалены как с помощью протокола потоковой репликации (см. Раздел 53.4), так и посредством функций SQL (см. Подраздел 9.28.6).

25.2.6.2. Пример конфигурации

Для создания слота репликации выполните:

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM pg_replication_slots;
  slot_name  | slot_type | active
-------------+-----------+--------
 node_a_slot | physical  | f
(1 row)
```

Чтобы резервный сервер использовал этот слот, укажите его в параметре `primary_slot_name` в конфигурации этого сервера. Например:

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

25.2.7. Каскадная репликация

Свойство каскадной репликации позволяет резервному серверу принимать соединения репликации и потоки WAL от других резервных, выступающих посредниками. Это может быть полезно для уменьшения числа непосредственных подключений к главному серверу, а также для уменьшения накладных расходов при передаче данных в интрасети.

Резервный сервер, выступающий как получатель и отправитель, называется каскадным резервным сервером. Резервные серверы, стоящие ближе к главному, называются серверами верхнего уровня, а более отдалённые — серверами нижнего уровня. Каскадная репликация не накладывает ограничений на количество или организацию последующих уровней, а каждый резервный соединяется только с одним сервером вышестоящего уровня, который в конце концов соединяется с единственным главным/ведущим сервером.

Резервный сервер каскадной репликации не только получает записи WAL от главного, но так же восстанавливает их из архива. Таким образом, даже если соединение с сервером более высокого уровня разорвётся, потоковая репликация для последующих уровней будет продолжаться до исчерпания доступных записей WAL.

Каскадная репликация в текущей реализации асинхронна. Параметры синхронной репликации (см. Подраздел 25.2.8) в настоящее время не оказывают влияние на каскадную репликацию.

Распространение обратной связи горячего резерва работает от нижестоящего уровня к вышестоящему уровню вне зависимости от способа организации связи.

Если вышестоящий резервный сервер будет преобразован в новый главный, нижестоящие серверы продолжат получать поток с нового главного при условии, что `recovery_target_timeline` имеет значение `latest` (по умолчанию).

Для использования каскадной репликации необходимо настроить резервный каскадный сервер на приём соединений репликации (то есть установить max_wal_senders и hot_standby, настроить host-based authentication). Так же может быть необходимо настроить на нижестоящем резервном значение `primary_conninfo` на каскадный резервный сервер.

25.2.8. Синхронная репликация

По умолчанию в Postgres Pro потоковая репликация асинхронна. Если ведущий сервер выходит из строя, некоторые транзакции, которые были подтверждены, но не переданы на резервный, могут быть потеряны. Объём потерянных данных пропорционален задержке репликации на момент отработки отказа.

Синхронная репликация предоставляет возможность гарантировать, что все изменения, внесённые в транзакции, были переданы одному или нескольким синхронным резервным серверам. Это повышает стандартный уровень надёжности, гарантируемый при фиксации транзакции. Этот уровень защиты соответствует второму уровню безопасности репликации из теории вычислительной техники, или групповой безопасности первого уровня (безопасности групповой и уровня 1), когда выбран режим `synchronous_commit remote_write`.

При синхронной репликации каждая фиксация пишущей транзакции ожидает подтверждения того, что запись фиксации помещена в журнал предзаписи на диске на обоих серверах: ведущем и резервном. При таком варианте потеря данных может произойти только в случае одновременного выхода из строя ведущего и резервного серверов. Это обеспечивает более высокий уровень надёжности, при условии продуманного подхода системного администратора к вопросам размещения и управления этими серверами. Ожидание подтверждения увеличивает уверенность в том, что данные не будут потеряны во время сбоя сервера, но при этом увеличивает время отклика для обработки транзакции. Минимальное время ожидания равно времени передачи данных от ведущего к резервному и обратно.

Транзакции только для чтения и откат транзакции не требуют ожидания для ответа с резервного сервера. Промежуточные подтверждения не ожидают ответа от резервного сервера, только подтверждение верхнего уровня. Долгие операции вида загрузки данных или построения индекса не ожидают финального подтверждения. Но все двухфазные подтверждения требуют ожидания, включая подготовку и непосредственно подтверждение.

Синхронным резервным сервером может быть резервный сервер при физической репликации или подписчик при логической репликации. Это также может быть другой потребитель потока логической или физической репликации, способный отправлять в ответ требуемые сообщения. Помимо встроенных систем логической и физической репликации, к таким потребителям относятся специальные программы, `pg_receivewal` и `pg_recvlogical`, а также некоторые сторонние системы репликации и внешние программы. Подробнее об организации синхронной репликации с их использованием можно узнать в соответствующей документации.

25.2.8.1. Базовая настройка

При настроенной потоковой репликации установка синхронной репликации требует только дополнительной настройки: необходимо выставить synchronous_standby_names в непустое значение. Так же необходимо установить `synchronous_commit` в значение `on`, но так как это значение по умолчанию, обычно действий не требуется. (См. Подраздел 18.5.1 и Подраздел 18.6.2.) В такой конфигурации каждая транзакция будет ожидать подтверждение того, что на резервном сервере произошла запись транзакции в надёжное хранилище. Значение `synchronous_commit` может быть выставлено для отдельного пользователя, может быть прописано в файле конфигурации, для конкретного пользователя или БД или динамически изменено приложением для управления степенью надёжности на уровне отдельных транзакций.

После сохранения записи о фиксации транзакции на диске ведущего сервера эта запись WAL передаётся резервному серверу. Резервный сервер отвечает подтверждающим сообщением после сохранения каждого нового блока данных WAL на диске, если только `wal_receiver_status_interval` на нём не равен нулю. В случае, когда выбран режим `synchronous_commit remote_apply`, резервный сервер передаёт подтверждение после воспроизведения записи фиксации, когда транзакция становится видимой. Если резервный сервер выбран на роль синхронного резервного в соответствии со значением `synchronous_standby_names` на ведущем, подтверждающие сообщения с этого сервера, в совокупности с сообщениями с других синхронных серверов, будут сигналом к завершению ожидания при фиксировании транзакций, требующих подтверждения сохранения записи фиксации. Эти параметры позволяют администратору определить, какие резервные серверы будут синхронными резервными. Заметьте, что настройка синхронной репликации в основном осуществляется на главном сервере. Перечисленные в списке резервных серверы должны быть подключены к нему непосредственно; он ничего не знает о резервных серверах, подключённых каскадно, через промежуточные серверы.

Если `synchronous_commit` имеет значение `remote_write`, то в случае подтверждения транзакции ответ от резервного сервера об успешном подтверждении будет передан, когда данные запишутся в операционной системе, но не когда данные будет реально сохранены на диске. При таком значении уровень надёжности снижается по сравнению со значением `on`. Резервный сервер может потерять данные в случае падения операционной системы, но не в случае падения Postgres Pro. Тем не менее этот вариант полезен на практике, так как позволяет сократить время отклика для транзакции. Потеря данных может произойти только в случае одновременного сбоя ведущего и резервного, осложнённого повреждением БД на ведущем.

Если `synchronous_commit` имеет значение `remote_apply`, то для завершения фиксирования транзакции потребуется дождаться, чтобы текущие синхронные резервные серверы сообщили, что они воспроизвели транзакцию и её могут видеть запросы пользователей. В простых случаях это позволяет обеспечить обычный уровень согласованности и распределение нагрузки.

Пользователи прекратят ожидание в случае запроса на быструю остановку сервера. В то время как при использовании асинхронной репликации сервер не будет полностью остановлен, пока все исходящие записи WAL не переместятся на текущий присоединённый резервный сервер.

25.2.8.2. Несколько синхронных резервных серверов

Синхронная репликация поддерживает применение одного или нескольких синхронных резервных серверов; транзакции будут ждать, пока все резервные серверы, считающиеся синхронными, не подтвердят получение своих данных. Число синхронных резервных серверов, от которых транзакции должны ждать подтверждения, задаётся в параметре `synchronous_standby_names`. В этом параметре также задаётся список имён резервных серверов и метод (`FIRST` или `ANY`) выбора синхронных из заданного списка.

С методом `FIRST` производится синхронная репликация на основе приоритетов, когда транзакции фиксируются только после того, как их записи в WAL реплицируются на заданное число синхронных резервных серверов, выбираемых согласно приоритетам. Серверы, имена которых идут в начале списка, имеют больший приоритет и выбираются на роль синхронных. Другие резервные серверы, идущие в этом списке за ними, считаются потенциальными синхронными. Если один из текущих синхронных резервных серверов по какой-либо причине отключается, он будет немедленно заменён следующим по порядку резервным сервером.

Пример значения `synchronous_standby_names` для нескольких синхронных резервных серверов, выбираемых по приоритетам:

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

В данном примере, если работают четыре резервных сервера `s1`, `s2`, `s3` и `s4`, два сервера `s1` и `s2` будут выбраны на роль синхронных резервных, так как их имена идут в начале этого списка. Сервер `s3` будет потенциальным резервным и возьмёт на себя роль синхронного резервного при отказе `s1` или `s2`. Сервер `s4` будет асинхронным резервным, так как его имя в этом списке отсутствует.

С методом `ANY` производится синхронная репликация на основе кворума, когда транзакции фиксируются только после того, как их записи в WAL реплицируются на *как минимум* заданное число синхронных серверов в списке.

Пример значения `synchronous_standby_names` для нескольких синхронных резервных серверов, образующих кворум:

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

В данном примере, если работают четыре резервных сервера `s1`, `s2`, `s3` и `s4`, транзакции будут фиксироваться только после получения ответов как минимум от двух резервных серверов из `s1`, `s2` и `s3`. Сервер `s4` будет асинхронным резервным, так как его имя в этом списке отсутствует.

Состояние синхронности резервных серверов можно увидеть в представлении `pg_stat_replication`.

25.2.8.3. Планирование производительности

Организуя синхронную репликацию, обычно нужно обстоятельно обдумать конфигурацию и размещение резервных серверов, чтобы обеспечить приемлемую производительность приложений. Ожидание не потребляет системные ресурсы, но блокировки транзакций будут сохраняться до подтверждения передачи. Как следствие, непродуманное использование синхронной репликации приведёт к снижению производительности БД из-за увеличения времени отклика и числа конфликтов.

Postgres Pro позволяет разработчикам выбрать требуемый уровень надёжности, обеспечиваемый при репликации. Он может быть установлен для системы в целом, для отдельного пользователя или соединения или даже для отдельной транзакции.

Например, в рабочей нагрузке приложения 10% изменений могут относиться к важным данным клиентов, а 90% — к менее критичным данным, потеряв которые, бизнес вполне сможет выжить (например, это могут быть текущие разговоры пользователей между собой).

При настройке уровня синхронности репликации на уровне приложения (на ведущем) можно задать синхронную репликацию для большинства важных изменений без замедления общего рабочего ритма. Возможность настройки на уровне приложения является важным и практичным средством для получения выгод синхронной репликации при высоком быстродействии.

Следует иметь в виду, что пропускная способность сети должна быть больше скорости генерирования данных WAL.

25.2.8.4. Планирование отказоустойчивости

В `synchronous_standby_names` задаётся количество и имена синхронных резервных серверов, от которых будет ожидаться подтверждение при фиксировании транзакции, когда параметру `synchronous_commit` присвоено значение `on`, `remote_apply` или `remote_write`. Фиксирование транзакции в таком режиме может не завершиться никогда, если один из синхронных резервных серверов выйдет из строя.

Поэтому для высокой степени доступности лучше всего обеспечить наличие синхронных резервных серверов в должном количестве. Для этого можно перечислить несколько потенциальных резервных серверов в строке `synchronous_standby_names`.

При синхронной репликации на основе приоритетов синхронными резервными серверами станут серверы, имена которых стоят в этом списке первыми. Следующие за ними серверы будут становиться синхронными резервными при отказе одного из текущих.

При синхронной репликации на основе кворума кандидатами на роль синхронных резервных будут все серверы в списке. И если один из них откажет, другие серверы будут продолжать исполнять эту роль.

Когда к ведущему серверу впервые присоединяется резервный, он ещё не будет полностью синхронизированным. Это называется состоянием `навёрстывания`. Как только отставание резервного от ведущего сервера сократится до нуля в первый раз, система перейдет в состояние `потоковой передачи` в реальном времени. Сразу после создания резервного сервера навёрстывание может быть длительным. В случае выключения резервного сервера длительность этого процесса увеличится соответственно продолжительности простоя. Резервный сервер может стать синхронным только по достижении состояния `потоковой передачи`. Это состояние можно проследить в представлении `pg_stat_replication`.

Если ведущий сервер перезапускается при наличии зафиксированных транзакций, ожидающих подтверждения, эти транзакции будут помечены как полностью зафиксированные после восстановления ведущего. При этом нельзя гарантировать, что все резервные серверы успели получить все текущие данные WAL к моменту падения ведущего. Таким образом, некоторые транзакции могут считаться незафиксированными на резервном сервере, даже если они считаются зафиксированными на ведущем. Гарантия, которую мы можем дать, состоит в том, что приложение не получит явного подтверждения успешной фиксации, пока не будет уверенности, что данные WAL получены всеми синхронными резервными серверами.

Если запустить синхронные резервные серверы в указанном количестве не удаётся, вам следует уменьшить число синхронных серверов, подтверждения которых требуются для завершения фиксации транзакций, в параметре `synchronous_standby_names` (или вовсе отключить его) и перезагрузить файл конфигурации на ведущем сервере.

В случае если ведущий сервер стал недоступным для оставшихся резервных, следует переключиться на наиболее подходящий из имеющихся резервных серверов.

Если необходимо пересоздать резервный сервер при наличии ожидающих подтверждения транзакций, функции `pg_backup_start()` и `pg_backup_stop()` нужно вызывать в сеансе с параметром `synchronous_commit = off`, в противном случае эти запросы будут бесконечно ждать появления резервного сервера.

25.2.9. Непрерывное архивирование на резервном сервере

Когда на резервном сервере применяется непрерывное архивирование WAL, возможны два различных сценария: архив WAL может быть общим для ведущего и резервного сервера, либо резервный сервер может иметь собственный архив WAL. Когда резервный работает с собственным архивом WAL, установите в `archive_mode` значение `always`, и он будет вызывать команду архивации для каждого сегмента WAL, который он получает при восстановлении из архива или потоковой репликации. В случае с общим архивом можно поступить аналогично, но `archive_command` или `archive_library` должна проверять, нет ли в архиве файла, идентичного архивируемому. Таким образом, команда `archive_command` или библиотека `archive_library` должна позаботиться о том, чтобы существующий файл не был заменён файлом с другим содержимым, а в случае попытки повторного архивирования должна сообщать об успешном выполнении. При этом все эти действия должны быть рассчитаны на условия гонки, возможные, если два сервера попытаются архивировать один и тот же файл одновременно.

Если в `archive_mode` установлено значение `on`, архивация в режиме восстановления или резерва не производится. В случае повышения резервного сервера, он начнёт архивацию после повышения, но в архив не попадут те файлы WAL или файлы истории линии времени, которые генерировал не он сам. Поэтому, чтобы в архиве оказался полный набор файлов WAL, необходимо обеспечить архивацию всех файлов WAL до того, как они попадут на резервный сервер. Это естественным образом происходит при трансляции файлов журналов, так как резервный сервер может восстановить только файлы, которые находятся в архиве, однако при потоковой репликации это не так. Когда сервер работает не в режиме резерва, различий между режимами `on` и `always` нет.

25.3. Отработка отказа

Если ведущий сервер отказывает, резервный должен начать процедуры отработки отказа.

Если отказывает резервный сервер, никакие действия по отработке отказа не требуются. Если резервный сервер будет перезапущен, даже через некоторое время, немедленно начнётся операция восстановления, благодаря возможности возобновляемого восстановления. Если вернуть резервный сервер в строй невозможно, необходимо создать полностью новый экземпляр резервного сервера.

Когда ведущий сервер отказывает и резервный сервер становится новым ведущим, а затем старый ведущий включается снова, необходим механизм для предотвращения возврата старого к роли ведущего. Иногда его называют STONITH (Shoot The Other Node In The Head, «Выстрелите в голову другому узлу»), что позволяет избежать ситуации, когда обе системы считают себя ведущими, и в результате возникают конфликты и потеря данных.

Во многих отказоустойчивых конструкциях используются всего две системы: ведущая и резервная, с некоторым контрольным механизмом, который постоянно проверяет соединение между ними и работоспособность ведущей. Также возможно применение третьей системы (называемой следящим сервером) для исключения некоторых вариантов нежелательной отработки отказа, но эта дополнительная сложность оправдана, только если вся схема достаточно хорошо продумана и тщательно протестирована.

Postgres Pro не предоставляет системного программного обеспечения, необходимого для определения сбоя на ведущем и уведомления резервного сервера баз данных. Имеется множество подобных инструментов, которые хорошо интегрируются со средствами операционной системы, требуемыми для успешной отработки отказа, например, для миграции IP-адреса.

Когда происходит переключение на резервный сервер, только один сервер продолжает работу. Это состояние называется ущербным. Бывший резервный сервер теперь является ведущим, а бывший ведущий отключён и может оставаться отключённым. Для возвращения к нормальному состоянию необходимо запустить новый резервный сервер, либо на бывшем ведущем, либо в третьей, возможно, новой системе. Ускорить этот процесс в больших кластерах позволяет утилита pg_rewind. По завершении этого процесса можно считать, что ведущий и резервный сервер поменялись ролями. Некоторые используют третий сервер в качестве запасного для нового ведущего, пока не будет воссоздан новый резервный сервер, хотя это, очевидно, усложняет конфигурацию системы и рабочие процедуры.

Таким образом, переключение с ведущего сервера на резервный может быть быстрым, но требует некоторого времени для повторной подготовки отказоустойчивого кластера. Регулярные переключения с ведущего сервера на резервный полезны, так как при этом появляется плановое время для отключения и проведения обслуживания. Это также позволяет убедиться в работоспособности механизма отработки отказа и гарантировать, что он действительно будет работать, когда потребуется. Эти административные процедуры рекомендуется документировать письменно.

Если выбрана синхронизация слотов логической репликации (см. Подраздел 47.2.3), то перед переключением на резервный сервер рекомендуется проверить, готовы ли к отработке отказа логические слоты, синхронизированные на резервном сервере. Чтобы это сделать, выполните шаги, описанные в Раздел 28.3.

Чтобы запустить отработку отказа резервного сервера трансляции журналов, запустите `pg_ctl promote` или вызовите `pg_promote()`. Если резервный сервер применяется для анализа данных, чтобы только разгрузить ведущий, выполняя запросы на чтение, а не обеспечивать отказоустойчивость, повышать его до ведущего не понадобится.

25.4. Горячий резерв

Термин «горячий резерв» используется для описания возможности подключаться к серверу и выполнять запросы на чтение, в то время как сервер находится в режиме резерва или восстановления архива. Это полезно и для целей репликации, и для восстановления желаемого состояния из резервной копии с высокой точностью. Так же термин «горячий резерв» описывает способность сервера переходить из режима восстановления к обычной работе, в то время как пользователи продолжают выполнять запросы и/или их соединения остаются открытыми.

В режиме горячего резерва запросы выполняются примерно так же, как и в обычном режиме, с некоторыми отличиями в использовании и администрировании, описанными ниже.

25.4.1. Обзор на уровне пользователя

Когда параметр hot_standby на резервном сервере установлен в true, то он начинает принимать соединения сразу как только система придёт в согласованное состояние в процессе восстановления. Для таких соединений будет разрешено только чтение, запись невозможна даже во временные таблицы.

Для того, чтобы данные с ведущего сервера были получены на резервном, требуется некоторое время. Таким образом, имеется измеряемая задержка между ведущим и резервным серверами. Поэтому запуск одинаковых запросов примерно в одно время на ведущем и резервном серверах может вернуть разный результат. Можно сказать, что данные на резервном сервере *в конечном счёте согласуются* с ведущим. После того как запись о зафиксированной транзакции воспроизводится на резервном сервере, изменения, совершённые в этой транзакции, становится видны в любых последующих снимках данных на резервном сервере. Снимок может быть сделан в начале каждого запроса или в начале каждой транзакции в зависимости от уровня изоляции транзакции. Более подробно см. Раздел 13.2.

Транзакции, запущенные в режиме горячего резерва, могут выполнять следующие команды:

- Доступ к данным: `SELECT`, `COPY TO`
- Команды для работы с курсором: `DECLARE`, `FETCH`, `CLOSE`
- Параметры: `SHOW`, `SET`, `RESET`
- Команды явного управления транзакциями:
  - `BEGIN`, `END`, `ABORT`, `START TRANSACTION`
  - `SAVEPOINT`, `RELEASE`, `ROLLBACK TO SAVEPOINT`
  - Блок `EXCEPTION` и другие внутренние подчиненные транзакции
- `LOCK TABLE`, только когда исполняется в явном виде в следующем режиме: `ACCESS SHARE`, `ROW SHARE` или `ROW EXCLUSIVE`.
- Планы и ресурсы: `PREPARE`, `EXECUTE`, `DEALLOCATE`, `DISCARD`
- Дополнения и расширения: `LOAD`
- `UNLISTEN`

Транзакции, запущенные в режиме горячего резерва, никогда не получают ID транзакции и не могут быть записаны в журнал предзаписи. Поэтому при попытке выполнить следующие действия возникнут ошибки:

- Команды манипуляции данными (DML): `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `COPY FROM`, `TRUNCATE`. Следует отметить, что нет разрешённых действий, которые приводили бы к срабатыванию триггера во время исполнения на резервном сервере. Это ограничение так же касается и временных таблиц, так как строки таблицы не могут быть прочитаны или записаны без обращения к ID транзакции, что в настоящее время не возможно в среде горячего резерва.
- Команды определения данных (DDL): `CREATE`, `DROP`, `ALTER`, `COMMENT`. Эти ограничения так же относятся и к временным таблицам, так как операции могут потребовать обновления таблиц системных каталогов.
- `SELECT ... FOR SHARE | UPDATE`, так как блокировка строки не может быть проведена без обновления соответствующих файлов данных.
- Правила для выражений `SELECT`, которые приводят к выполнению команд DML.
- `LOCK` которая явно требует режим более строгий чем `ROW EXCLUSIVE MODE`.
- `LOCK` в короткой форме с умолчаниями, так как требует `ACCESS EXCLUSIVE MODE`.
- Команды управления транзакциями, которые в явном виде требуют режим не только для чтения
  - `BEGIN READ WRITE`, `START TRANSACTION READ WRITE`
  - `SET TRANSACTION READ WRITE`, `SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE`
  - `SET transaction_read_only = off`
- Команды двухфазной фиксации: `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED`, так как даже транзакции только для чтения нуждаются в записи в WAL на этапе подготовки (первая фаза двухфазной фиксации).
- Обновление последовательностей: `nextval()`, `setval()`
- `LISTEN`, `NOTIFY`

При обычной работе транзакции «только для чтения» могут использовать команды `LISTEN` и `NOTIFY`; таким образом, сеансы горячего резерва работают с несколько большими ограничениями, чем обычные только читающие сеансы. Возможно, что некоторые из этих ограничений будут ослаблены в следующих выпусках.

В режиме горячего резерва параметр `transaction_read_only` всегда имеет значение true и изменить его нельзя. Но если не пытаться модифицировать содержимое БД, подключение к серверу в этом режиме не отличается от подключений к обычным базам данных. При отработке отказа или переключении ролей база данных переходит в обычный режим работы. Когда сервер меняет режим работы, установленные сеансы остаются подключёнными. После выхода из режима горячего резерва становится возможным запускать пишущие транзакции (даже в сеансах, начатых ещё в режиме горячего резерва).

Пользователи могут узнать, активен ли режим горячего резерва в их сеансе, выполнив команду `SHOW in_hot_standby`. (Параметр `in_hot_standby` появился в 14 версии сервера; с более старыми серверами работает другой вариант — `SHOW transaction_read_only`.) Кроме того, пользователи могут получить информацию о резервном сервере, используя ряд функций (см. Таблицу 9.96). Эти средства позволяют как создавать программы, учитывающие текущий статус базы данных, так и отслеживать процесс восстановления или разрабатывать более сложные программы для восстановления баз данных в нужном состоянии.

25.4.2. Обработка конфликтов запросов

Ведущий и резервный серверы связаны между собой многими слабыми связями. События на ведущем сервере оказывают влияние на резервный. В результате имеется потенциальная возможность отрицательного влияния или конфликта между ними. Наиболее простой для понимания конфликт — быстродействие: если на ведущем происходит загрузка очень большого объёма данных, то происходит создание соответствующего потока записей WAL на резервный сервер. Таким образом, запросы на резервном конкурируют за системные ресурсы, например, ввод-вывод.

Так же может возникнуть дополнительный тип конфликта на сервере горячего резерва. Этот конфликт называется *жёстким конфликтом*, оказывает влияние на запросы, приводя к их отмене, а в некоторых случаях и к обрыву сеанса для разрешения конфликтов. Пользователям предоставлен набор средств для обработки подобных конфликтов. Случаи конфликтов включают:

- Установка эксклюзивной блокировки на ведущем сервере, как с помощью явной команды `LOCK`, так и при различных DDL, что приводит к конфликту доступа к таблицам на резервном.
- Удаление табличного пространства на ведущем сервере приводит к конфликту на резервном когда запросы используют это пространство для хранения временных рабочих файлов.
- Удаление базы данных на ведущем сервере конфликтует с сеансами, подключёнными к этой БД на резервном.
- Приложение очистки устаревших транзакций из WAL конфликтует с транзакциями на резервном сервере, которые используют снимок данных, который всё ещё видит какие-то из очищенных на ведущем строк.
- Приложение очистки устаревших транзакций из WAL конфликтует с запросами к целевой странице на резервном сервере вне зависимости от того, являются ли данные удалёнными или видимыми.

В этих случаях на ведущем сервере просто происходит ожидание; пользователю следует выбрать какую их конфликтующих сторон отменить. Тем не менее на резервном нет выбора: действия из WAL уже произошли на ведущем, поэтому резервный обязан применить их. Более того, позволять обработчику WAL ожидать неограниченно долго может быть крайне нежелательно, так как отставание резервного сервера от ведущего может всё возрастать. Таким образом, механизм обеспечивает принудительную отмену запросов на резервном сервере, которые конфликтуют с применяемыми записями WAL.

Примером такой проблемы может быть ситуация: администратор на ведущем сервере выполнил команду `DROP TABLE` для таблицы, которая сейчас участвует в запросе на резервном. Понятно, что этот запрос нельзя будет выполнять дальше, если команда `DROP TABLE` применится на резервном. Если бы этот запрос выполнялся на ведущем, команда `DROP TABLE` ждала бы его окончания. Но когда на ведущем выполняется только команда `DROP TABLE`, ведущий сервер не знает, какие запросы выполняются на резервном, поэтому он не может ждать завершения подобных запросов. Поэтому если записи WAL с изменением прибудут на резервный сервер, когда запрос будет продолжать выполняться, возникнет конфликт. В этом случае резервный сервер должен либо задержать применение этих записей WAL (и всех остальных, следующих за ними), либо отменить конфликтующий запрос, чтобы можно было применить `DROP TABLE`.

Если конфликтный запрос короткий, обычно желательно разрешить ему завершиться, ненадолго задержав применение записей WAL, но слишком большая задержка в применении WAL обычно нежелательна. Поэтому механизм отмены имеет параметры max_standby_archive_delay и max_standby_streaming_delay, которые определяют максимально допустимое время задержки применения WAL. Конфликтующие запросы будут отменены, если они длятся дольше допустимого времени задержки применения очередных записей WAL. Два параметра существуют для того, чтобы можно было задать разные значения для чтения записей WAL из архива (то есть при начальном восстановлении из базовой копии либо при «навёрстывании» ведущего сервера в случае большого отставания) и для получения записей WAL при потоковой репликации.

На резервном сервере, созданном преимущественно для отказоустойчивости, лучше выставлять параметры задержек относительно небольшими, чтобы он не мог сильно отстать от ведущего из-за задержек, связанных с ожиданием запросов горячего резерва. Однако если резервный сервер предназначен для выполнения длительных запросов, то высокое значение или даже бесконечное ожидание могут быть предпочтительнее. Тем не менее следует иметь в виду, что длительные запросы могут оказать влияние на другие сеансы на резервном сервере в виде отсутствия последних изменений от ведущего из-за задержки применения записей WAL.

В случае, если задержка, определённая `max_standby_archive_delay` или `max_standby_streaming_delay` будет превышена, конфликтующий запрос будет отменён. Обычно это выражается в виде ошибки отмены, но в случае проигрывания команды `DROP DATABASE` обрывается весь конфликтный сеанс. Так же, если конфликт произошел при блокировке, вызванной транзакцией в состоянии IDLE, конфликтный сеанс разрывается (это поведение может изменить в будущем).

Отменённые запросы могут быть немедленно повторены (конечно после старта новой транзакции). Так как причина отмены зависит от природы проигрываемых записей WAL, запрос, который был отменён, может быть успешно выполнен вновь.

Следует учесть, что параметры задержки отсчитываются от времени получения резервным сервером данных WAL. Таким образом, период дозволенной работы для запроса на резервном сервере никогда не может быть длиннее параметра задержки и может быть существенно короче, если резервный уже находится в режиме задержки в результате ожидания предыдущего запроса или результат не доступен из-за высокой нагрузки обновлений.

Наиболее частой причиной конфликтов между запросами на резервном сервере и проигрыванием WAL является преждевременная очистка. Обычно Postgres Pro допускает очистку старых версий записей при условии что ни одна из транзакций их не видит согласно правилам видимости данных для MVCC. Тем не менее эти правила применяются только для транзакций, выполняемых на главном сервере. Таким образом, допустима ситуация, когда на главном запись уже очищена, но эта же запись всё ещё видна для транзакций на резервном сервере.

Очистка старых версий строк — не единственная потенциальная причина конфликтов с запросами на резервном сервере. Все сканирования только индекса (включая те, которые выполняются на резервных серверах) должны использовать снимок MVCC, который «согласуется» с картой видимости. Поэтому конфликты необходимы всякий раз, когда `VACUUM` помечает страницу, в которой есть одна или несколько строк, *невидимых* для всех запросов резервного сервера, как полностью видимую в карте видимости. Таким образом, даже выполнение `VACUUM` для таблицы без изменённых или удалённых строк, требующих очистки, может привести к конфликтам.

Пользователи должны понимать, что регулярное и активное изменение данных в таблицах на ведущем сервере чревато отменой длительных запросов на резервном. В таком случае установка конечного значения для `max_standby_archive_delay` или `max_standby_streaming_delay` действует подобно ограничению `statement_timeout`.

В случае, если количество отменённых запросов на резервном сервере получается неприемлемым, существует ряд дополнительных возможностей. Первая возможность — установить параметр `hot_standby_feedback`, который не даёт команде `VACUUM` удалять записи, ставшие недействительными недавно, что предотвращает конфликты очистки. При этом следует учесть, что это вызывает задержку очистки мёртвых строк на ведущем, что может привести к нежелательному раздуванию таблицы. Тем не менее в итоге ситуация будет не хуже, чем если бы запросы к резервному серверу исполнялись непосредственно на ведущем, но при этом сохранится положительный эффект от разделения нагрузки. В случае, когда соединение резервных серверов с ведущим часто разрывается, следует скорректировать период, в течение которого обратная связь через `hot_standby_feedback` не обеспечивается. Например, следует подумать об увеличении `max_standby_archive_delay`, чтобы запросы отменялись не сразу при конфликтах с архивом WAL в период разъединения. Также может иметь смысл увеличить `max_standby_streaming_delay` для предотвращения быстрой отмены запросов из-за полученных записей WAL после восстановления соединения.

Количество отменённых запросов и причины отмены можно просмотреть через системное представление `pg_stat_database_conflicts` на резервном сервере. Системное представление `pg_stat_database` так же содержит итоговую информацию.

Используя параметр log_recovery_conflict_waits, пользователь может регулировать, вносятся ли в журнал сообщения о том, что при воспроизведении WAL время ожидания разрешения конфликта превысило `deadlock_timeout`.

25.4.3. Обзор административной части

Если в файле `postgresql.conf` параметр `hot_standby` имеет значение `on` (значение по умолчанию) и существует файл `standby.signal`, сервер запустится в режиме горячего резерва. Однако может пройти некоторое время, прежде чем к нему можно будет подключиться, так как он не будет принимать подключения, пока не произведёт восстановление до согласованного состояния, подходящего для выполнения запросов. (Информация о согласованности состояния записывается на ведущем сервере в контрольной точке.) В течение этого периода клиенты при попытке подключения будут получать сообщение об ошибке. Убедиться, что сервер включился в работу, можно либо повторяя попытки подключения из приложения до успешного подключения, либо дождавшись появления в журналах сервера таких сообщений:

```
LOG:  entering standby mode

... then some time later ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read-only connections
```

Включить горячий резерв нельзя, если WAL был записан в период, когда на ведущем сервере параметр `wal_level` имел значение, отличное от `replica` и `logical`. Достижение согласованного состояния также может быть отсрочено, если имеют место оба этих условия:

- Пишущая транзакция имеет более 64 подтранзакций
- Очень длительные пишущие транзакции

Если вы применяете файловую репликацию журналов («тёплый резерв»), возможно, придётся ожидать прибытия следующего файла WAL (максимальное время ожидания задаётся параметром `archive_timeout` на ведущем сервере).

Значения некоторых параметров определяют размер общей памяти для отслеживания идентификаторов транзакций, блокировок и подготовленных транзакций. Эти структуры общей памяти на резервном сервере должны быть не меньше, чем аналогичные структуры на ведущем, чтобы гарантировать, что резервный сервер не исчерпает общую память во время восстановления. Например, если на ведущем сервере выполнялась подготовленная транзакция, но резервный сервер не выделил никакой общей памяти для отслеживания подготовленных транзакций, то восстановление не сможет продолжаться пока не будет изменена конфигурация резервного сервера. Это касается следующих параметров:

- `max_connections`
- `max_prepared_transactions`

- `max_locks_per_transaction`
- `max_wal_senders`
- `max_worker_processes`

Самый простой способ избежать данной проблемы — установить на резервных серверах такие значения этих параметров, которые будут больше значений на основном или равны им. Поэтому, если вы хотите увеличить эти значения, сначала вы должны сделать это на всех резервных серверах, а затем внести изменения на ведущем сервере. И наоборот, если вы хотите уменьшить эти значения, сначала следует сделать это на ведущем сервере, а затем вносить изменения на всех резервных серверах. Имейте в виду, что когда резервный сервер повышается до ведущего, он становится новым источником необходимых значений параметров для следующих за ним резервных серверов. Чтобы это не стало проблемой во время переключения узлов или отработки отказа, рекомендуется устанавливать одинаковые значения на всех резервных серверах.

Изменения этих параметров на ведущем сервере отслеживаются в WAL. Если в процессе чтения WAL сервер горячего резерва обнаруживает, что на ведущем сервере текущее значение оказалось больше, чем значение в его собственной конфигурации, он выдаёт предупреждение и приостанавливает восстановление, например:

```
WARNING:  hot standby is not possible because of insufficient parameter settings
DETAIL:  max_connections = 80 is a lower setting than on the primary server, where its
 value was 100.
LOG:  recovery has paused
DETAIL:  If recovery is unpaused, the server will shut down.
HINT:  You can then restart the server after making the necessary configuration
 changes.
```

Прежде чем продолжить восстановление, на этом этапе необходимо изменить значения на резервном сервере и перезапустить экземпляр СУБД. Если резервный сервер не является сервером горячего резерва, то при обнаружении несовместимого изменения параметра он немедленно отключится без приостановки, поскольку в этом случае нет смысла поддерживать его в рабочем состоянии.

Очень важно для администратора выбрать подходящие значения для max_standby_archive_delay и max_standby_streaming_delay. Оптимальное значение зависит от приоритетов. Например, если основное назначение сервера — обеспечение высокой степени доступности, то следует установить короткий период, возможно даже нулевой, хотя это очень жёсткий вариант. Если резервный сервер планируется как дополнительный сервер для аналитических запросов, то приемлемой будет максимальная задержка в несколько часов или даже -1, что означает бесконечное ожидание окончания запроса.

Вспомогательные биты статуса транзакций, записанные на ведущем, не попадают в WAL, так что они, скорее всего, будут перезаписаны на нём при работе с данными. Таким образом, резервный сервер будет производить запись на диск, даже если все пользователи только читают данные, ничего не меняя. Кроме того, пользователи будут записывать временные файлы при сортировке больших объёмов и обновлять файлы кеша. Поэтому в режиме горячего резерва ни одна часть базы данных фактически не работает в режиме «только чтение». Следует отметить, что также возможно выполнить запись в удалённую базу данных с помощью модуля dblink и другие операции вне базы данных с применением PL-функций, несмотря на то, что транзакции по-прежнему смогут только читать данные.

Следующие типы административных команд недоступны в течение режима восстановления:

- Команды определения данных (DDL): `CREATE INDEX` и т. п.
- Команды управления правами и назначения владельца: `GRANT`, `REVOKE`, `REASSIGN`
- Команды обслуживания: `ANALYZE`, `VACUUM`, `CLUSTER`, `REINDEX`

Ещё раз следует отметить, что некоторые из этих команд фактически доступны на ведущем сервере для транзакций в режиме только для чтения.

В результате нельзя создать дополнительные индексы или статистику, чтобы они существовали только на резервном. Если подобные административные команды нужны, то их следует выполнить на ведущем сервере, затем эти изменения будут распространены на резервные серверы.

Функции `pg_cancel_backend()` и `pg_terminate_backend()` работают на стороне пользователя, но не для процесса запуска, который обеспечивает восстановление. Представление `pg_stat_activity` не показывает восстанавливаемые транзакции как активные. Поэтому представление `pg_prepared_xacts` всегда пусто в ходе восстановления. Если требуется разобрать сомнительные подготовленные транзакции, следует обратиться к `pg_prepared_xacts` на ведущем и выполнить команды для разбора транзакций там либо разобрать их по окончании восстановления.

`pg_locks` отображает блокировки, происходящие в процессе работы сервера как обычно. `pg_locks` так же показывает виртуальные транзакции, обработанные процессом запуска, которому принадлежат все `AccessExclusiveLocks`, наложенные транзакциями в режиме восстановления. Следует отметить, что процесс запуска не запрашивает блокировки, чтобы внести изменения в базу данных, поэтому блокировки, отличные от `AccessExclusiveLocks` не показываются в `pg_locks` для процесса запуска, подразумевается их существование.

Модуль check_pgsql для Nagios будет работать, так как сервер выдаёт простую информацию, наличие которой он проверяет. Скрипт мониторинга check_postgres так же работает, хотя для некоторых выдаваемых показателей результаты могут различаться или вводить в заблуждение. Например, нельзя отследить время последней очистки, так как очистка не производится на резервном сервере. Очистка запускается на ведущем сервере и результаты её работы передаются резервному.

Команды управления файлами WAL, например `pg_backup_start`, `pg_switch_wal` и т. д. не будут работать во время восстановления.

Динамически загружаемые модули работать будут, включая `pg_stat_statements`.

Рекомендательная блокировка работает обычно при восстановлении, включая обнаружение взаимных блокировок. Следует отметить, что рекомендательная блокировка никогда не попадает в WAL, таким образом для рекомендательной блокировки как на ведущем сервере, так и на резервном, невозможен конфликт с проигрыванием WAL. Но возможно получение рекомендательной блокировки на ведущем сервере, а затем получение подобной рекомендательной блокировки на резервном. Рекомендательная блокировка относится только к серверу, на котором она получена.

Системы репликации на базе триггеров, подобные Slony, Londiste и Bucardo не могут запускаться на резервном сервере вовсе, хотя они превосходно работают на ведущем до тех пор, пока не будет подана команда не пересылать изменения на резервный. Проигрывание WAL не основано на триггерах, поэтому поток WAL нельзя транслировать с резервного сервера в другую систему, которая требует дополнительной записи в БД или работает на основе триггеров.

Новые OID не могут быть выданы, хотя, например генераторы UUID смогут работать, если они не пытаются записывать новое состояние в базу данных.

На данный момент создание временных таблиц недопустимо в транзакции только для чтения, поэтому в некоторых случаях существующие скрипты будут работать некорректно. Сейчас это не поддерживается как из соображений совместимости со стандартом SQL, так и из соображений технического плана.

Команда `DROP TABLESPACE` может быть выполнена только если табличное пространство пусто. Некоторые пользователи резервного сервера могут активно использовать табличное пространство через параметр `temp_tablespaces`. Если имеются временные файлы в табличных пространствах, все активные запросы отменяются для обеспечения удаления временных файлов, затем табличное пространство может быть удалено и продолжено проигрывание WAL.

Выполнение команды `DROP DATABASE` или `ALTER DATABASE ... SET TABLESPACE` на ведущем сервере приводит к созданию записи в WAL, которая вызывает принудительное отключение всех пользователей, подключённых к этой базе данных на резервном. Это происходит немедленно, вне зависимости от значения `max_standby_streaming_delay`. Следует отметить, что команда `ALTER DATABASE ... RENAME` не приводит к отключению пользователей, так что обычно она действует незаметно, хотя в некоторых случаях возможны сбои программ, которые зависят от имени базы данных.

Если вы в обычном режиме (не в режиме восстановления) выполните `DROP USER` или `DROP ROLE` для роли с возможностью подключения, в момент, когда этот пользователь подключён, на данном пользователе это никак не отразится — он останется подключённым. Однако переподключиться он уже не сможет. Это же поведение действует в режиме восстановления — если выполнить `DROP USER` на ведущем сервере, пользователь не будет отключён от резервного.

Система накопительной статистики работает во время восстановления. Все операции сканирования, чтения, использование блоков, индексов и т. п. будут отражены в статистике на резервном сервере обычным образом. Однако при воспроизведении WAL не будут увеличиваться счётчики уровня отношений и баз данных, то есть при воспроизведении не будут увеличиваться значения столбцов в `pg_stat_all_tables` (например, `n_tup_ins`), операции чтения и записи, выполняемые процессом запуска, не будут отражаться в представлениях `pg_statio_`, а также не будут увеличиваться значения связанных столбцов `pg_stat_database`.

Автоматическая очистка не работает во время восстановления. Она запустится в обычном режиме после завершения восстановления.

Во время восстановления работает процесс контрольных точек и процесс фоновой записи. Процесс контрольных точек обрабатывает точки перезапуска (подобные контрольным точкам на ведущем сервере), а процесс фоновой записи выполняет обычные операции по очистке блоков. В том числе он может обновлять вспомогательные биты, сохранённые на резервном сервере. Во время восстановления принимается команда `CHECKPOINT`, но она производит точку перезапуска, а не создаёт новую точку восстановления.

25.4.4. Ссылки на параметры горячего резерва

Различные параметры были упомянуты выше в Подразделе 25.4.2 и Подразделе 25.4.3.

На ведущем сервере может применяться параметр wal_level. Параметры max_standby_archive_delay и max_standby_streaming_delay на ведущем сервере не действуют.

На резервном сервере могут применяться параметры hot_standby, max_standby_archive_delay и max_standby_streaming_delay.

25.4.5. Ограничения

Имеются следующие ограничения горячего резерва. Они могут и скорее всего будут исправлены в следующих выпусках:

- Требуется информация о всех запущенных транзакциях перед тем как будет создан снимок данных. Транзакции, использующие большое количество подтранзакций (в настоящий момент больше 64), будут задерживать начало соединения только для чтения до завершения самой длинной пишущей транзакции. При возникновении этой ситуации поясняющее сообщение будет записано в журнал сервера.
- Подходящие стартовые точки для запросов на резервном сервере создаются при каждой контрольной точке на главном. Если резервный сервер отключается, в то время как главный был в отключённом состоянии, может оказаться невозможным возобновить его работу в режиме горячего резерва, до того, как запустится ведущий и добавит следующие стартовые точки в журналы WAL. Подобная ситуация не является проблемой для большинства случаев, в которых она может произойти. Обычно, если ведущий сервер выключен и больше не доступен, это является следствием серьёзного сбоя и в любом случае требует преобразования резервного в новый ведущий. Так же в ситуации, когда ведущий отключён намеренно, проверка готовности резервного к преобразованию в ведущий тоже является обычной процедурой.
- В конце восстановления блокировки `AccessExclusiveLocks`, вызванные подготовленными транзакциями, требуют удвоенное, в сравнении с нормальным, количество блокировок за-

писей таблицы. Если планируется использовать либо большое количество конкурирующих подготовленных транзакций, обычно вызывающие `AccessExclusiveLocks`, либо большие транзакции с применением большого количества `AccessExclusiveLocks`, то рекомендуется выбрать большое значение параметра `max_locks_per_transaction`, возможно в два раза большее, чем значение параметра на ведущем сервере. Всё это не имеет значения, когда `max_prepared_transactions` равно 0.

- Уровень изоляции транзакции Serializable в настоящее время недоступен в горячем резерве. (За подробностями обратитесь к Подразделу 13.2.3 и Подразделу 13.4.1) Попытка выставить для транзакции такой уровень изоляции в режиме горячего резерва вызовет ошибку.

Глава 26. Мониторинг работы СУБД

Администратор базы данных часто задумывается — «чем в данный момент занята система?» В этой главе рассказывается о том, как это выяснить.

Для мониторинга работы СУБД и анализа её производительности существуют различные инструменты. Большая часть этой главы посвящена описанию системы накопительной статистики Postgres Pro, однако не следует пренебрегать и обычными командами мониторинга Unix, такими как `ps`, `top`, `iostat`, и `vmstat`. Кроме того, после обнаружения запроса с низкой производительностью может потребоваться дополнительное исследование с использованием Postgres Pro команды `EXPLAIN`. В Разделе 14.1 рассматриваются `EXPLAIN` и другие методы для изучения поведения отдельного запроса.

26.1. Стандартные инструменты Unix

В большинстве Unix-платформ Postgres Pro модифицирует заголовок команды, который выводится на экран при выполнении команды `ps`, так что серверные процессы можно легко различить. Пример вывода этой команды:

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres: background
 writer
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres:
 checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: walwriter
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres: autovacuum
 launcher
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe runbug
 127.0.0.1 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl
 regression [local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl
 regression [local] idle in transaction
```

(Формат вызова `ps`, а также детали отображаемой информации зависят от платформы. Это пример для одной из последних Linux-систем.) Первым здесь отображается главный процесс сервера. Для этого процесса отображены аргументы команды, которые использовались при его запуске. Следующие четыре процесса — это фоновые рабочие процессы, которые были автоматически запущены процессом сервера. (Фонового процесса «autovacuum launcher» в этом списке не будет, если автоочистка отключена в системе.) Во всех остальных строках перечислены серверные процессы, каждый из которых обслуживает одно клиентское подключение. Командная строка каждого такого процесса имеет следующий формат:

`postgres:` *`пользователь база_данных компьютер активность`*

Пользователь, СУБД и компьютер (клиента) остаются неизменными на протяжении всего клиентского подключения, а индикатор деятельности меняется. Возможные виды деятельности: `idle` (т. е. ожидание команды клиента), `idle in transaction` (ожидание клиента внутри блока `BEGIN`) или название типа команды, например, `SELECT`. Кроме того, если в настоящий момент серверный процесс ожидает высвобождения блокировки, которую держит другой сеанс, то к виду деятельности добавляется `waiting`. В приведённом выше примере мы видим, что процесс 15606 ожидает, когда процесс 15610 завершит свою транзакцию и, следовательно, освободит какую-то блокировку. (Процесс 15610 является блокирующим, поскольку никаких других активных сеансов нет. В более сложных случаях может потребоваться обращение к системному представлению `pg_locks`, для того чтобы определить, кто кого блокирует.)

Если установлено значение cluster_name, имя кластера также будет показываться в выводе команды `ps`:

```
$ psql -c 'SHOW cluster_name'
```

```
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00 postgres: server1:
 background writer
...
```

Если параметр update_process_title был отключён, то индикатор деятельности не обновляется; название процесса устанавливается только один раз при запуске нового процесса. На некоторых платформах это позволяет значительно сократить накладные расходы при выполнении команды; на других платформах этот выигрыш может быть незначителен.

> **Подсказка**
>
> В Solaris требуется особый подход. Следует использовать `/usr/ucb/ps` вместо `/bin/ps`. Также следует использовать два флага `w`, а не один. Кроме того, при выводе статусов команд с помощью `ps` статус для исходной команды `postgres` должен отображаться в сокращённом формате для каждого серверного процесса. Если вы не сделаете все три вещи, то вывод `ps` для каждого серверного процесса будет исходной командной строкой `postgres`.

26.2. Система накопительной статистики

Система накопительной статистики в Postgres Pro представляет собой подсистему, которая собирает и отображает информацию о работе сервера. В настоящее время в ней отслеживаются обращения к таблицам и индексам как на уровне блоков на диске, так и на уровне отдельных строк. Также подсчитывается общее число строк в каждой таблице, и собирается информация о выполняемых для каждой таблицы действиях очистки и анализа. Кроме того, может быть включён подсчёт вызовов пользовательских функций и времени, затраченного на выполнение каждой из них.

Кроме того, Postgres Pro может предоставить динамическую информацию о том, что происходит в системе прямо сейчас, в частности, сообщить, какие именно команды выполняются другими серверными процессами и какие другие соединения существуют в системе. Эта возможность не зависит от системы накопительной статистики.

26.2.1. Конфигурация системы сбора статистики

Поскольку сбор статистики несколько увеличивает накладные расходы при выполнении запроса, есть возможность настроить СУБД так, чтобы выполнять или не выполнять сбор статистической информации. Это контролируется конфигурационными параметрами, которые обычно устанавливаются в файле `postgresql.conf`. (Подробно установка конфигурационных параметров описывается в Главе 18.)

Параметр track_activities включает мониторинг текущих команд, выполняемой любым серверным процессом.

Параметр track_counts определяет, будет ли собираться накопительная статистика по обращениям к таблицам и индексам.

Параметр track_functions включает отслеживание использования пользовательских функций.

Параметр track_io_timing включает мониторинг времени чтения, записи и расширения блоков, а также времени их синхронизации с файловой системой.

Параметр track_wal_io_timing включает мониторинг времени записи WAL и времени его синхронизации с файловой системой.

Обычно эти параметры устанавливаются в `postgresql.conf`, поэтому они применяются ко всем серверным процессам, однако, используя команду SET, их можно включать и выключать в отдельных сеансах. (Для того чтобы обычные пользователи не скрывали свою работу от администратора СУБД, изменять эти параметры с помощью команды `SET` могут только суперпользователи.)

Накопительная статистика собирается в общей памяти. Каждый процесс Postgres Pro собирает статистику локально, а затем обновляет общие данные с некоторым интервалом. При штатной остановке сервера, включая физическую реплику, постоянная копия статистической информации сохраняется в подкаталоге `pg_stat`, поэтому статистика не теряется после перезапуска сервера. Если же сервер запускается после аварийной остановки (например, после остановки в режиме `immediate`, отказа сервера, восстановления на момент времени), все счётчики статистики сбрасываются.

26.2.2. Просмотр статистики

Для просмотра текущего состояния системы предназначены несколько предопределённых представлений, которые перечислены в Таблице 26.1. В дополнение к ним есть несколько других представлений, перечисленных в Таблице 26.2, позволяющих просмотреть собранную статистику. Кроме того, на базе нижележащих функций накопительной статистики можно создать собственные представления, как описано в Подразделе 26.2.26.

Используя функции и представления накопительной статистики в целях мониторинга, важно понимать, что эта информация обновляется не сразу. Каждый серверный процесс передаёт собранную статистику в общую память перед переходом в режим ожидания, но не чаще одного раза в интервал, заданный в миллисекундах константой `PGSTAT_MIN_INTERVAL` (1 секунда, если эта константа не менялась при компиляции сервера); то есть запрос или транзакция в процессе выполнения не влияют на отображаемые данные статистики, а отображаемая информация отстаёт от того, что происходит в настоящий момент. Однако информация о текущем запросе, собираемая с параметром `track_activities`, всегда актуальна.

Ещё одним важным моментом является то, что когда в серверном процессе запрашивают какую-либо статистику, в конфигурации по умолчанию полученные значения кешируются до конца текущей транзакции. Так что на протяжении одной транзакции эта статистическая информация меняться не будет. При этом информация о текущих запросах во всех сеансах собирается в тот момент, когда она впервые запрашивается в рамках транзакции, и эта же информация будет отображаться на протяжении всей транзакции. Это не ошибка, а полезное свойство СУБД, поскольку оно позволяет выполнять запросы к статистическим данным и сравнивать результаты, не беспокоясь о том, что статистические данные меняются. При интерактивном анализе статистики, или когда выполняются ресурсоёмкие запросы, получаемая статистика может оказаться смещённой из-за задержки между обращениями к отдельным показателям. Установив для параметра `stats_fetch_consistency` значение `snapshot`, можно минимизировать смещение ценой увеличения объёма памяти для кеширования лишних показателей статистики. И наоборот, когда известно, что обращения к статистике будут единичными, кешировать статистику, к которой уже обращались, не нужно, и этого можно избежать, установив для `stats_fetch_consistency` значение `none`. Сбросить снимок статистики или кешируемые значения (если таковые имеются) в текущей транзакции можно, вызвав функцию `pg_stat_clear_snapshot()`. При следующем обращении к данным статистики будет создан новый снимок (в режиме `snapshot`), или запрошенные данные сохранятся в кеше (в режиме `cache`).

Через представления `pg_stat_xact_all_tables`, `pg_stat_xact_sys_tables`, `pg_stat_xact_user_tables`, и `pg_stat_xact_user_functions` транзакции также доступна её собственная статистика (ещё не переданная в общую память статистики). Данные в этих представлениях ведут себя не так, как описано выше; наоборот, они постоянно обновляются в течение транзакции.

Часть информации в представлениях с динамическими статистическими данными, перечисленных в Таблице 26.1 скрыта по соображениям безопасности. Обычные пользователи могут получать информацию только о своих собственных сеансах (сеансах, принадлежащих роли, членами которой они являются). В строках, относящимся к другим сеансам, многие столбцы будут содержать

NULL. Заметьте однако, что информация о самом сеансе и его общих свойствах, например, текущем пользователе и базе данных, доступна всем пользователям. Суперпользователи и роли с правами встроенной роли `pg_read_all_stats` (см. также Раздел 20.5) могут получить всю информацию о любом сеансе.

Таблица 26.1. Динамические статистические представления

Имя представления	Описание
`pg_stat_activity`	Одна строка для каждого серверного процесса с информацией о текущей активности процесса, включая его состояние и текущий запрос. Подробности в `pg_stat_activity` .
`pg_stat_replication`	По одной строке для каждого процесса-передатчика WAL со статистикой по репликации на ведомом сервере, к которому подключён этот процесс. Подробности в `pg_stat_replication` .
`pg_stat_wal_receiver`	Только одна строка со статистикой приёмника WAL, полученной с сервера, на котором работает приёмник. Подробности в `pg_stat_wal_receiver` .
`pg_stat_recovery_prefetch`	Только одна строка со статистикой блоков, предварительно выбранных во время восстановления. Подробности в `pg_stat_recovery_prefetch` .
`pg_stat_subscription`	Как минимум одна строка для подписки, сообщающая о рабочих процессах подписки. Подробности в `pg_stat_subscription` .
`pg_stat_ssl`	Одна строка для каждого подключения (обычного и реплицирующего), в которой показывается информация об использовании SSL для данного подключения. Подробности в `pg_stat_ssl` .
`pg_stat_gssapi`	Одна строка для каждого подключения (обычного и реплицирующего), в которой показывается информация об использовании аутентификации и шифровании GSSAPI для данного подключения. Подробности в `pg_stat_gssapi` .
`pg_stat_progress_analyze`	По одной строке с текущим состоянием для каждого обслуживающего процесса (включая рабочие процессы автоочистки), в котором работает `ANALYZE`. См. Подраздел 26.4.1.
`pg_stat_progress_create_index`	По одной строке с текущим состоянием для каждого обслуживающего процесса, в котором выполняется `CREATE INDEX` или `REINDEX`. См. Подраздел 26.4.4.
`pg_stat_progress_vacuum`	По одной строке с текущим состоянием для каждого обслуживающего процесса (включая рабочие процессы автоочистки), в котором работает `VACUUM`. См. Подраздел 26.4.5.
`pg_stat_progress_cluster`	По одной строке с текущим состоянием для каждого обслуживающего процесса, в котором выполняется `CLUSTER` или `VACUUM FULL`. См. Подраздел 26.4.2.

Имя представления	**Описание**
`pg_stat_progress_basebackup`	По одной строке с текущим состоянием для каждого передающего WAL процесса, транслирующего базовую копию. См. Подраздел 26.4.6.
`pg_stat_progress_copy`	По одной строке с текущим состоянием для каждого обслуживающего процесса, в котором выполняется `COPY`. См. Подраздел 26.4.3.

Таблица 26.2. Представления собранной статистики

Имя представления	**Описание**
`pg_stat_archiver`	Только одна строка со статистикой работы процесса архивации WAL. Подробности в `pg_stat_archiver`.
`pg_stat_bgwriter`	Только одна строка со статистикой работы фонового процесса записи. Подробности в `pg_stat_bgwriter`.
`pg_stat_checkpointer`	Только одна строка со статистикой работы процесса контрольных точек. Подробности в `pg_stat_checkpointer`.
`pg_stat_database`	Одна строка для каждой базы данных со статистикой на уровне базы. Подробности в `pg_stat_database`.
`pg_stat_database_conflicts`	По одной строке на каждую базу данных со статистикой по отменам запросов, выполненным вследствие конфликта с процессами восстановления на ведомых серверах. Подробности в `pg_stat_database_conflicts`.
`pg_stat_io`	Одна строка для каждой комбинации типа обслуживающего процесса, контекста и целевого объекта, содержащая статистику ввода-вывода для всего кластера. За подробностями обратитесь к `pg_stat_io`.
`pg_stat_replication_slots`	По одной строке со статистикой по использованию слота репликации для каждого такого слота. Подробности в `pg_stat_replication_slots`.
`pg_stat_slru`	Одна строка со статистикой работы для каждого SLRU-кеша. Подробности в `pg_stat_slru`.
`pg_stat_subscription_stats`	Одна строка для каждой подписки, показывающая статистику ошибок. Подробности в `pg_stat_subscription_stats`.
`pg_stat_wal`	Только одна строка со статистикой работы WAL. Подробности в `pg_stat_wal`.
`pg_stat_all_tables`	По одной строке на каждую таблицу в текущей базе данных со статистикой по обращениям к этой таблице. Подробности в `pg_stat_all_tables`.
`pg_stat_sys_tables`	Аналогично `pg_stat_all_tables`, за исключением того, что отображаются только системные таблицы.

Имя представления	Описание
`pg_stat_user_tables`	Аналогично `pg_stat_all_tables`, за исключением того, что отображаются только пользовательские таблицы.
`pg_stat_xact_all_tables`	Подобно `pg_stat_all_tables`, но подсчитывает действия, выполненные в текущей транзакции к настоящему моменту (которые ещё *не* вошли в `pg_stat_all_tables` и связанные представления). Столбцы для числа живых и мёртвых строк, а также количества операций очистки и сбора статистики, в этом представлении отсутствуют.
`pg_stat_xact_sys_tables`	Аналогично `pg_stat_xact_all_tables`, за исключением того, что отображаются только системные таблицы.
`pg_stat_xact_user_tables`	Аналогично `pg_stat_xact_all_tables`, за исключением того, что отображаются только пользовательские таблицы.
`pg_stat_all_indexes`	По одной строке для каждого индекса в текущей базе данных со статистикой по обращениям к этому индексу. Подробности в `pg_stat_all_indexes`.
`pg_stat_sys_indexes`	Аналогично `pg_stat_all_indexes`, за исключением того, что показываются только индексы по системным таблицам.
`pg_stat_user_indexes`	Аналогично `pg_stat_all_indexes`, за исключением того, что показываются только индексы по пользовательским таблицам.
`pg_stat_user_functions`	По одной строке для каждой отслеживаемой функции со статистикой по выполнениям этой функции. Подробности в `pg_stat_user_functions`.
`pg_stat_xact_user_functions`	Аналогично `pg_stat_user_functions`, однако подсчитываются только вызовы функций, выполненные в текущей транзакции (которые ещё *не* были включены в `pg_stat_user_functions`).
`pg_statio_all_tables`	По одной строке для каждой таблицы в текущей базе данных со статистикой по операциям ввода-вывода с этой таблицей. Подробности в `pg_statio_all_tables`.
`pg_statio_sys_tables`	Аналогично `pg_statio_all_tables`, за исключением того, что показываются только системные таблицы.
`pg_statio_user_tables`	Аналогично `pg_statio_all_tables`, за исключением того, что показываются только пользовательские таблицы.
`pg_statio_all_indexes`	По одной строке для каждого индекса в текущей базе данных со статистикой по операциям ввода-вывода для этого индекса. Подробности в `pg_statio_all_indexes`.

Имя представления	**Описание**
`pg_statio_sys_indexes`	Аналогично `pg_statio_all_indexes`, за исключением того, что показываются только индексы по системным таблицам.
`pg_statio_user_indexes`	Аналогично `pg_statio_all_indexes`, за исключением того, что показываются только индексы по пользовательским таблицам.
`pg_statio_all_sequences`	По одной строке для каждой последовательности в текущей базе данных со статистикой по операциям ввода-вывода с этой последовательностью. Подробности в `pg_statio_all_sequences`.
`pg_statio_sys_sequences`	Аналогично `pg_statio_all_sequences`, за исключением того, что показываются только системные последовательности. (В настоящее время системных последовательностей нет, поэтому это представление всегда пусто.)
`pg_statio_user_sequences`	Аналогично `pg_statio_all_sequences`, за исключением того, что показываются только пользовательские последовательности.

Статистика по отдельным индексам особенно полезна для определения того, какие индексы используются и насколько они эффективны.

Набор представлений `pg_statio_` и `pg_stat_io` полезны для определения эффективности кеша буфера. Их можно использовать для расчёта процента попадания в кеш. Обратите внимание: в статистике ввода-вывода Postgres Pro фиксируется большинство случаев вызова ядра для выполнения ввода-вывода, но не разграничиваются случаи, когда обращение к данным, которые необходимо получить с диска или данным, которые уже находятся в кеше страниц ядра. Пользователям рекомендуется использовать статистические представления Postgres Pro в сочетании с утилитами операционной системы для получения более полной картины производительности ввода-вывода в базе данных.

26.2.3. `pg_stat_activity`

В представлении `pg_stat_activity` для каждого серверного процесса будет присутствовать по одной строке с информацией, относящейся к текущей деятельности этого процесса.

Таблица 26.3. Представление `pg_stat_activity`

Тип столбца **Описание**
`datid oid` OID базы данных, к которой подключён этот серверный процесс
`datname name` Имя базы данных, к которой подключён этот серверный процесс
`pid integer` Идентификатор процесса этого серверного процесса
`leader_pid integer` Идентификатор ведущего процесса группы параллельных процессов, если этот процесс является исполнителем параллельного запроса, или идентификатор ведущего процесса применения изменений, если этот процесс является исполнителем параллельного применения. `NULL` указывает, что этот процесс является ведущим процессом в группе параллельных процессов или ведущим процессов применения изменений, или не участвует в каких-либо параллельных операциях.

Тип столбца	**Описание**
`usesysid` `oid`	OID пользователя, подключённого к этому серверному процессу
`usename` `name`	Имя пользователя, подключённого к этому серверному процессу
`application_name` `text`	Название приложения, подключённого к этому серверному процессу
`client_addr` `inet`	IP-адрес клиента, подключённого к этому серверному процессу. Значение null в этом поле означает, что клиент подключён через сокет Unix на стороне сервера или что это внутренний процесс, например, автоочистка.
`client_hostname` `text`	Имя компьютера для подключённого клиента, получаемое в результате обратного поиска в DNS по `client_addr` . Это поле будет отлично от null только в случае соединений по IP и только при включённом режиме log_hostname.
`client_port` `integer`	Номер TCP-порта, который используется клиентом для соединения с этим обслуживающим процессом, или `-1`, если используется сокет Unix. Если поле содержит NULL, это означает, что это внутренний серверный процесс.
`backend_start` `timestamp with time zone`	Время запуска процесса. Для процессов, обслуживающих клиентов, это время подключения клиента к серверу.
`xact_start` `timestamp with time zone`	Время начала текущей транзакции в этом процессе или null при отсутствии активной транзакции. Если текущий запрос был первым в своей транзакции, то значение в этом столбце совпадает со значением столбца `query_start` .
`query_start` `timestamp with time zone`	Время начала выполнения активного в данный момент запроса, или, если `state` не `active`, то время начала выполнения последнего запроса
`state_change` `timestamp with time zone`	Время последнего изменения состояния (поля `state`)
`wait_event_type` `text`	Тип события, которого ждёт обслуживающий процесс, если это ожидание имеет место; в противном случае — NULL. См. Таблицу 26.4.
`wait_event` `text`	Имя ожидаемого события, если обслуживающий процесс находится в состоянии ожидания, а в противном случае — NULL. См. также Таблица 26.5 – Таблица 26.13.
`state` `text`	Общее текущее состояние этого серверного процесса. Возможные значения: • `active`: серверный процесс выполняет запрос. • `idle`: серверный процесс ожидает новой команды от клиента. • `idle in transaction`: серверный процесс находится внутри транзакции, но в настоящее время не выполняет никакой запрос. • `idle in transaction (aborted)` : Это состояние подобно `idle in transaction`, за исключением того, что один из операторов в транзакции вызывал ошибку. • `fastpath function call`: серверный процесс выполняет fast-path функцию. • `disabled`: Это состояние отображается для серверных процессов, у которых параметр track_activities отключён.

Тип столбца	**Описание**
`backend_xid xid`	Идентификатор транзакции верхнего уровня данного серверного процесса, если есть; см. Раздел 64.1.
`backend_xmin xid`	Текущая граница `xmin` для серверного процесса.
`query_id bigint`	Идентификатор последнего запроса этого серверного процесса. Если `state` имеет значение `active`, то в этом поле отображается идентификатор запроса, выполняемого в данный момент. Если процесс находится в любом другом состоянии, то в этом поле отображается идентификатор последнего выполненного запроса. По умолчанию идентификаторы запросов не вычисляются, поэтому данное поле будет иметь значение NULL, если не включён параметр compute_query_id или если не загружен сторонний модуль, вычисляющий идентификаторы запросов.
`query text`	Текст последнего запроса этого серверного процесса. Если `state` имеет значение `active`, то в этом поле отображается запрос, который выполняется в настоящий момент. Если процесс находится в любом другом состоянии, то в этом поле отображается последний выполненный запрос. По умолчанию текст запроса обрезается до 1024 байт; это число определяется параметром track_activity_query_size.
`backend_type text`	Тип текущего серверного процесса. Возможные варианты: `autovacuum launcher`, `autovacuum worker`, `logical replication launcher`, `logical replication worker`, `parallel worker`, `background writer`, `client backend`, `checkpointer`, `archiver`, `standalone backend`, `startup`, `walreceiver`, `walsender`, `walwriter` и `walsummarizer`. Кроме того, фоновые рабочие процессы, регистрируемые расширениями, могут иметь дополнительные типы.

Примечание

Значения в столбцах `wait_event` и `state` не зависят друг от друга. Если обслуживающий процесс находится в состоянии `active` (активен), он может ожидать какое-то событие, или не ожидать никакое. Если состояние `active` и поле `wait_event` не содержит NULL, это означает, что запрос выполняется, но заблокирован чем-то в системе.

Таблица 26.4. Типы событий ожидания

Тип события ожидания	Описание
`Activity`	Серверный процесс простаивает. Это состояние показывает, что процесс ожидает активности в основном цикле обработки. В `wait_event` обозначается конкретное место ожидания; см. Таблицу 26.5.
`BufferPin`	Серверный процесс ожидает исключительного доступа к буферу данных. Ожидание закрепления буфера может растягиваться, если другой процесс удерживает открытый курсор, который до этого читал данные из целевого буфера. См. Таблицу 26.6.
`Client`	Серверный процесс ожидает в сокете некоторую активность пользовательского приложения. То есть сервер ждёт, что произойдёт ка-

Тип события ожидания	Описание
	кое-то событие, не зависящее от его внутренних процессов. В `wait_event` обозначается конкретное место ожидания; см. Таблицу 26.7.
`Extension`	Серверный процесс ожидает условия, возникающего в модуле расширения. См. Таблицу 26.8.
`InjectionPoint`	Серверный процесс ожидает, пока точка внедрения достигнет заданного в тесте результата. За подробностями обратитесь к Подразделу 36.10.13. Этот тип не имеет предопределённых точек ожидания.
`IO`	Серверный процесс ожидает завершения операции ввода-вывода. В `wait_event` обозначается конкретное место ожидания; см. Таблицу 26.9.
`IPC`	Серверный процесс ожидает взаимодействия с другим процессом. В `wait_event` обозначается конкретное место ожидания; см. Таблицу 26.10.
`Lock`	Серверный процесс ожидает тяжёлую блокировку. Тяжёлые блокировки, также называемые блокировками менеджера блокировок или просто блокировками, в основном защищают объекты уровня SQL, такие как таблицы. Однако они также применяются для взаимоисключающего выполнения некоторых внутренних операций, например, расширения отношений. Тип ожидаемой блокировки показывается в `wait_event`; см. Таблицу 26.11.
`LWLock`	Серверный процесс ожидает лёгкую блокировку. В большинстве своём такие блокировки защищают определённые структуры данных в общей памяти. В `wait_event` будет содержаться имя, отражающее цель получения лёгкой блокировки. (Некоторые блокировки имеют особые имена; другие объединяются в группы блокировок с похожим предназначением.) См. Таблицу 26.12.
`Timeout`	Серверный процесс ожидает истечения определённого времени. В `wait_event` обозначается конкретное место ожидания; см. Таблицу 26.13.

Таблица 26.5. События ожидания, относящиеся к типу `Activity`

Событие ожидания `Activity`	Описание
`ArchiverMain`	Ожидание в основном цикле процесса архиватора.
`AutoVacuumMain`	Ожидание в основном цикле процесса запуска автоочистки.
`BgWriterHibernate`	Ожидание в фоновом процессе записи, переход в режим «заморозки».
`BgWriterMain`	Ожидание в основном цикле процесса фоновой записи.

Событие ожидания `Activity`	**Описание**
`CheckpointerMain`	Ожидание в основном цикле процесса контрольной точки.
`LogicalApplyMain`	Ожидание в основном цикле процесса применения логической репликации.
`LogicalLauncherMain`	Ожидание в основном цикле процесса запуска обработчиков логической репликации.
`LogicalParallelApplyMain`	Ожидание в основном цикле параллельного процесса применения логической репликации.
`RecoveryWalStream`	Ожидание поступления записей WAL в основном цикле стартового процесса во время восстановления.
`ReplicationSlotsyncMain`	Ожидание в основном цикле рабочего процесса синхронизации слотов.
`ReplicationSlotsyncShutdown`	Ожидание завершения рабочего процесса синхронизации слотов.
`SysLoggerMain`	Ожидание в основном цикле процесса системного журнала (syslogger).
`WalReceiverMain`	Ожидание в основном цикле процесса-приёмника WAL.
`WalSenderMain`	Ожидание в основном цикле процесса-передатчика WAL.
`WalSummarizerWal`	Ожидание создания большего количества сводок процессом создания сводок WAL.
`WalWriterMain`	Ожидание в основном цикле процесса, пишущего WAL.

Таблица 26.6. События ожидания, относящиеся к типу `BufferPin`

Событие ожидания `BufferPin`	**Описание**
`BufferPin`	Ожидание получения исключительного закрепления буфера.

Таблица 26.7. События ожидания, относящиеся к типу `Client`

События ожидания `Client`	**Описание**
`ClientRead`	Ожидание при чтении данных, получаемых от клиента.
`ClientWrite`	Ожидание при записи данных, передаваемых клиенту.
`GSSOpenServer`	Ожидание при чтении данных, получаемых от клиента при установлении сеанса GSSAPI.
`LibPQWalReceiverConnect`	Ожидание в приёмнике WAL установления подключения к удалённому серверу.
`LibpqwalreceiverReceive`	Ожидание в приёмнике WAL поступления данных от удалённого сервера.
`SSLOpenServer`	Ожидание SSL при попытке установления соединения.
`WaitForStandbyConfirmation`	Ожидание получения и сохранения WAL на диске резервным сервером физической репликации.

События ожидания `Client`	**Описание**
`WalSenderWaitForWAL`	Ожидание сброса WAL в процессе-передатчике WAL.
`WalSenderWriteData`	Ожидание какой-либо активности при обработке ответов от WAL-приёмника в процессе-передатчике WAL.

Таблица 26.8. События ожидания, относящиеся к типу `Extension`

Событие ожидания `Extension`	**Описание**
`Extension`	Ожидание в расширении.

Таблица 26.9. События ожидания, относящиеся к типу `IO`

Событие ожидания `IO`	**Описание**
`BaseBackupRead`	Ожидание чтения резервной копии из файла.
`BaseBackupSync`	Ожидание помещения данных, записанных процессом базового резервного копирования, в надёжное хранилище.
`BaseBackupWrite`	Ожидание записи резервной копии в файл.
`BufFileRead`	Ожидание чтения из буферизованного файла.
`BufFileTruncate`	Ожидание усечения буферизованного файла.
`BufFileWrite`	Ожидание записи в буферизованный файл.
`ControlFileRead`	Ожидание чтения из файла `pg_control` .
`ControlFileSync`	Ожидание помещения файла `pg_control` в надёжное хранилище.
`ControlFileSyncUpdate`	Ожидание переноса изменений файла `pg_control` в надёжное хранилище.
`ControlFileWrite`	Ожидание записи в файл `pg_control` .
`ControlFileWriteUpdate`	Ожидание записи для изменения файла `pg_control` .
`CopyFileRead`	Ожидание чтения во время операции копирования файла.
`CopyFileWrite`	Ожидание записи во время операции копирования файла.
`DataFileExtend`	Ожидание расширения файла данных отношения.
`DataFileFlush`	Ожидание помещения файла данных отношения в надёжное хранилище.
`DataFileImmediateSync`	Ожидание немедленной синхронизации файла данных отношения с надёжным хранилищем.
`DataFilePrefetch`	Ожидание асинхронной предвыборки из файла данных отношения.
`DataFileRead`	Ожидание чтения из файла данных отношения.
`DataFileSync`	Ожидание переноса изменений в файле данных отношения в надёжное хранилище.
`DataFileTruncate`	Ожидание усечения файла данных отношения.
`DataFileWrite`	Ожидание записи в файл данных отношения.

Событие ожидания `IO`	**Описание**
`DSMAllocate`	Ожидание выделения сегмента динамической общей памяти.
`DSMFillZeroWrite`	Ожидание заполнения нулями файла, применяемого для поддержки динамической общей памяти.
`LockFileAddToDataDirRead`	Ожидание чтения при добавлении строки в файл блокировки каталога данных.
`LockFileAddToDataDirSync`	Ожидание помещения данных в надёжное хранилище при добавлении строки в файл блокировки каталога данных.
`LockFileAddToDataDirWrite`	Ожидание записи при добавлении строки в файл блокировки каталога данных.
`LockFileCreateRead`	Ожидание чтения при создании файла блокировки каталога данных.
`LockFileCreateSync`	Ожидание помещения данных в надёжное хранилище при создании файла блокировки каталога данных.
`LockFileCreateWrite`	Ожидание записи при создании файла блокировки каталога данных.
`LockFileReCheckDataDirRead`	Ожидание чтения во время перепроверки файла блокировки каталога данных.
`LogicalRewriteCheckpointSync`	Ожидание помещения отображений логической перезаписи в надёжное хранилище во время контрольной точки.
`LogicalRewriteMappingSync`	Ожидание помещения данных отображений в надёжное хранилище в процессе логической перезаписи.
`LogicalRewriteMappingWrite`	Ожидание записи данных отображений в процессе логической перезаписи.
`LogicalRewriteSync`	Ожидание помещения отображений логической перезаписи в надёжное хранилище.
`LogicalRewriteTruncate`	Ожидание усечения файла отображений в процессе логической перезаписи.
`LogicalRewriteWrite`	Ожидание сохранения отображений логической перезаписи.
`RelationMapRead`	Ожидание чтения файла отображений отношений.
`RelationMapReplace`	Ожидание надёжной замены файла карты отношений.
`RelationMapWrite`	Ожидание записи в файл отображений отношений.
`ReorderBufferRead`	Ожидание чтения при работе с буфером переупорядочивания.
`ReorderBufferWrite`	Ожидание записи при работе с буфером переупорядочивания.
`ReorderLogicalMappingRead`	Ожидание чтения логического отображения при работе с буфером переупорядочивания.

Событие ожидания `IO`	**Описание**
`ReplicationSlotRead`	Ожидание чтения из управляющего файла слота репликации.
`ReplicationSlotRestoreSync`	Ожидание помещения в надёжное хранилище управляющего файла слота репликации при восстановлении его в памяти.
`ReplicationSlotSync`	Ожидание помещения в надёжное хранилище управляющего файла слота репликации.
`ReplicationSlotWrite`	Ожидание записи в управляющий файл слота репликации.
`SlruFlushSync`	Ожидание помещения данных SLRU в надёжное хранилище во время контрольной точки или отключения базы данных.
`SlruRead`	Ожидание чтения страницы SLRU.
`SlruSync`	Ожидание помещения данных SLRU в надёжное хранилище после записи страницы.
`SlruWrite`	Ожидание записи страницы SLRU.
`SnapbuildRead`	Ожидание чтения сериализованного исторического снимка каталога БД.
`SnapbuildSync`	Ожидание помещения сериализованного исторического снимка каталога БД в надёжное хранилище.
`SnapbuildWrite`	Ожидание записи сериализованного исторического снимка каталога БД.
`TimelineHistoryFileSync`	Ожидание помещения в надёжное хранилище файла истории линии времени, полученного через потоковую репликацию.
`TimelineHistoryFileWrite`	Ожидание записи файла истории линии времени, полученного через потоковую репликацию.
`TimelineHistoryRead`	Ожидание чтения файла истории линии времени.
`TimelineHistorySync`	Ожидание помещения в надёжное хранилище только что созданного файла истории линии времени.
`TimelineHistoryWrite`	Ожидание записи только что созданного файла истории линии времени.
`TwophaseFileRead`	Ожидание чтения файла двухфазного состояния.
`TwophaseFileSync`	Ожидание помещения файла двухфазного состояния в надёжное хранилище.
`TwophaseFileWrite`	Ожидание записи файла двухфазного состояния.
`VersionFileSync`	Ожидание попадания файла версии в надёжное хранилище при создании базы данных.
`VersionFileWrite`	Ожидание записи файла версии при создании базы данных.
`WALSenderTimelineHistoryRead`	Ожидание чтения из файла истории линии времени при обработки процессом walsender команды timeline.

Событие ожидания `IO`	**Описание**
`WALBootstrapSync`	Ожидание помещения WAL в надёжное хранилище в процессе начальной загрузки.
`WALBootstrapWrite`	Ожидание записи страницы WAL в процессе начальной загрузки.
`WALCopyRead`	Ожидание чтения при создании нового сегмента WAL путём копирования существующего.
`WALCopySync`	Ожидание помещения в надёжное хранилище нового сегмента WAL, созданного путём копирования существующего.
`WALCopyWrite`	Ожидание записи при создании нового сегмента WAL путём копирования существующего.
`WALInitSync`	Ожидание помещения в надёжное хранилище нового инициализированного файла WAL.
`WALInitWrite`	Ожидание записи при инициализации нового файла WAL.
`WalRead`	Ожидание чтения из файла WAL.
`WalSummaryRead`	Ожидание чтения из файла сводок WAL.
`WalSummaryWrite`	Ожидание записи в файл сводок WAL.
`WalSync`	Ожидание помещения файла WAL в надёжное хранилище.
`WALSyncMethodAssign`	Ожидание помещения данных в надёжное хранилище при смене метода синхронизации WAL.
`WalWrite`	Ожидание записи в файл WAL.

Таблица 26.10. События ожидания, относящиеся к типу `Ipc`

События ожидания `IPC`	**Описание**
`AppendReady`	Ожидание готовности узлов вложенных планов узла `Append`.
`ArchiveCleanupCommand`	Ожидание завершения archive_cleanup_command.
`ArchiveCommand`	Ожидание завершения archive_command.
`BackendTermination`	Ожидание завершения другого серверного процесса.
`BackupWaitWalArchive`	Ожидание файлов WAL, необходимых для успешного завершения архивации.
`BgWorkerShutdown`	Ожидание завершения фонового рабочего процесса.
`BgworkerStartup`	Ожидание запуска фонового рабочего процесса.
`BtreePage`	Ожидание доступности номера страницы, необходимого для продолжения параллельного сканирования B-дерева.
`BufferIO`	Ожидание завершения буферного ввода-вывода.
`CheckpointDelayComplete`	Ожидание обслуживающего процесса, который блокирует завершение контрольной точки.

События ожидания `IPC`	**Описание**
`CheckpointDelayStart`	Ожидание обслуживающего процесса, который блокирует начало контрольной точки.
`CheckpointDone`	Ожидание завершения контрольной точки.
`CheckpointStart`	Ожидание начала контрольной точки.
`ExecuteGather`	Ожидание активности дочернего процесса при выполнении узла плана `Gather`.
`HashBatchAllocate`	Ожидание выделения хеш-таблицы выбранным участником параллельного хеширования.
`HashBatchElect`	Ожидание выделения хеш-таблицы выбранным участником параллельного хеширования.
`HashBatchLoad`	Ожидание завершения загрузки хеш-таблицы другими участниками параллельного хеширования.
`HashBuildAllocate`	Ожидание выделения начальной хеш-таблицы выбранным участником параллельного хеширования.
`HashBuildElect`	Ожидание в процессе выбора участника параллельного хеширования для выделения начальной хеш-таблицы.
`HashBuildHashInner`	Ожидание завершения хеширования внутреннего отношения другими участниками параллельного хеширования.
`HashBuildHashOuter`	Ожидание завершения хеширования внешнего отношения другими участниками параллельного хеширования.
`HashGrowBatchesDecide`	Ожидание в процессе выбора участника параллельного хеширования для принятия решения о предстоящем добавлении пакетов.
`HashGrowBatchesElect`	Ожидание в процессе выбора участника параллельного хеширования для выделения дополнительных пакетов.
`HashGrowBatchesFinish`	Ожидание решения о предстоящем добавлении пакетов участником параллельного хеширования.
`HashGrowBatchesReallocate`	Ожидание выделения дополнительных пакетов выбранным участником параллельного хеширования.
`HashGrowBatchesRepartition`	Ожидание завершения пересекционирования другими участниками параллельного хеширования.
`HashGrowBucketsElect`	Ожидание в процессе выбора участника параллельного хеширования для выделения дополнительных групп.
`HashGrowBucketsReallocate`	Ожидание завершения выделения дополнительных групп выбранным участником параллельного хеширования.
`HashGrowBucketsReinsert`	Ожидание завершения добавления кортежей в новые группы другими участниками параллельного хеширования.

События ожидания `IPC`	**Описание**
`LogicalApplySendData`	Ожидание отправки данных ведущим процессом применения изменений логической репликации параллельному процессу применения изменений.
`LogicalParallelApplyStateChange`	Ожидание изменения состояния параллельного процесса применения изменений логической репликации.
`LogicalSyncData`	Ожидание от удалённого сервера, осуществляющего логическую репликацию, передачи данных для начальной синхронизации таблиц.
`LogicalSyncStateChange`	Ожидание изменения состояния удалённого сервера, осуществляющего логическую репликацию.
`MessageQueueInternal`	Ожидание подключения другого процесса к общей очереди сообщений.
`MessageQueuePutMessage`	Ожидание записи сообщения протокола в общую очередь сообщений.
`MessageQueueReceive`	Ожидание получения байтов из общей очереди сообщений.
`MessageQueueSend`	Ожидание передачи байтов в общую очередь сообщений.
`MultixactCreation`	Ожидание завершения создания мультитранзакции.
`ParallelBitmapScan`	Ожидание инициализации параллельного сканирования по битовой карте.
`ParallelCreateIndexScan`	Ожидание завершения сканирования кучи параллельными исполнителями `CREATE INDEX`.
`ParallelFinish`	Ожидание завершения вычислений параллельными рабочими процессами.
`ProcArrayGroupUpdate`	Ожидание очистки ведущим группы идентификатора транзакции в конце транзакции.
`ProcSignalBarrier`	Ожидание обработки события барьера всеми обслуживающими процессами.
`Promote`	Ожидание повышения ведомого.
`RecoveryConflictSnapshot`	Ожидание разрешения конфликтов восстановления для осуществления очистки.
`RecoveryConflictTablespace`	Ожидание разрешения конфликтов восстановления для удаления табличного пространства.
`RecoveryEndCommand`	Ожидание завершения recovery_end_command.
`RecoveryPause`	Ожидание возобновления восстановления.
`ReplicationOriginDrop`	Ожидание перехода источника репликации в неактивное состояние, что позволит затем удалить его.
`ReplicationSlotDrop`	Ожидание перехода слота репликации в неактивное состояние, что позволит затем удалить его.
`RestoreCommand`	Ожидание завершения restore_command.

События ожидания `IPC`	**Описание**
`SafeSnapshot`	Ожидание получения безопасного снимка для транзакции `READ ONLY DEFERRABLE`.
`SyncRep`	Ожидание подтверждения от удалённого сервера при синхронной репликации.
`WalReceiverExit`	Ожидание завершения процесса-приёмника WAL.
`WalReceiverWaitStart`	Ожидание от стартового процесса передачи начальных данных для потоковой репликации.
`WalSummaryReady`	Ожидание создания новой сводки WAL.
`XactGroupUpdate`	Ожидание изменения состояния завершённой транзакции ведущим процессом группы.

Таблица 26.11. События ожидания, относящиеся к типу `Lock`

Событие ожидания `Lock`	**Описание**
`advisory`	Ожидание при запросе рекомендательной пользовательской блокировки.
`applytransaction`	Ожидание получения блокировки для удалённой транзакции, применяемой подписчиком логической репликации.
`extend`	Ожидание при расширении отношения.
`frozenid`	Ожидание изменения `pg_database .datfrozenxid` и `pg_database .datminmxid`.
`object`	Ожидание при запросе блокировки для нереляционного объекта БД.
`page`	Ожидание при запросе блокировки для страницы отношения.
`relation`	Ожидание при запросе блокировки для отношения.
`spectoken`	Ожидание при запросе блокировки спекулятивного добавления.
`transactionid`	Ожидание завершения транзакции.
`tuple`	Ожидание при запросе блокировки для кортежа.
`userlock`	Ожидание при запросе пользовательской блокировки.
`virtualxid`	Ожидание при запросе блокировки виртуального ID транзакции; см. Раздел 64.1.

Таблица 26.12. События ожидания, относящиеся к типу `LWLock`

Событие ожидания `LWLock`	**Описание**
`AddinShmemInit`	Ожидание при распределении блоков общей памяти для расширений.
`AutoFile`	Ожидание при изменении файла `postgresql.auto.conf`.
`Autovacuum`	Ожидание при чтении или изменении текущего состояния рабочих процессов автоочистки.

Событие ожидания `LWLock`	**Описание**
`AutovacuumSchedule`	Ожидание при подтверждении, что таблица, выбранная для автоочистки, всё ещё нуждается в очистке.
`BackgroundWorker`	Ожидание при чтении или изменении состояния фонового рабочего процесса.
`BtreeVacuum`	Ожидание при чтении или изменении информации, связанной с очисткой, для индекса-B-дерева.
`BufferContent`	Ожидание при обращении к странице данных в памяти.
`BufferMapping`	Ожидание при связывании блока данных с буфером в пуле буферов.
`CheckpointerComm`	Ожидание при управлении запросами fsync.
`CommitTs`	Ожидание при чтении или изменении последнего значения, заданного в качестве времени фиксирования транзакции.
`CommitTsBuffer`	Ожидание ввода-вывода с SLRU-буфером данных о времени фиксирования транзакций.
`CommitTsSLRU`	Ожидание при обращении к SLRU-кешу данных о времени фиксирования транзакций.
`ControlFile`	Ожидание при чтении или изменении файла `pg_control` либо при создании нового файла WAL.
`DSMRegistry`	Ожидание при чтении или изменении реестра динамической общей памяти.
`DSMRegistryDSA`	Ожидание при обращении к механизму управления памятью реестра динамической общей памяти.
`DSMRegistryHash`	Ожидание при обращении к общей хеш-таблице реестра динамической общей памяти.
`DynamicSharedMemoryControl`	Ожидание при чтении или изменении информации о выделении динамической общей памяти.
`InjectionPoint`	Ожидание при чтении или изменении информации о точках внедрения.
`LockFastPath`	Ожидание при чтении или изменении информации процесса о блокировках по быстрому пути.
`LockManager`	Ожидание при чтении или изменении информации о «тяжёлых» блокировках.
`LogicalRepLauncherDSA`	Ожидание выделения области динамической общей памяти для запуска логической репликации.
`LogicalRepLauncherHash`	Ожидание доступа к общей хеш-таблице процесса запуска логической репликации.
`LogicalRepWorker`	Ожидание при чтении или изменении состояния рабочих процессов логической репликации.

Событие ожидания `LWLock`	**Описание**
`MultiXactGen`	Ожидание при чтении или изменении общего состояния мультитранзакций.
`MultiXactMemberBuffer`	Ожидание ввода-вывода с SLRU-буфером данных о членах мультитранзакций.
`MultiXactMemberSLRU`	Ожидание при обращении к SLRU-кешу данных о членах мультитранзакций.
`MultiXactOffsetBuffer`	Ожидание ввода-вывода с SLRU-буфером данных о смещениях мультитранзакций.
`MultiXactOffsetSLRU`	Ожидание при обращении к SLRU-кешу данных о смещениях мультитранзакций.
`MultiXactTruncation`	Ожидание при чтении или очистке информации мультитранзакций.
`NotifyBuffer`	Ожидание ввода-вывода с SLRU-буфером сообщений `NOTIFY`.
`NotifyQueue`	Ожидание при чтении или изменении сообщений `NOTIFY`.
`NotifyQueueTail`	Ожидание при изменении границы массива сообщений `NOTIFY`.
`NotifySLRU`	Ожидание при обращении к SLRU-кешу сообщений `NOTIFY`.
`OidGen`	Ожидание при выделении нового OID.
`ParallelAppend`	Ожидание выбора следующего подплана в процессе выполнения узла параллельного добавления (Parallel Append).
`ParallelHashJoin`	Ожидание синхронизации рабочих процессов в процессе выполнения узла плана Parallel Hash Join.
`ParallelQueryDSA`	Ожидание выделения динамической общей памяти для параллельного запроса.
`ParallelVacuumDSA`	Ожидание выделения динамической общей памяти для параллельной очистки.
`PerSessionDSA`	Ожидание выделения динамической общей памяти для параллельного запроса.
`PerSessionRecordType`	Ожидание при обращении к информации параллельного запроса о составных типах.
`PerSessionRecordTypmod`	Ожидание при обращении к информации параллельного запроса о модификаторах типов, относящихся к типам анонимных записей.
`PerXactPredicateList`	Ожидание при обращении к списку предикатных блокировок, удерживаемых текущей сериализуемой транзакцией, во время параллельного запроса.
`PgStatsData`	Ожидание доступа к данным статистики в общей памяти.
`PgStatsDSA`	Ожидание выделения области динамической общей памяти для статистики.
`PgStatsHash`	Ожидание доступа к хеш-таблице статистики в общей памяти.

Событие ожидания `LWLock`	**Описание**
`PredicateLockManager`	Ожидание при обращении к информации о предикатных блокировках, используемой сериализуемыми транзакциями.
`ProcArray`	Ожидание при обращении к общим структурам данных в рамках процесса (например, при получении снимка или чтении идентификатора транзакции в сеансе).
`RelationMapping`	Ожидание при чтении или изменении файла `pg_filenode.map`, в котором отслеживаются назначения файловых узлов для некоторых системных каталогов.
`RelCacheInit`	Ожидание при чтении или изменении файла инициализации кеша отношения (`pg_internal.init`).
`ReplicationOrigin`	Ожидание при создании, удалении или использовании источника репликации.
`ReplicationOriginState`	Ожидание при чтении или изменении состояния одного источника репликации.
`ReplicationSlotAllocation`	Ожидание при выделении или освобождении слота репликации.
`ReplicationSlotControl`	Ожидание при чтении или изменении состояния слота репликации.
`ReplicationSlotIO`	Ожидание при вводе-выводе со слотом репликации.
`SerialBuffer`	Ожидание ввода-вывода с SLRU-буфером данных о конфликтах сериализуемых транзакций.
`SerialControl`	Ожидание при чтении или изменении общего состояния `pg_serial`.
`SerializableFinishedList`	Ожидание при обращении к списку завершённых сериализуемых транзакций.
`SerializablePredicateList`	Ожидание при обращении к списку предикатных блокировок, удерживаемых сериализуемыми транзакциями.
`SerializableXactHash`	Ожидание при чтении или изменении информации о сериализуемых транзакциях.
`SerialSLRU`	Ожидание при обращении к SLRU-кешу данных о конфликтах сериализуемых транзакций.
`SharedTidBitmap`	Ожидание при обращении к разделяемой битовой карте TID в процессе параллельного сканирования индекса по битовой карте.
`SharedTupleStore`	Ожидание при обращении к разделяемому хранилищу кортежей во время параллельного запроса.
`ShmemIndex`	Ожидание при поиске или выделении области в разделяемой памяти.
`SInvalRead`	Ожидание при получении сообщений из общей очереди сообщений аннулирования.
`SInvalWrite`	Ожидание при добавлении в общую очередь сообщения аннулирования каталога.

Событие ожидания `LWLock`	**Описание**
`SubtransBuffer`	Ожидание ввода-вывода с SLRU-буфером данных о подтранзакциях.
`SubtransSLRU`	Ожидание при обращении к SLRU-кешу данных подтранзакций.
`SyncRep`	Ожидание при чтении или изменении информации о состоянии синхронной репликации.
`SyncScan`	Ожидание при выборе начального положения для синхронизированного сканирования таблицы.
`TablespaceCreate`	Ожидание при создании или удалении табличного пространства.
`TwoPhaseState`	Ожидание при чтении или изменении состояния подготовленных транзакций.
`WaitEventCustom`	Ожидание при чтении или изменении информации о пользовательских событиях ожидания.
`WALBufMapping`	Ожидание при замене страницы в буферах WAL.
`WALInsert`	Ожидание при добавлении записей WAL в буфер в памяти.
`WALSummarizer`	Ожидание при чтении или изменении состояния процесса создания сводок WAL.
`WALWrite`	Ожидание при записи буферов WAL на диск.
`WrapLimitsVacuum`	Ожидание при изменении лимитов идентификаторов транзакций и мультитранзакций.
`XactBuffer`	Ожидание ввода-вывода с SLRU-буфером данных о состоянии транзакций.
`XactSLRU`	Ожидание при обращении к SLRU-кешу данных о состоянии транзакций.
`XactTruncation`	Ожидание выполнения функции `pg_xact_status` или обновления самого старого видимого в ней идентификатора транзакции.
`XidGen`	Ожидание при выделении нового идентификатора транзакции.

Таблица 26.13. События ожидания, относящиеся к типу `Timeout`

Событие ожидания `Timeout`	**Описание**
`BaseBackupThrottle`	Ожидание в процессе базового резервного копирования из-за ограничения активности.
`CheckpointWriteDelay`	Ожидание между операциями записи при выполнении контрольной точки.
`PgSleep`	Ожидание в результате вызова `pg_sleep` или родственной функции.
`RecoveryApplyDelay`	Ожидание применения WAL при восстановлении, вызванное установленной задержкой.
`RecoveryRetrieveRetryInterval`	Ожидание в процессе восстановления, когда нужные сегменты WAL нельзя получить из какого-либо источника (каталога `pg_wal`, архива или потока).

Событие ожидания `Timeout`	**Описание**
`RegisterSyncRequest`	Ожидание при передаче запросов синхронизации процессу контрольной точки из-за переполнения очереди запросов.
`SpinDelay`	Ожидание получения конкурирующей циклической блокировки.
`VacuumDelay`	Ожидание, вызванное задержкой очистки по критерию стоимости.
`VacuumTruncate`	Ожидание исключительной блокировки для отсечения пустых страниц в конце очищаемой таблицы.
`WalSummarizerError`	Ожидание после ошибки процесса создания сводок WAL.

Следующие команды показывают, как можно просмотреть события ожидания:

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE wait_event is NOT
 NULL;
 pid  | wait_event_type | wait_event
------+-----------------+------------
 2540 | Lock            | relation
 6644 | LWLock          | ProcArray
(2 rows)
```

```
SELECT a.pid, a.wait_event, w.description
  FROM pg_stat_activity a JOIN
       pg_wait_events w ON (a.wait_event_type = w.type AND
                            a.wait_event = w.name)
  WHERE a.wait_event is NOT NULL and a.state = 'active';
-[ RECORD 1 ]------------------------------------------------------------------
pid         | 686674
wait_event  | WALInitSync
description | Waiting for a newly initialized WAL file to reach durable storage
```

Примечание

Приложения могут добавлять дополнительные события `Extension`, `InjectionPoint` и `LWLock` в списки, показанные в Таблице 26.8 и Таблице 26.12. В некоторых случаях имя `LWLock`, назначенное расширением, может быть не видно во всех серверных процессах, поэтому это событие ожидания может отображаться с именем «`extension`», а не тем, что было назначено.

26.2.4. `pg_stat_replication`

Представление `pg_stat_replication` для каждого процесса-передатчика WAL будет содержать по одной строке со статистикой о репликации на ведомый сервер, к которому подключён этот процесс. В представлении перечисляются только ведомые серверы, подключённые напрямую; информация о ведомых серверах, подключённых опосредованно, не представлена.

Таблица 26.14. Представление `pg_stat_replication`

Тип столбца **Описание**
`pid` `integer` Идентификатор процесса-передатчика WAL
`usesysid` `oid` OID пользователя, подключённого к этому процессу-передатчику WAL

Тип столбца **Описание**
`usename` `name` Имя пользователя, подключённого к этому процессу-передатчику WAL
`application_name` `text` Имя приложения, которое подключено к этому процессу-передатчику WAL
`client_addr` `inet` IP-адрес клиента, подключённого к этому процессу-передатчику WAL. Значение null в этом поле говорит о том, что клиент подсоединён через сокет Unix на сервере.
`client_hostname` `text` Имя компьютера для подключённого клиента, получаемое в результате обратного поиска в DNS по `client_addr` . Это поле будет отлично от null только в случае соединений по IP и только при включённом режиме log_hostname.
`client_port` `integer` Номер TCP-порта, который используется клиентом для взаимодействия с процессом-передатчиком WAL, или `-1`, если используется сокет Unix
`backend_start` `timestamp with time zone` Время запуска процесса, т. е. время подключения клиента к этому процессу-передатчику WAL
`backend_xmin` `xid` Значение `xmin`, полученное от ведомого сервера при включённом hot_standby_feedback.
`state` `text` Текущее состояние процесса-передатчика WAL. Возможные значения: • `startup`: Передатчик WAL запускается. • `catchup`: Подключённый к этому передатчику WAL ведомый сервер догоняет ведущий. • `streaming`: Передатчик WAL транслирует изменения после того, как подключённый к нему ведомый сервер нагнал ведущий. • `backup`: Передатчик WAL передаёт резервную копию. • `stopping`: Передатчик WAL останавливается.
`sent_lsn` `pg_lsn` Последняя позиция в журнале предзаписи, переданная через это соединение
`write_lsn` `pg_lsn` Последняя позиция в журнале предзаписи, записанная на диск этим ведомым сервером
`flush_lsn` `pg_lsn` Последняя позиция в журнале предзаписи, сброшенная на диск этим ведомым сервером
`replay_lsn` `pg_lsn` Последняя позиция в журнале предзаписи, воспроизведённая в базе данных этим ведомым сервером
`write_lag` `interval` Время, прошедшее с момента локального сброса последних данных WAL до получения уведомления о том, что этот ведомый сервер записал их (но ещё не сбросил на диск и не применил). Это позволяет оценить задержку, возникающую при фиксации транзакции, когда в `synchronous_commit` выбран уровень `remote_write` , если данный сервер будет настроен как синхронный ведомый.
`flush_lag` `interval` Время, прошедшее с момента локального сброса последних данных WAL до получения уведомления о том, что этот ведомый сервер записал и сбросил их на диск (но ещё не применил). Это позволяет оценить задержку, возникающую при фиксации транзакции,

Тип столбца **Описание**
когда в `synchronous_commit` выбран уровень `on`, если данный сервер будет настроен как синхронный ведомый.
`replay_lag interval` Время, прошедшее с момента локального сброса последних данных WAL до получения уведомления о том, что этот ведомый сервер записал, сбросил на диск и применил их. Это позволяет оценить задержку, возникающую при фиксации транзакции, когда в `synchronous_commit` выбран уровень `remote_apply` , если данный сервер будет настроен как синхронный ведомый.
`sync_priority integer` Приоритет этого ведомого сервера для выбора в качестве синхронного ведомого при синхронной репликации с учётом приоритетов. При синхронной репликации с учётом кворума не имеет значения.
`sync_state text` Состояние синхронизации этого ведомого сервера. Возможные значения: • `async`: Этот ведомый сервер является асинхронным. • `potential`: Этот ведомый сервер сейчас является асинхронным, но может стать синхронным в случае отказа одного из текущих синхронных серверов. • `sync`: Этот ведомый сервер является синхронным. • `quorum`: Этот ведомый сервер считается кандитатом на участие в кворуме.
`reply_time timestamp with time zone` Время отправки последнего ответного сообщения, полученного от ведомого сервера

Длительность задержек, показываемая в представлении `pg_stat_replication`, включает время, которое потребовалось для того, чтобы записать, сбросить на диск и воспроизвести последние записи WAL и для того, чтобы передатчик WAL узнал об этом. Эта длительность отражает задержку фиксации, которая была (или могла быть) добавлена на уровнях синхронной фиксации, если ведомый сервер был настроен как синхронный. Для асинхронного ведомого в столбце `replay_lag` показывается примерная задержка перед тем, как последние транзакции становятся видны для запросов. Если ведомый сервер нагоняет передающий и в WAL отсутствует активность, последние длительности задержек будут отображаться ещё некоторое время, а затем сменятся на NULL.

Длительность задержек автоматически определяется при физической репликации. Модули логического декодирования могут не выдавать необходимые контрольные сообщения; в их отсутствие механизм отслеживания просто выводит задержку NULL.

Примечание

Выдаваемые длительности задержек не являются предсказанием времени, которое потребуется ведомому серверу, чтобы нагнать передающий сервер с учётом текущей скорости воспроизведения. Эти показатели будут близки в процессе генерирования нового WAL, но не в то время, когда передающий сервер будет простаивать. В частности, когда ведомый сервер нагоняет ведущий, в `pg_stat_replication` показывается, сколько времени потребовалось для записи, сброса на диск и воспроизведения последних данных WAL, а не 0, как могли ожидать некоторые пользователи. Это соответствует задаче измерения задержек синхронной фиксации и видимости транзакций для недавно записанных транзакций. Чтобы меньше смущать пользователей, ожидающих видеть другую модель задержек, столбцы задержек сбрасываются в NULL после небольшой паузы в системе, которая воспроизвела все изменения и теперь простаивает. Системы мониторинга могут представлять это как отсутствующие данные, 0 или продолжать показывать последнее известное значение.

26.2.5. `pg_stat_replication_slots`

Представление `pg_stat_replication_slots` содержит одну строку со статистикой использования на каждый слот логической репликации.

Таблица 26.15. Представление `pg_stat_replication_slots`

Тип столбца Описание
`slot_name` `text` Уникальный в рамках кластера идентификатор для слота репликации
`spill_txns` `bigint` Количество транзакций, вытесненных на диск после того, как объём памяти, используемый при логическом декодировании изменений из WAL, превысил значение `logical_decoding_work_mem`. Счётчик учитывает как транзакции верхнего уровня, так и подтранзакции.
`spill_count` `bigint` Сколько раз транзакции были вытеснены на диск при декодировании изменений из WAL для этого слота. Этот счётчик увеличивается каждый раз, когда транзакция вытесняется; одна и та же транзакция может быть вытеснена несколько раз.
`spill_bytes` `bigint` Объём декодированных транзакционных данных, вытесненных на диск при выполнении декодирования изменений из WAL для этого слота. Этот и другие счётчики, связанные с вытеснением, помогают оценить объём связанного с логическим декодированием ввода-вывода и настроить параметр `logical_decoding_work_mem`.
`stream_txns` `bigint` Количество текущих транзакций, переданных в потоке в модуль вывода декодирования после того, как объём памяти, используемый для логического декодирования изменений из WAL для этого слота, превысил значение `logical_decoding_work_mem`. В потоке передаются только транзакции верхнего уровня (подтранзакции не могут передаваться в потоке отдельно), поэтому счётчик не учитывает подтранзакции.
`stream_count` `bigint` Сколько раз текущие транзакции были переданы в потоке модулю вывода при декодировании изменений из WAL для этого слота. Этот счётчик увеличивается каждый раз, когда транзакция передаётся в потоке; одна и та же транзакция может быть передана в потоке несколько раз.
`stream_bytes` `bigint` Объём транзакционных данных, декодированных для передачи текущих транзакций в потоке в модуль вывода при декодировании изменений из WAL для этого слота. Этот и другие счётчики передаваемых транзакций данного слота могут быть полезны для настройки `logical_decoding_work_mem`.
`total_txns` `bigint` Количество декодированных транзакций, отправленных в модуль вывода декодирования для этого слота. Учитываются только транзакции верхнего уровня, но не подтранзакции. Обратите внимание, что в этом количестве учитываются передаваемые в потоке и/или вытесняемые транзакции.
`total_bytes` `bigint` Объём транзакционных данных, декодированных для отправки транзакций в модуль вывода при декодировании изменений из WAL для этого слота. Обратите внимание, что в этом объёме учитываются передаваемые в потоке и/или вытесняемые данные.
`stats_reset` `timestamp with time zone` Последнее время сброса этих статистических данных

26.2.6. `pg_stat_wal_receiver`

Представление `pg_stat_wal_receiver` будет иметь только одну строку со статистикой приёмника WAL от сервера, на котором работает приёмник.

Таблица 26.16. Представление `pg_stat_wal_receiver`

Тип столбца	Описание
`pid integer`	Идентификатор процесса WAL-приёмника
`status text`	Состояние активности процесса WAL-приёмника
`receive_start_lsn pg_lsn`	Первая позиция в журнале предзаписи в момент запуска приёмника WAL
`receive_start_tli integer`	Первый номер линии времени в момент запуска приёмника WAL
`written_lsn pg_lsn`	Последняя позиция в журнале предзаписи, уже полученная и переданная для записи на диск, но ещё не сброшенная. Эту позицию не следует использовать при проверках целостности данных.
`flushed_lsn pg_lsn`	Последняя позиция в журнале предзаписи, уже полученная и сброшенная на диск; начальным значением этого поля будет первая позиция в журнале в момент запуска приёмника WAL
`received_tli integer`	Номер линии времени последней позиции в журнале предзаписи, уже полученной и сброшенной на диск; начальным значением этого поля будет номер линии времени первой позиции в момент запуска приёмника WAL
`last_msg_send_time timestamp with time zone`	Время отправки последнего сообщения, полученного от изначального передатчика WAL
`last_msg_receipt_time timestamp with time zone`	Время поступления последнего сообщения, полученного от изначального передатчика WAL
`latest_end_lsn pg_lsn`	Последняя позиция в журнале предзаписи, сообщённая изначальному передатчику WAL
`latest_end_time timestamp with time zone`	Время последней позиции в журнале предзаписи, сообщённой изначальному передатчику WAL
`slot_name text`	Имя слота репликации, используемого этим приёмником WAL
`sender_host text`	Узел, где работает сервер Postgres Pro, обслуживающий данный приёмник WAL. Может задаваться именем или IP-адресом компьютера либо путём каталога (если подключение установлено через сокет Unix). (Подключение к сокету легко распознать, потому что путь всегда абсолютный и начинается с `/`.)
`sender_port integer`	Номер порта сервера Postgres Pro, к которому подключён этот приёмник WAL.
`conninfo text`	Строка подключения, используемая этим приёмником WAL (секретные поля в ней скрыты).

26.2.7. `pg_stat_recovery_prefetch`

Представление `pg_stat_recovery_prefetch` будет содержать только одну строку. В столбцах `wal_distance`, `block_distance` и `io_depth` показываются текущие значения, а в других столбцах — накопительные счётчики, которые можно сбросить функцией `pg_stat_reset_shared`.

Таблица 26.17. Представление `pg_stat_recovery_prefetch`

Тип столбца Описание
`stats_reset timestamp with time zone` Последнее время сброса этих статистических данных
`prefetch bigint` Количество блоков, выбранных предварительно ввиду их отсутствия в пуле буферов
`hit bigint` Количество блоков, не выбранных предварительно из-за того, что они уже находились в пуле буферов
`skip_init bigint` Количество блоков, не выбранных предварительно, потому что они должны инициализироваться нулями
`skip_new bigint` Количество блоков, не выбранных предварительно, потому что они не существовали
`skip_fpw bigint` Количество блоков, не выбранных предварительно, так как в WAL были записаны образы полных страниц
`skip_rep bigint` Количество блоков, не выбранных предварительно, потому что они были предвыбраны ранее
`wal_distance int` Объём WAL, просматриваемый для предвыборки блоков (в байтах)
`block_distance int` Количество блоков в процессе предвыборки
`io_depth int` Число инициированных операций предвыборки, которые ещё не считаются завершёнными

26.2.8. `pg_stat_subscription`

Таблица 26.18. Представление `pg_stat_subscription`

Тип столбца Описание
`subid oid` OID подписки
`subname name` Имя подписки
`worker_type text` Тип рабочего процесса подписки. Возможные варианты: `apply`, `parallel apply` и `table synchronize`.
`pid integer` Идентификатор рабочего процесса, обслуживающего подписку
`leader_pid integer` Идентификатор ведущего рабочего процесса применения, если этот процесс является параллельным рабочим процессом применения; NULL, если этот процесс является ведущим процессом применения изменений или рабочим процессом синхронизации таблиц
`relid oid` OID отношения, которое синхронизирует рабочий процесс сейчас; NULL для ведущего и параллельных процессов применения изменений

Тип столбца **Описание**
`received_lsn pg_lsn` Последняя позиция в журнале предзаписи (начальное значение этого поля — 0); NULL для параллельных рабочих процессов применения
`last_msg_send_time timestamp with time zone` Время отправки последнего сообщения, полученного от изначального передатчика WAL; NULL для параллельных рабочих процессов применения изменений
`last_msg_receipt_time timestamp with time zone` Время поступления последнего сообщения, полученного от изначального передатчика WAL; NULL для параллельных рабочих процессов применения изменений
`latest_end_lsn pg_lsn` Последняя позиция в журнале предзаписи, сообщённая изначальному передатчику WAL; NULL для параллельных рабочих процессов применения изменений
`latest_end_time timestamp with time zone` Время последней позиции в журнале предзаписи, сообщённой изначальному передатчику WAL; NULL для параллельных рабочих процессов применения изменений

26.2.9. `pg_stat_subscription_stats`

Представление `pg_stat_subscription_stats` будет содержать одну строку для каждой подписки.

Таблица 26.19. Представление `pg_stat_subscription_stats`

Тип столбца **Описание**
`subid oid` OID подписки
`subname name` Имя подписки
`apply_error_count bigint` Число ошибок, возникших при применении изменений
`sync_error_count bigint` Сколько раз возникали ошибки во время начальной синхронизации таблиц
`stats_reset timestamp with time zone` Последнее время сброса этих статистических данных

26.2.10. `pg_stat_ssl`

Представление `pg_stat_ssl` содержит по одной строке для каждого обслуживающего процесса и процесса, передающего WAL, и показывает статистику использования SSL для подключений. Его можно соединить с `pg_stat_activity` или `pg_stat_replication` по столбцу `pid` и получить дополнительные сведения о подключении.

Таблица 26.20. Представление `pg_stat_ssl`

Тип столбца **Описание**
`pid integer` Идентификатор обслуживающего процесса или процесса, передающего WAL
`ssl boolean` True, если для этого подключения используется SSL
`version text`

Тип столбца **Описание**
Версия SSL либо NULL, если SSL для этого подключения не используется
`cipher text` Имя применяемого шифра SSL либо NULL, если SSL для этого подключения не используется
`bits integer` Число бит в применяемом алгоритме шифрования либо NULL, если SSL для этого подключения не используется
`client_dn text` Поле DN (Distinguished Name, Уникальное имя) из используемого клиентского сертификата либо NULL, если клиент не передал сертификат или SSL для этого подключения не используется. Это значение усекается, если оказывается длиннее `NAMEDATALEN` символов (64 символа в стандартной сборке).
`client_serial numeric` Серийный номер клиентского сертификата либо NULL, если клиент не передал сертификат или SSL для этого подключения не используется. В сочетании с именем центра сертификации, выдавшего сертификат, серийный номер однозначно идентифицирует сертификат (если только ЦС не выдаёт по ошибке одинаковые серийные номера).
`issuer_dn text` Уникальное имя (Distinguished Name, DN) центра сертификации, выдавшего клиентский сертификат, либо NULL, если клиент не передал сертификат или SSL для этого подключения не используется. Это значение усекается подобно `client_dn` .

26.2.11. `pg_stat_gssapi`

Представление `pg_stat_gssapi` содержит по одной строке для каждого обслуживающего процесса и показывает статистику использования GSSAPI для конкретного подключения. Его можно соединить с `pg_stat_activity` или `pg_stat_replication` по столбцу `pid` и получить дополнительные сведения о подключении.

Таблица 26.21. Представление `pg_stat_gssapi`

Тип столбца **Описание**
`pid integer` Идентификатор (PID) обслуживающего процесса
`gss_authenticated boolean` True, если для этого подключения используется аутентификация GSSAPI
`principal text` Принципал, для которого было аутентифицировано это подключение, либо NULL, если клиент не передал сертификат или GSSAPI для этого подключения не используется. Это значение усекается, если оказывается длиннее `NAMEDATALEN` символов (64 символа в стандартной сборке).
`encrypted boolean` True, если для этого подключения используется шифрование GSSAPI
`credentials_delegated boolean` True, если для этого подключения были делегированы учётные данные GSSAPI.

26.2.12. `pg_stat_archiver`

В представлении `pg_stat_archiver` всегда будет одна строка, содержащая данные о текущем состоянии процесса архивации в кластере.

Таблица 26.22. Представление `pg_stat_archiver`

Тип столбца **Описание**
`archived_count` `bigint` Число файлов WAL, которые были успешно архивированы
`last_archived_wal` `text` Имя последнего успешно заархивированного файла WAL
`last_archived_time` `timestamp with time zone` Время последней успешной архивации
`failed_count` `bigint` Число неудачных попыток архивации файлов WAL
`last_failed_wal` `text` Имя файла WAL, который обрабатывался в ходе последней неудавшейся архивации
`last_failed_time` `timestamp with time zone` Время последней неудавшейся архивации
`stats_reset` `timestamp with time zone` Последнее время сброса этих статистических данных

Обычно файлы WAL архивируются в порядке от старых к новым, но это не гарантируется и не выполняется при особых обстоятельствах, например при повышении ведомого сервера или после восстановления после сбоя. Поэтому считать, что все файлы старше `last_archived_wal` также были успешно заархивированы, небезопасно.

26.2.13. `pg_stat_io`

Представление `pg_stat_io` содержит по одной строке со статистикой ввода-вывода на уровне кластера для каждой комбинации типа обслуживающего процесса, целевого объекта ввода-вывода и контекста ввода-вывода. Комбинации, не имеющие смысла, исключаются.

В настоящее время отслеживаются операции ввода-вывода для отношений (например, таблиц, индексов). Однако операции ввода-вывода для отношений, которые не используют буферный кеш (например, при перемещении таблицы из одного табличного пространства в другое), в настоящее время не отслеживаются.

Таблица 26.23. Представление `pg_stat_io`

Тип столбца **Описание**
`backend_type` `text` Тип обслуживающего процесса (например, фоновый рабочий процесс, рабочий процесс автоочистки). Чтобы узнать больше о `backend_type` , обратитесь к описанию представления `pg_stat_activity` . По некоторым типам обслуживающих процессов (`backend_type`) не накапливается статистика операций ввода-вывода, и она не включаются в представление.
`object` `text` Целевой объект операции ввода-вывода. Возможные значения: • `relation`: постоянные отношения. • `temp relation`: временные отношения.
`context` `text` Контекст операции ввода-вывода. Возможные значения: • `normal`: используемый по умолчанию или стандартный контекст для операции ввода-вывода этого типа. Например, по умолчанию данные отношений считываются и записываются через

Тип столбца	**Описание**
	общий буфер, то есть операции чтения и записи, выполняемые через буфер, будут учитываться с контекстом `normal`. • `vacuum`: контекст для операций ввода-вывода, выполняемых за пределами общих буферов при очистке и анализе постоянных отношений. Операции очистки временных таблиц используют тот же пул локальных буферов, что и другие операции ввода-вывода с временными таблицами, поэтому учитываются с контекстом `normal`. • `bulkread`: некоторые большие операции чтения, выполняемые за пределами общих буферов, например последовательное сканирование большой таблицы. • `bulkwrite`: некоторые большие операции записи, выполняемые за пределами общих буферов, например `COPY`.
`reads` `bigint`	Количество операций чтения, размер каждой указан в `op_bytes`.
`read_time` `double precision`	Время, затраченное операциями чтения, в миллисекундах (если включён параметр track_io_timing; в противном случае 0).
`writes` `bigint`	Количество операций записи, размер каждой указан в `op_bytes`.
`write_time` `double precision`	Время, затраченное операциями записи, в миллисекундах (если включён параметр track_io_timing; в противном случае 0).
`writebacks` `bigint`	Количество блоков размером `op_bytes`, запрошенных процессом для записи ядром в постоянное хранилище.
`writeback_time` `double precision`	Время, затраченное на операции отложенной записи, в миллисекундах (если параметр track_io_timing включён, в противном случае 0). Сюда входит время, потраченное на постановку в очередь запросов на запись, и в принципе время, потраченное на запись «грязных» данных.
`extends` `bigint`	Количество операций расширения отношений, размер каждой указан в `op_bytes`.
`extend_time` `double precision`	Время, затраченное операциями расширения, в миллисекундах (если включён параметр track_io_timing; в противном случае 0).
`op_bytes` `bigint`	Число байтов на единицу ввода-вывода, прочитанную, записанную или расширенную. Чтение, запись и расширение отношений выполняются в единицах `block_size`, рассчитанных из параметра `BLCKSZ`, задаваемого в процессе сборки и по умолчанию равного `8192`.
`hits` `bigint`	Сколько раз нужный блок был найден в общем буфере.
`evictions` `bigint`	Сколько раз блок был записан из общего или локального буфера, чтобы его можно было переиспользовать. В контексте `normal` подсчитывает количество вытеснений блока из буфера и замены другим блоком. В контекстах `bulkwrite`, `bulkread` и `vacuum` подсчитывает количество вытеснений блока из общего буфера, чтобы добавить общий буфер в отдельный кольцевой буфер с ограниченным размером для использования в массовой операции ввода-вывода.
`reuses` `bigint`	

Тип столбца Описание
Сколько раз существующий буфер в кольцевом буфере с ограниченным размером за пределами общих буферов был повторно использован как часть операции ввода-вывода в контекстах `bulkread`, `bulkwrite` или `vacuum`.
`fsyncs` `bigint` Количество вызовов `fsync`. Они отслеживаются только в контексте `normal`.
`fsync_time` `double precision` Время, затраченное операциями fsync, в миллисекундах (если включён параметр track_io_timing; в противном случае 0).
`stats_reset` `timestamp with time zone` Последнее время сброса этих статистических данных.

Некоторые типы обслуживающих процессов никогда не выполняют операции ввода-вывода с некоторыми объектами ввода-вывода и/или в некоторых контекстах ввода-вывода. Эти строки исключаются из представления. Например, процесс контрольной точки не обрабатывает временные таблицы, поэтому не будет строк для объектов типа `checkpointer` объектов `temp relation`.

Кроме того, некоторые операции ввода-вывода никогда не будут выполняться ни некоторыми типами обслуживающих процессов, ни с некоторыми объектами ввода-вывода и/или в некоторых контекстах ввода-вывода. Эти ячейки будут иметь значение NULL. Например, `fsync` временных таблиц не поддерживается, поэтому `fsyncs` будет иметь значение NULL для `object temp relation`. Кроме того, фоновый рабочий процесс записи не выполняет чтение, поэтому `reads` будет иметь значение NULL в строках для `backend_type background writer`.

`pg_stat_io` можно использовать для информирования о настройке базы данных. Например:

- Большое количество вытеснений (`evictions`) в столбце может указывать на необходимость увеличения общих буферов.
- Чтобы гарантировать сохранение данных в постоянном хранилище, клиентские процессы задействуют процесс контрольных точек. Большое количество `fsyncs` со стороны `client backend` может указывать на неправильную настройку общих буферов или процесса контрольных точек. Настройка этого процесса подробно описана в Разделе 27.5.
- Обычно клиентские процессы должны иметь возможность задействовать вспомогательные процессы, такие как процесс контрольных точек и процесс фоновой записи, для записи как можно большего количества «грязных» данных. Большое количество записей со стороны клиентских обслуживающих процессов может указывать на неправильную настройку общих буферов или процесса контрольных точек. Настройка процесса контрольных точек подробно описана в Разделе 27.5.

Примечание

Столбцы, отслеживающие время ввода-вывода, будут ненулевыми, только если включён параметр track_io_timing. Следует с осторожностью обращаться к этим столбцам совместно с соответствующими столбцами, отслеживающими количество операций ввода-вывода, если параметр `track_io_timing` не был включён в течение всего времени с момента последнего сброса статистики.

26.2.14. `pg_stat_bgwriter`

В представлении `pg_stat_bgwriter` всегда будет одна строка, содержащая данные о текущем состоянии процесса фоновой записи в кластере.

Таблица 26.24. Представление `pg_stat_bgwriter`

Тип столбца	**Описание**
`buffers_clean bigint`	Количество буферов, записанных фоновым процессом записи
`maxwritten_clean bigint`	Сколько раз фоновый процесс записи останавливал сброс грязных страниц на диск из-за того, что записал слишком много буферов
`buffers_alloc bigint`	Количество выделенных буферов
`stats_reset timestamp with time zone`	Последнее время сброса этих статистических данных

26.2.15. `pg_stat_checkpointer`

В представлении `pg_stat_checkpointer` всегда будет одна строка, содержащая данные о текущем состоянии процесса контрольных точек в кластере.

Таблица 26.25. Представление `pg_stat_checkpointer`

Тип столбца	**Описание**
`num_timed bigint`	Количество запланированных из-за тайм-аута контрольных точек. Обратите внимание, что при бездействии сервера с момента последней контрольной точки, они могут пропускаться, и это значение учитывает как пройденные, так и пропущенные контрольные точки
`num_requested bigint`	Количество запрошенных контрольных точек, которые уже были выполнены
`restartpoints_timed bigint`	Количество запланированных точек перезапуска из-за тайм-аута или после неудачной попытки выполнить перезапуск
`restartpoints_req bigint`	Количество запрошенных точек перезапуска
`restartpoints_done bigint`	Количество точек перезапуска, которые уже были выполнены
`write_time double precision`	Общее время, затраченное на этап обработки контрольных точек и точек перезапуска, на котором файлы записываются на диск, в миллисекундах
`sync_time double precision`	Общее время, затраченное на этап обработки контрольных точек и точек перезапуска, на котором файлы синхронизируются с диском, в миллисекундах
`buffers_written bigint`	Количество буферов, записанных при выполнении контрольных точек и точек перезапуска
`stats_reset timestamp with time zone`	Последнее время сброса этих статистических данных

26.2.16. `pg_stat_wal`

В представлении `pg_stat_wal` всегда будет одна строка, содержащая данные о работе WAL в кластере.

Таблица 26.26. Представление `pg_stat_wal`

Тип столбца **Описание**
`wal_records bigint` Общее число сгенерированных записей WAL
`wal_fpi bigint` Общее число сгенерированных образов полных страниц в WAL
`wal_bytes numeric` Общий сгенерированный объём WAL в байтах
`wal_buffers_full bigint` Сколько раз данные WAL записывались на диск из-за переполнения буферов WAL
`wal_write bigint` Сколько раз буферы WAL были записаны на диск функцией `XLogWrite`. Более подробную информацию о внутренней функции WAL `XLogWrite` можно найти в Разделе 27.5.
`wal_sync bigint` Сколько раз файлы WAL сбрасывались на диск функцией `issue_xlog_fsync` (если параметр fsync включён (`on`) и wal_sync_method имеет значение `fdatasync`, `fsync` или `fsync_writethrough`, в противном случае будет выведен ноль). Более подробную информацию о внутренней функции WAL `issue_xlog_fsync` можно найти в Разделе 27.5.
`wal_write_time double precision` Общее время, затраченное на запись буферов WAL на диск функцией `XLogWrite`, в миллисекундах (если включён параметр track_wal_io_timing, в противном случае 0). Также учитывается время синхронизации, когда `wal_sync_method` имеет значение `open_datasync` или `open_sync`.
`wal_sync_time double precision` Общее время, затраченное на синхронизацию файлов WAL с диском функцией `issue_xlog_fsync`, в миллисекундах (если включён `track_wal_io_timing`, `fsync` имеет значение `on`, а `wal_sync_method` имеет значение `fdatasync`, `fsync` или `fsync_writethrough`, в противном случае 0).
`stats_reset timestamp with time zone` Последнее время сброса этих статистических данных

26.2.17. `pg_stat_database`

Представление `pg_stat_database` содержит по одной строке со статистикой уровня базы данных для каждой базы кластера, и ещё одну для общих объектов.

Таблица 26.27. Представление `pg_stat_database`

Тип столбца **Описание**
`datid oid` OID базы данных либо 0 для объектов, относящихся к общим отношениям
`datname name` Имя базы данных либо `NULL` для общих объектов.
`numbackends integer` Количество обслуживающих процессов, в настоящее время подключённых к этой базе данных, либо `NULL` для общих объектов. Это единственный столбец в представлении, значение в котором отражает текущее состояние; все другие столбцы возвращают суммарные значения со времени последнего сброса статистики.
`xact_commit bigint` Количество зафиксированных транзакций в этой базе данных

Тип столбца	**Описание**
`xact_rollback bigint`	Количество транзакций в этой базе данных, для которых был выполнен откат транзакции
`blks_read bigint`	Количество прочитанных дисковых блоков в этой базе данных
`blks_hit bigint`	Сколько раз дисковые блоки обнаруживались в буферном кеше, так что чтение с диска не потребовалось (в значение входят только случаи обнаружения в буферном кеше Postgres Pro, а не в кеше файловой системы ОС)
`tup_returned bigint`	Количество «живых» строк, выбранных при последовательном сканировании, и записей индекса, выбранных при сканировании индексов, в этой базе данных
`tup_fetched bigint`	Количество «живых» строк, выбранных при сканировании индексов в этой базе данных
`tup_inserted bigint`	Количество строк, вставленное запросами в этой базе данных
`tup_updated bigint`	Количество строк, изменённое запросами в этой базе данных
`tup_deleted bigint`	Количество строк, удалённое запросами в этой базе данных
`conflicts bigint`	Количество запросов, отменённых из-за конфликта с восстановлением в этой базе данных. (Конфликты происходят только на ведомых серверах; подробности в `pg_stat_database_conflicts` .)
`temp_files bigint`	Количество временных файлов, созданных запросами в этой базе данных. Подсчитываются все временные файлы независимо от причины их создания (например, для сортировки или для хеширования) и независимо от установленного значения log_temp_files
`temp_bytes bigint`	Общий объём данных, записанных во временные файлы запросами в этой базе данных. Учитываются все временные файлы, вне зависимости от того, по какой причине они созданы и вне зависимости от значения log_temp_files.
`deadlocks bigint`	Количество взаимных блокировок, зафиксированное в этой базе данных
`checksum_failures bigint`	Количество ошибок контрольных сумм в страницах данных этой базы (или общего объекта) либо NULL, если контрольные суммы не проверяются.
`checksum_last_failure timestamp with time zone`	Время выявления последней ошибки контрольной суммы в страницах данных этой базы (или общего объекта) либо NULL, если контрольные суммы не проверяются.
`blk_read_time double precision`	Время, которое затратили обслуживающие процессы в этой базе на чтение блоков из файлов данных, в миллисекундах (если включён параметр track_io_timing; в противном случае 0).
`blk_write_time double precision`	Время, которое затратили обслуживающие процессы в этой базе на запись блоков в файлы данных, в миллисекундах (если включён параметр track_io_timing; в противном случае 0)

Тип столбца **Описание**
`session_time double precision` Длительность сеансов в этой базе в миллисекундах (обратите внимание, что статистика обновляется только при изменении состояния сеанса, поэтому, если сеансы простаивают в течение длительного времени, оно не будет учитываться)
`active_time double precision` Время, затраченное на выполнение операторов SQL в этой базе, в миллисекундах (соответствует состояниям `active` и `fastpath function call` в `pg_stat_activity`)
`idle_in_transaction_time double precision` Время простоя в транзакциях в этой базе в миллисекундах (соответствует состояниям `idle in transaction` и `idle in transaction (aborted)` в `pg_stat_activity`)
`sessions bigint` Общее количество сеансов, относящихся к этой базе
`sessions_abandoned bigint` Количество сеансов в этой базе данных, прерванных из-за потери соединения с клиентом
`sessions_fatal bigint` Количество сеансов в этой базе данных, прерванных из-за критических ошибок
`sessions_killed bigint` Количество сеансов в этой базе данных, прерванных из-за вмешательства оператора
`stats_reset timestamp with time zone` Последнее время сброса этих статистических данных

26.2.18. `pg_stat_database_conflicts`

Представление `pg_stat_database_conflicts` для каждой базы данных будет содержать по одной строке со статистикой на уровне базы по отменам запросов, произошедшим вследствие конфликтов с процессами восстановления на ведомых серверах. В этом представлении будет содержаться только информация по ведомым серверам, поскольку на ведущих серверах конфликты не возникают.

Таблица 26.28. Представление `pg_stat_database_conflicts`

Тип столбца **Описание**
`datid oid` OID базы данных
`datname name` Имя базы данных
`confl_tablespace bigint` Количество запросов в этой базе данных, отменённых из-за того, что табличные пространства были удалены
`confl_lock bigint` Количество запросов в этой базе данных, отменённых по истечении времени ожидания блокировки
`confl_snapshot bigint` Количество запросов в этой базе данных, отменённых из-за устаревших снимков данных
`confl_bufferpin bigint` Количество запросов в этой базе данных, отменённых из-за прикреплённых страниц буфера
`confl_deadlock bigint`

Тип столбца	Описание
	Количество запросов в этой базе данных, отменённых из-за взаимных блокировок
`confl_active_logicalslot` `bigint`	Количество использований логических слотов в этой базе данных, отменённых из-за старых снимков или слишком низкого значения wal_level на ведущем сервере

26.2.19. `pg_stat_all_tables`

Представление `pg_stat_all_tables` будет содержать по одной строке на каждую таблицу в текущей базе данных (включая таблицы TOAST) со статистикой по обращениям к этой таблице. Представления `pg_stat_user_tables` и `pg_stat_sys_tables` содержат ту же самую информацию, но отфильтрованную так, чтобы показывать только пользовательские и системные таблицы соответственно.

Таблица 26.29. Представление `pg_stat_all_tables`

Тип столбца	Описание
`relid` `oid`	OID таблицы
`schemaname` `name`	Имя схемы, в которой расположена эта таблица
`relname` `name`	Имя таблицы
`seq_scan` `bigint`	Количество последовательных чтений, произведённых в этой таблице
`last_seq_scan` `timestamp with time zone`	Время последнего последовательного сканирования этой таблицы, исходя из времени последней остановки транзакции
`seq_tup_read` `bigint`	Количество «живых» строк, прочитанных при последовательных чтениях
`idx_scan` `bigint`	Количество сканирований по индексу, произведённых в этой таблице
`last_idx_scan` `timestamp with time zone`	Время последнего последовательного сканирования индекса этой таблицы, исходя из времени последней остановки транзакции
`idx_tup_fetch` `bigint`	Количество «живых» строк, отобранных при сканированиях по индексу
`n_tup_ins` `bigint`	Общее количество вставленных строк
`n_tup_upd` `bigint`	Общее количество изменённых строк. (Включая изменения строк, учитываемые в `n_tup_hot_upd` и `n_tup_newpage_upd` , а также остальные изменения, не относящиеся к HOT.)
`n_tup_del` `bigint`	Общее количество удалённых строк
`n_tup_hot_upd` `bigint`	Количество строк, изменённых по схеме HOT. Для таких изменений не требуются последующие версии в индексах.
`n_tup_newpage_upd` `bigint`	

Тип столбца **Описание**
Количество изменённых строк с переходом новой версии строки на *новую* страницу кучи, оставляющих после себя исходную версию с полем `t_ctid`, которое указывает на другую страницу кучи. Это всегда изменения, не относящиеся к HOT.
`n_live_tup` `bigint` Оценочное количество «живых» строк
`n_dead_tup` `bigint` Оценочное количество «мёртвых» строк
`n_mod_since_analyze` `bigint` Оценочное число строк, изменённых в этой таблице с момента последнего сбора статистики
`n_ins_since_vacuum` `bigint` Примерное число строк, вставленных в эту таблицу с момента последней очистки
`last_vacuum` `timestamp with time zone` Время последней очистки этой таблицы вручную (`VACUUM FULL` не учитывается)
`last_autovacuum` `timestamp with time zone` Время последней очистки таблицы фоновым процессом автоочистки
`last_analyze` `timestamp with time zone` Время последнего выполнения сбора статистики для этой таблицы вручную
`last_autoanalyze` `timestamp with time zone` Время последнего выполнения сбора статистики для этой таблицы фоновым процессом автоочистки
`vacuum_count` `bigint` Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)
`autovacuum_count` `bigint` Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки
`analyze_count` `bigint` Сколько раз сбор статистики для этой таблицы был выполнен вручную
`autoanalyze_count` `bigint` Сколько раз сбор статистики для этой таблицы был выполнен фоновым процессом автоочистки

26.2.20. `pg_stat_all_indexes`

Представление `pg_stat_all_indexes` для каждого индекса в текущей базе данных будет содержать по одной строке со статистикой по обращениям к этому индексу. Представления `pg_stat_user_indexes` и `pg_stat_sys_indexes` содержат ту же самую информацию, но отфильтрованную так, чтобы показывать только пользовательские и системные индексы соответственно.

Таблица 26.30. Представление `pg_stat_all_indexes`

Тип столбца **Описание**
`relid` `oid` OID таблицы для индекса
`indexrelid` `oid` OID индекса
`schemaname` `name` Имя схемы, в которой расположен индекс
`relname` `name`

Тип столбца	**Описание**
	Имя таблицы для индекса
`indexrelname` `name`	Имя индекса
`idx_scan` `bigint`	Количество произведённых сканирований по этому индексу
`last_idx_scan` `timestamp with time zone`	Время последнего сканирования этого индекса, исходя из времени последней остановки транзакции
`idx_tup_read` `bigint`	Количество элементов индекса, возвращённых при сканированиях по этому индексу
`idx_tup_fetch` `bigint`	Количество живых строк таблицы, отобранных при простых сканированиях по этому индексу

Индексы могут использоваться при простом сканировании по индексу, при сканировании «битовой карты» индекса и в работе оптимизатора. Результаты сканирования битовых карт разных индексов могут объединяться логическим умножением или сложением, поэтому когда применяются битовые карты, сложно связать выборки отдельных строк с определёнными индексами. Поэтому при сканировании битовых карт увеличиваются счётчики `pg_stat_all_indexes.idx_tup_read` для задействованных индексов и счётчик `pg_stat_all_tables.idx_tup_fetch` для каждой таблицы, а `pg_stat_all_indexes.idx_tup_fetch` не меняется. Оптимизатор тоже обращается к индексам для проверки переданных констант, значения которых оказываются вне диапазона, записанного в статистике оптимизатора, так как эта статистика может быть неактуальной.

Примечание

Значения счётчиков `idx_tup_read` и `idx_tup_fetch` могут различаться, даже если сканирование с использованием битовой карты не используется, поскольку `idx_tup_read` подсчитывает полученные из индекса элементы, а `idx_tup_fetch` — количество «живых» строк, выбранных из таблицы. Различие будет меньше, если «мёртвые» или ещё не зафиксированные строки будут извлекаться с использованием индекса или если для получения строк таблицы будет использоваться сканирование только по индексу.

Примечание

Запросы, которые используют определённые конструкции SQL для поиска строк, соответствующих любому значению из списка или массива нескольких скалярных значений (см. Раздел 9.25), выполняют несколько «примитивных» сканирований индекса (до одного сканирования на скалярное значение) во время выполнения запроса. Каждое внутреннее примитивное сканирование индекса увеличивает значение `pg_stat_all_indexes.idx_scan`, поэтому возможно, что количество сканирований индекса значительно превысит общее количество сканирований узла-исполнителя.

26.2.21. `pg_statio_all_tables`

Представление `pg_statio_all_tables` для каждой таблицы (включая таблицы TOAST) в текущей базе данных будет содержать по одной строке со статистикой по операциям ввода-вывода для этой таблицы. Представления `pg_statio_user_tables` и `pg_statio_sys_tables` содержат ту же самую информацию, но отфильтрованную так, чтобы показывать только пользовательские или системные таблицы соответственно.

Таблица 26.31. Представление `pg_statio_all_tables`

Тип столбца Описание
`relid` `oid` OID таблицы
`schemaname` `name` Имя схемы, в которой расположена эта таблица
`relname` `name` Имя таблицы
`heap_blks_read` `bigint` Количество дисковых блоков, прочитанных из этой таблицы
`heap_blks_hit` `bigint` Число попаданий в буфер для этой таблицы
`idx_blks_read` `bigint` Количество дисковых блоков, прочитанных из всех индексов этой таблицы
`idx_blks_hit` `bigint` Число попаданий в буфер для всех индексов по этой таблице
`toast_blks_read` `bigint` Количество прочитанных дисковых блоков TOAST (если есть) для этой таблицы
`toast_blks_hit` `bigint` Число попаданий в буфер в таблице TOAST для этой таблицы (если такие есть)
`tidx_blks_read` `bigint` Количество прочитанных дисковых блоков из индекса по TOAST (если есть) для этой таблицы
`tidx_blks_hit` `bigint` Число попаданий в буфер для индекса по TOAST (если есть) для этой таблицы

26.2.22. `pg_statio_all_indexes`

Представление `pg_statio_all_indexes` для каждого индекса в текущей базе данных будет содержать по одной строке со статистикой по операциям ввода-вывода для этого индекса. Представления `pg_statio_user_indexes` и `pg_statio_sys_indexes` содержат ту же самую информацию, но отфильтрованную так, чтобы показывать только пользовательские или системные индексы соответственно.

Таблица 26.32. Представление `pg_statio_all_indexes`

Тип столбца Описание
`relid` `oid` OID таблицы для индекса
`indexrelid` `oid` OID индекса
`schemaname` `name` Имя схемы, в которой расположен индекс
`relname` `name` Имя таблицы для индекса
`indexrelname` `name` Имя индекса
`idx_blks_read` `bigint` Количество дисковых блоков, прочитанных из этого индекса

Тип столбца **Описание**
`idx_blks_hit` `bigint` Число попаданий в буфер для этого индекса

26.2.23. `pg_statio_all_sequences`

Представление `pg_statio_all_sequences` для каждой последовательности в текущей базе данных будет содержать по одной строке со статистикой по операциям ввода-вывода для этой последовательности.

Таблица 26.33. Представление `pg_statio_all_sequences`

Тип столбца **Описание**
`relid` `oid` OID последовательности
`schemaname` `name` Имя схемы, в которой расположена эта последовательность
`relname` `name` Имя последовательности
`blks_read` `bigint` Количество дисковых блоков, прочитанных из этой последовательности
`blks_hit` `bigint` Число попаданий в буфер в этой последовательности

26.2.24. `pg_stat_user_functions`

Представление `pg_stat_user_functions` для каждой отслеживаемой функции будет содержать по одной строке со статистикой по выполнениям этой функции. Отслеживаемые функции определяются параметром track_functions.

Таблица 26.34. Представление `pg_stat_user_functions`

Тип столбца **Описание**
`funcid` `oid` OID функции
`schemaname` `name` Имя схемы, в которой расположена функция
`funcname` `name` Имя функции
`calls` `bigint` Сколько раз вызывалась функция
`total_time` `double precision` Общее время, затраченное на выполнение этой функции и всех других функций, вызванных ею, в миллисекундах
`self_time` `double precision` Общее время, затраченное на выполнение самой функции, без учёта других функций, которые были ею вызваны, в миллисекундах

26.2.25. `pg_stat_slru`

Postgres Pro обращается к некоторой информации на диске через кеши, работающие по принципу `SLRU` (*simple least-recently-used*, простое вытеснение давно не используемых). Представление `pg_stat_slru` содержит по одной строке для каждого SLRU-кеша и даёт статистику по обращениям к его страницам.

Для каждого `SLRU`-кеша, входящего в состав ядра сервера, существует параметр конфигурации, который управляет его размером, с добавленным суффиксом `_buffers`.

Таблица 26.35. Представление `pg_stat_slru`

Тип столбца	**Описание**
`name` `text`	Имя SLRU-кеша
`blks_zeroed` `bigint`	Количество блоков, обнулённых при инициализации
`blks_hit` `bigint`	Количество случаев, когда дисковые блоки обнаруживались в SLRU-кеше, так что чтение с диска не потребовалось (здесь учитываются только случаи обнаружения в этом кеше, а не в файловом кеше ОС)
`blks_read` `bigint`	Количество дисковых блоков, прочитанных через этот SLRU-кеш
`blks_written` `bigint`	Количество дисковых блоков, записанных из этого SLRU-кеша
`blks_exists` `bigint`	Количество блоков, проверенных на предмет наличия в этом SLRU-кеше
`flushes` `bigint`	Количество операций сброса «грязных» данных для этого SLRU-кеша
`truncates` `bigint`	Количество операций усечения для этого SLRU-кеша
`stats_reset` `timestamp with time zone`	Последнее время сброса этих статистических данных

26.2.26. Статистические функции

Статистическую информацию можно просматривать и другими способами. Для этого можно написать запросы, использующие те же функции доступа к статистике, что лежат в основе описанных выше стандартных представлений. За более подробной информацией, например, об именах этих функций, обратитесь к определениям этих стандартных представлений. (Например, в psql можно выполнить `\d+ pg_stat_activity`.) В качестве аргумента функции, предоставляющие доступ к статистике на уровне базы, принимают OID базы данных, по которой должна быть выдана информация. Функции, которые работают на уровне таблиц и индексов, принимают в качестве аргумента OID таблицы или индекса. Аргументом для функции, предоставляющей статистику на уровне функций, является OID функции. Обратите внимание, что с помощью этих функций можно получить информацию по таблицам, индексам и функциям исключительно в текущей базе данных.

Дополнительные функции, связанные с системой накопительной статистики, перечислены в Таблице 26.36.

Таблица 26.36. Дополнительные статистические функции

Функция	**Описание**
`pg_backend_pid` () → `integer`	Выдаёт идентификатор серверного процесса, обслуживающего текущий сеанс.
`pg_stat_get_activity` (`integer`) → `setof record`	

Функция **Описание**
Возвращает запись с информацией о серверном процессе с заданным ID или по одной строке для каждого активного серверного процесса, если был указан `NULL`. Возвращаемые поля являются подмножеством столбцов представления `pg_stat_activity`.
`pg_stat_get_snapshot_timestamp` () → `timestamp with time zone` Возвращает метку времени текущего снимка статистики или NULL, если снимок статистики не был получен. Снимок создаётся при первом обращении к накопительной статистике в транзакции, если для параметра `stats_fetch_consistency` задано значение `snapshot`.
`pg_stat_get_xact_blocks_fetched` (`oid`) → `bigint` Возвращает количество запросов на чтение блоков для таблицы или индекса в текущей транзакции. Это число минус `pg_stat_get_xact_blocks_hit` даёт количество обращений к ядру функцией `read()`; количество реальных физических операций чтения обычно меньше благодаря буферизации на уровне ядра.
`pg_stat_get_xact_blocks_hit` (`oid`) → `bigint` Возвращает количество запросов на чтение блоков, уже обнаруженных в кеше, для таблицы или индекса в текущей транзакции (не вызывает обращение к ядру функцией `read()`).
`pg_stat_clear_snapshot` () → `void` Сбрасывает текущий снимок статистики или кешированные данные.
`pg_stat_reset` () → `void` Обнуляет все статистические счётчики в текущей базе данных. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_stat_reset_shared` ([*`target`* `text` DEFAULT NULL]) → `void` Сбрасывает некоторые счётчики статистики кластера до нуля, в зависимости от аргумента. Аргумент *`target`* может принимать следующие значения: • `archiver`: Сбросить все счётчики, показанные в представлении `pg_stat_archiver`. • `bgwriter`: Сбросить все счётчики, показанные в представлении `pg_stat_bgwriter`. • `checkpointer`: Сбросить все счётчики, показанные в представлении `pg_stat_checkpointer`. • `io`: Сбросить все счётчики, показанные в представлении `pg_stat_io`. • `recovery_prefetch`: Сбросить все счётчики, показанные в представлении `pg_stat_recovery_prefetch`. • `slru`: Сбросить все счётчики, показанные в представлении `pg_stat_slru`. • `wal`: Сбросить все счётчики, показанные в представлении `pg_stat_wal`. • `NULL` или не указан: сбрасываются все счётчики вышеуказанных представлений. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_stat_reset_single_table_counters` (`oid`) → `void` Обнуляет статистику по отдельной таблице или индексу в текущей базе данных, а также общую статистику для всех баз данных в кластере. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_stat_reset_single_function_counters` (`oid`) → `void` Обнуляет статистику для отдельной функции в текущей базе данных.

Функция **Описание**
По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_stat_reset_slru` ([*target* text DEFAULT NULL]) → void Обнуляет статистику для отдельного SLRU-кеша или для всех SLRU-кешей в кластере. Если аргумент *target* равен `NULL` или не указан, обнуляются все счётчики, показанные в представлении `pg_stat_slru`, для всех SLRU-кешей. Также в аргументе можно передать текстовое значение `commit_timestamp`, `multixact_member`, `multixact_offset`, `notify`, `serializable`, `subtransaction` или `transaction`, чтобы сбросить счётчики только для указанного кеша. Если в аргументе передаётся `other` (на самом деле это может быть любая нераспознанная строка), обнулены будут все счётчики для всех кешей, кроме перечисленных поимённо выше. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_stat_reset_replication_slot` (text) → void Сбрасывает статистику указанного в аргументе слота репликации. Если в аргументе передаётся `NULL`, сбрасывает статистику для всех слотов репликации. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.
`pg_stat_reset_subscription_stats` (oid) → void Обнуляет статистику для отдельной подписки, отображаемую в представлении `pg_stat_subscription_stats`. Если в аргументе передаётся `NULL`, сбрасывается статистика для всех подписок. По умолчанию доступ к этой функции имеют только суперпользователи, но право на её выполнение (EXECUTE) можно дать и другим пользователям.

Предупреждение

Функция `pg_stat_reset()` также сбрасывает счётчики, на основании которых процесс автоочистки принимает решение о запуске очистки или анализа. В результате сброса процесс автоочистки может не выполнить необходимые действия, что чревато раздуванием таблиц, потерей актуальности статистики и тому подобными проблемами. Поэтому после сброса счётчиков рекомендуется выполнить `ANALYZE` на уровне базы.

Функция `pg_stat_get_activity`, на которой основано представление `pg_stat_activity`, возвращает набор строк, содержащих всю доступную информацию о каждом серверном процессе. Иногда может быть удобнее получить только часть этой информации. В таких случаях можно использовать набор более старых функций, дающих доступ к статистике на уровне серверных процессов; эти функции описаны в Таблице 26.37. Они используют идентификатор серверного процесса, который представляет собой небольшое целое число (>= 0), отличное от идентификатора любого параллельного сеанса, хотя по завершении сеанса идентификатор сразу может переиспользоваться. Идентификатор серверного процесса используется, помимо прочего, для определения временной схемы сеанса, если она есть. Функция `pg_stat_get_backend_idset` генерирует по одной строке для каждого активного серверного процесса, что необходимо для вызова этих функций. Например, для того, чтобы отобразить значения PID и текущие запросы всех серверных процессов:

```
SELECT pg_stat_get_backend_pid(backendid) AS pid,
       pg_stat_get_backend_activity(backendid) AS query
FROM pg_stat_get_backend_idset() AS backendid;
```

Таблица 26.37. Статистические функции на уровне серверных процессов

Функция **Описание**
`pg_stat_get_backend_activity` (`integer`) → `text` Выдаёт текст последнего запроса этого серверного процесса.
`pg_stat_get_backend_activity_start` (`integer`) → `timestamp with time zone` Выдаёт время начала выполнения последнего запроса в рамках данного процесса.
`pg_stat_get_backend_client_addr` (`integer`) → `inet` Выдаёт IP-адрес клиента, подключённого к этому серверному процессу.
`pg_stat_get_backend_client_port` (`integer`) → `integer` Выдаёт номер TCP-порта, который клиент использует для взаимодействия с сервером.
`pg_stat_get_backend_dbid` (`integer`) → `oid` Выдаёт OID базы данных, к которой подключён этот серверный процесс.
`pg_stat_get_backend_idset` () → `setof integer` Выдаёт идентификаторы активных в настоящий момент серверных процессов.
`pg_stat_get_backend_pid` (`integer`) → `integer` Выдаёт идентификатор (PID) этого серверного процесса.
`pg_stat_get_backend_start` (`integer`) → `timestamp with time zone` Выдаёт время, когда был запущен данный процесс.
`pg_stat_get_backend_subxact` (`integer`) → `record` Возвращает запись с информацией о подтранзакциях обслуживающего процесса с указанным идентификатором. Возвращаемые поля: *`subxact_count`* , которое представляет собой количество подтранзакций в кеше подтранзакций обслуживающего процесса, и *`subxact_overflow`* , который указывает, переполнен кеш подтранзакций обслуживающего процесса или нет.
`pg_stat_get_backend_userid` (`integer`) → `oid` Выдаёт OID пользователя, подключённого к этому серверному процессу.
`pg_stat_get_backend_wait_event` (`integer`) → `text` Выдаёт имя ожидаемого события, если серверный процесс сейчас находится в состоянии ожидания, или NULL в противном случае. См. также Таблица 26.5 – Таблица 26.13.
`pg_stat_get_backend_wait_event_type` (`integer`) → `text` Выдаёт имя типа ожидаемого события, если серверный процесс сейчас находится в состоянии ожидания, или NULL в противном случае. За подробностями обратитесь к Таблице 26.4.
`pg_stat_get_backend_xact_start` (`integer`) → `timestamp with time zone` Выдаёт время начала текущей транзакции в данном процессе.

26.3. Просмотр информации о блокировках

Ещё одним удобным средством для отслеживания работы базы данных является системная таблица `pg_locks`. Она позволяет администратору базы просматривать информацию об имеющихся блокировках в менеджере блокировок. Например, это может использоваться для:

- просмотра всех имеющихся на данный момент блокировок, всех блокировок на отношения в определённой базе данных, всех блокировок на определённое отношение или всех блокировок, которые удерживает определённый сеанс Postgres Pro.
- определения отношения в текущей базе данных с наибольшим количеством неразрешённых блокировок (оно может быть причиной конкуренции между клиентами базы данных).

- определения воздействия конкуренции за блокировку на производительность базы данных в целом, а так же то, как меняется конкуренция в зависимости от загруженности базы.

Более детально представление `pg_locks` описано в Разделе 52.12. Более подробную информацию о блокировках и управлению параллельным доступом в Postgres Pro можно получить в Главе 13.

26.4. Отслеживание выполнения

В Postgres Pro имеется возможность отслеживать выполнение определённых команд. В настоящее время такое отслеживание поддерживается только для команд `ANALYZE`, `CLUSTER`, `CREATE INDEX`, `VACUUM`, `COPY` и BASE_BACKUP (то есть для команды репликации, которую выполняет pg_basebackup). В будущем эта поддержка может быть расширена.

26.4.1. Отслеживание выполнения ANALYZE

Во время выполнения `ANALYZE` представление `pg_stat_progress_analyze` будет содержать по одной строке для каждого обслуживающего процесса, выполняющего эту команду. Таблицы ниже показывают, какая информация будет отслеживаться, и поясняют, как её интерпретировать.

Таблица 26.38. Представление `pg_stat_progress_analyze`

Тип столбца Описание
`pid integer` Идентификатор (PID) обслуживающего процесса
`datid oid` OID базы данных, к которой подключён этот обслуживающий процесс.
`datname name` Имя базы данных, к которой подключён этот обслуживающий процесс.
`relid oid` OID анализируемой таблицы.
`phase text` Текущая фаза обработки. См. Таблицу 26.39.
`sample_blks_total bigint` Общее количество блоков кучи, которые попадут в выборку.
`sample_blks_scanned bigint` Количество просканированных блоков кучи.
`ext_stats_total bigint` Количество объектов расширенной статистики.
`ext_stats_computed bigint` Количество вычисленных объектов расширенной статистики. Этот счётчик увеличивается только в фазе `computing extended statistics` (вычисление расширенной статистики).
`child_tables_total bigint` Количество дочерних таблиц.
`child_tables_done bigint` Количество просканированных дочерних таблиц. Этот счётчик увеличивается только в фазе `acquiring inherited sample rows` (извлечение строк выборки через наследование).
`current_child_table_relid oid` OID дочерней таблицы, сканируемой в данный момент. Это поле содержит актуальное значение только в фазе `acquiring inherited sample rows` (извлечение строк выборки через наследование).

Таблица 26.39. Фазы ANALYZE

Фаза	Описание
`initializing`	Команда готовится начать сканирование кучи. Эта фаза должна быть очень быстрой.
`acquiring sample rows`	Команда сканирует таблицу с указанным `relid`, считывая строки выборки.
`acquiring inherited sample rows`	Команда сканирует дочерние таблицы, считывая строки выборки. Выполнение процедуры в этой фазе отражается в столбцах `child_tables_total`, `child_tables_done` и `current_child_table_relid`.
`computing statistics`	Команда вычисляет статистику по строкам выборки, полученным при сканировании таблицы.
`computing extended statistics`	Команда вычисляет расширенную статистику по строкам выборки, полученным при сканировании таблицы.
`finalizing analyze`	Команда вносит изменения в `pg_class`. После этой фазы `ANALYZE` завершит работу.

Примечание

Заметьте, что когда `ANALYZE` обрабатывает секционированную таблицу, все её секции также рекурсивно анализируются. В этом случае сначала сообщается о ходе выполнения `ANALYZE` для родительской таблицы, которое сопровождается сбором наследуемой статистики, а затем о ходе обработки каждой её секции.

26.4.2. Отслеживание выполнения CLUSTER

Во время выполнения `CLUSTER` или `VACUUM FULL` представление `pg_stat_progress_cluster` будет содержать по одной строке для каждого обслуживающего процесса, выполняющего любую из этих команд. Таблицы ниже показывают, какая информация будет отслеживаться, и поясняют, как её интерпретировать.

Таблица 26.40. Представление `pg_stat_progress_cluster`

Тип столбца Описание
`pid integer` Идентификатор (PID) обслуживающего процесса
`datid oid` OID базы данных, к которой подключён этот обслуживающий процесс.
`datname name` Имя базы данных, к которой подключён этот обслуживающий процесс.
`relid oid` OID обрабатываемой таблицы.
`command text` Выполняемая команда: `CLUSTER` или `VACUUM FULL`.
`phase text` Текущая фаза обработки. См. Таблицу 26.41.
`cluster_index_relid oid` Если таблица сканируется по индексу, это поле содержит OID данного индекса, а иначе — 0.
`heap_tuples_scanned bigint`

Тип столбца	Описание
	Число просканированных кортежей кучи. Этот счётчик увеличивается только в фазе `seq scanning heap`, `index scanning heap` или `writing new heap`.
`heap_tuples_written bigint`	Число записанных кортежей кучи. Этот счётчик увеличивается только в фазе `seq scanning heap`, `index scanning heap` или `writing new heap`.
`heap_blks_total bigint`	Общее число блоков кучи в таблице. Это число отражает состояние в начале фазы `seq scanning heap`.
`heap_blks_scanned bigint`	Число просканированных блоков кучи. Этот счётчик увеличивается только в фазе `seq scanning heap`.
`index_rebuild_count bigint`	Число перестроенных индексов. Это счётчик увеличивается только в фазе `rebuilding index`.

Таблица 26.41. Фазы CLUSTER и VACUUM FULL

Фаза	Описание
`initializing`	Команда готовится начать сканирование кучи. Эта фаза должна быть очень быстрой.
`seq scanning heap`	Команда в данный момент сканирует таблицу последовательным образом.
`index scanning heap`	`CLUSTER` в данный момент сканирует таблицу по индексу.
`sorting tuples`	`CLUSTER` в данный момент сортирует кортежи.
`writing new heap`	`CLUSTER` в данный момент записывает новую кучу.
`swapping relation files`	Команда в данный момент переставляет только что построенные файлы на место.
`rebuilding index`	Команда в данный момент перестраивает индекс.
`performing final cleanup`	Команда выполняет окончательную очистку. После этой фазы `CLUSTER` или `VACUUM FULL` завершит работу.

26.4.3. Отслеживание выполнения COPY

Во время выполнения `COPY` представление `pg_stat_progress_copy` будет содержать по одной строке для каждого обслуживающего процесса, выполняющего эту команду. Таблицы ниже показывают, какая информация будет отслеживаться, и поясняют, как её интерпретировать.

Таблица 26.42. Представление `pg_stat_progress_copy`

Тип столбца	Описание
`pid integer`	Идентификатор (PID) обслуживающего процесса
`datid oid`	OID базы данных, к которой подключён этот обслуживающий процесс.
`datname name`	Имя базы данных, к которой подключён этот обслуживающий процесс.
`relid oid`	OID таблицы, в которой выполняется команда `COPY`. При копировании из запроса `SELECT` это поле содержит `0`.

Тип столбца	**Описание**
`command` `text`	Выполняемая команда: `COPY FROM` или `COPY TO`.
`type` `text`	Метод ввода-вывода, посредством которого считываются или записываются данные: `FILE`, `PROGRAM`, `PIPE` (для `COPY FROM STDIN` и `COPY TO STDOUT`) или `CALLBACK` (используется, например, во время начальной синхронизации таблицы при логической репликации).
`bytes_processed` `bigint`	Число байт, уже обработанных командой `COPY`.
`bytes_total` `bigint`	Размер исходного файла для команды `COPY FROM` в байтах. Если определить размер нельзя, это поле содержит `0`.
`tuples_processed` `bigint`	Число кортежей, уже обработанных командой `COPY`.
`tuples_excluded` `bigint`	Количество необработанных кортежей, которые были исключены предложением `WHERE` команды `COPY`.
`tuples_skipped` `bigint`	Количество кортежей, пропущенных из-за того, что они содержат неверные данные. Этот счётчик увеличивается только тогда, когда для параметра `ON_ERROR` указано значение, отличное от `stop`.

26.4.4. Отслеживание выполнения CREATE INDEX

Во время выполнения `CREATE INDEX` или `REINDEX` представление `pg_stat_progress_create_index` будет содержать по одной строке для каждого обслуживающего процесса, создающего индексы в этот момент. Таблицы ниже показывают, какая информация будет отслеживаться, и поясняют, как её интерпретировать.

Таблица 26.43. Представление `pg_stat_progress_create_index`

Тип столбца	**Описание**
`pid` `integer`	Идентификатор серверного процесса, создающего индексы.
`datid` `oid`	OID базы данных, к которой подключён этот обслуживающий процесс.
`datname` `name`	Имя базы данных, к которой подключён этот обслуживающий процесс.
`relid` `oid`	OID таблицы, в которой создаётся индекс.
`index_relid` `oid`	OID создаваемого или перестраиваемого индекса. При выполнении `CREATE INDEX` в неблокирующем режиме содержит 0.
`command` `text`	Определённый тип команды: `CREATE INDEX`, `CREATE INDEX CONCURRENTLY`, `REINDEX` или `REINDEX CONCURRENTLY`.
`phase` `text`	Текущая фаза создания индекса. См. Таблицу 26.44.
`lockers_total` `bigint`	Общее число процессов, потребовавших ожидания, если таковые имеются.

Тип столбца **Описание**
`lockers_done bigint` Число процессов, ожидание которых уже завершено.
`current_locker_pid bigint` Идентификатор процесса, удерживающего конфликтующую блокировку в данный момент.
`blocks_total bigint` Общее число блоков, которые должны быть обработаны в текущей фазе.
`blocks_done bigint` Число блоков, уже обработанных в текущей фазе.
`tuples_total bigint` Общее число кортежей, которые должны быть обработаны в текущей фазе.
`tuples_done bigint` Число кортежей, уже обработанных в текущей фазе.
`partitions_total bigint` Общее количество секций, для которых должен быть создан или прикреплён индекс, включая как прямые, так и косвенные секции. Имеет значение `0`, когда выполняется `REINDEX` или если индекс не секционирован.
`partitions_done bigint` Количество секций, для которых уже был создан или прикреплён индекс, включая как прямые, так и косвенные секции. Имеет значение `0`, когда выполняется `REINDEX` или если индекс не секционирован.

Таблица 26.44. Фазы CREATE INDEX

Фаза	Описание
`initializing`	Инициализация — процедура `CREATE INDEX` или `REINDEX` подготавливается к созданию индекса. Эта фаза должна быть очень быстрой.
`waiting for writers before build`	Ожидание окончания записи перед построением — процедура `CREATE INDEX CONCURRENTLY` или `REINDEX CONCURRENTLY` ожидает завершения транзакций, которые удерживают блокировки записи и могут читать таблицу. Эта фаза пропускается при выполнении операции в неблокирующем режиме. Выполнение процедуры в этой фазе отражается в столбцах `lockers_total`, `lockers_done` и `current_locker_pid`.
`building index`	Построение индекса — код, реализующий метод доступа, строит индекс. В этой фазе методы доступа, поддерживающие отслеживание процесса, передают свои данные о текущем состоянии, и в этом столбце видна внутренняя фаза. Обычно ход построения индекса отражается в столбцах `blocks_total` и `blocks_done`, но также могут меняться и столбцы `tuples_total` и `tuples_done`.
`waiting for writers before validation`	Ожидание окончания записи перед проверкой — процедура `CREATE INDEX CONCURRENTLY` или `REINDEX CONCURRENTLY` ожидает завершения транзакций, которые удерживают блокировки записи и могут записывать в таблицу. Эта фаза пропускается при выполнении операции в неблокирующем режиме. Выполнение процедуры в этой фазе отражается в столбцах `lockers_total`, `lockers_done` и `current_locker_pid`.
`index validation: scanning index`	Проверка индекса: сканирование — процедура `CREATE INDEX CONCURRENTLY` сканирует индекс, находя кортежи, требующие

Фаза	Описание
	проверки. Эта фаза пропускается при выполнении операции в неблокирующем режиме. Выполнение процедуры в этой фазе отражается в столбцах `blocks_total` (показывающем общий размер индекса) и `blocks_done`.
`index validation: sorting tuples`	Проверка индекса: сортировка кортежей — процедура `CREATE INDEX CONCURRENTLY` сортирует результат фазы сканирования индекса.
`index validation: scanning table`	Проверка индекса: сканирование таблицы — процедура `CREATE INDEX CONCURRENTLY` сканирует таблицу, чтобы проверить кортежи индекса, собранные в предыдущих двух фазах. Эта фаза пропускается при выполнении операции в неблокирующем режиме. Выполнение процедуры в этой фазе отражается в столбцах `blocks_total` (показывающем общий размер таблицы) и `blocks_done`.
`waiting for old snapshots`	Ожидание старых снимков — процедура `CREATE INDEX CONCURRENTLY` или `REINDEX CONCURRENTLY` ожидает освобождения снимков теми транзакциями, которые могут видеть содержимое таблицы. Эта фаза пропускается при выполнении операции в неблокирующем режиме. Выполнение процедуры в этой фазе отражается в столбцах `lockers_total`, `lockers_done` и `current_locker_pid`.
`waiting for readers before marking dead`	Ожидание завершения чтения перед отключением старого индекса — процедура `REINDEX CONCURRENTLY` ожидает завершения транзакций, которые удерживают блокировки чтения, прежде чем пометить старый индекс как нерабочий. Эта фаза пропускается при выполнении операции в неблокирующем режиме. Выполнение процедуры в этой фазе отражается в столбцах `lockers_total`, `lockers_done` и `current_locker_pid`.
`waiting for readers before dropping`	Ожидание завершения чтения перед удалением старого индекса — процедура `REINDEX CONCURRENTLY` ожидает завершения транзакций, которые удерживают блокировки чтения, прежде чем удалить старый индекс. Эта фаза пропускается при выполнении операции в неблокирующем режиме. Выполнение процедуры в этой фазе отражается в столбцах `lockers_total`, `lockers_done` и `current_locker_pid`.

26.4.5. Отслеживание выполнения VACUUM

В процессе выполнения `VACUUM` представление `pg_stat_progress_vacuum` будет содержать по одной строке для каждого обслуживающего процесса (включая рабочие процессы автоочистки), производящего очистку в данный момент. Таблицы ниже показывают, какая информация будет отслеживаться, и поясняют, как её интерпретировать. Выполнение команд `VACUUM FULL` отслеживается через `pg_stat_progress_cluster`, так как и `VACUUM FULL`, и `CLUSTER` перезаписывают таблицу, тогда как обычная команда `VACUUM` модифицирует её саму. См. Подраздел 26.4.2.

Таблица 26.45. Представление `pg_stat_progress_vacuum`

Тип столбца Описание
`pid integer` Идентификатор (PID) обслуживающего процесса
`datid oid` OID базы данных, к которой подключён этот обслуживающий процесс.
`datname name`

Тип столбца	Описание
	Имя базы данных, к которой подключён этот обслуживающий процесс.
`relid` `oid`	OID очищаемой таблицы.
`phase` `text`	Текущая фаза очистки. См. Таблицу 26.46.
`heap_blks_total` `bigint`	Общее число блоков кучи в таблице. Это число отражает состояние в начале сканирования; блоки, добавленные позже, не будут (и не должны) обрабатываться текущей командой `VACUUM`.
`heap_blks_scanned` `bigint`	Число просканированных блоков кучи. Так как для оптимизации сканирования применяется карта видимости, некоторые блоки могут пропускаться без осмотра; пропущенные блоки входят в это общее число, так что по завершении очистки это число станет равно `heap_blks_total` . Этот счётчик увеличивается только в фазе `scanning heap`.
`heap_blks_vacuumed` `bigint`	Число очищенных блоков кучи. Если в таблице нет индексов, этот счётчик увеличивается только в фазе `vacuuming heap` (очистка кучи). Блоки, не содержащие «мёртвых» кортежей, при этом пропускаются, так что этот счётчик иногда может увеличиваться резкими рывками.
`index_vacuum_count` `bigint`	Количество завершённых циклов очистки индекса.
`max_dead_tuple_bytes` `bigint`	Объём данных «мёртвых» кортежей, который можно сохранить, прежде чем потребуется выполнить цикл очистки индекса, в зависимости от maintenance_work_mem.
`dead_tuple_bytes` `bigint`	Объём данных «мёртвых» кортежей, собранных со времени последнего цикла очистки индекса.
`num_dead_item_ids` `bigint`	Число идентификаторов «мёртвых» кортежей, собранных со времени последнего цикла очистки индекса.
`indexes_total` `bigint`	Общее число индексов, которые будут очищены или убраны. Это число сообщается в начале фазы `vacuuming indexes` или `cleaning up indexes`.
`indexes_processed` `bigint`	Число обработанных индексов. Этот счётчик увеличивается только во время фазы `vacuuming indexes` или `cleaning up indexes`.

Таблица 26.46. Фазы VACUUM

Фаза	Описание
`initializing`	Инициализация — `VACUUM` готовится начать сканирование кучи. Эта фаза должна быть очень быстрой.
`scanning heap`	Сканирование кучи — `VACUUM` в настоящее время сканирует кучу. При этом будет очищена и, если требуется, дефрагментирована каждая страница, а возможно, также будет произведена заморозка. Отслеживать процесс сканирования можно, следя за содержимым столбца `heap_blks_scanned` .
`vacuuming indexes`	Очистка индексов — `VACUUM` в настоящее время очищает индексы. Если у таблицы есть какие-либо индексы, эта фаза будет наблюдаться минимум единожды в процессе очистки,

Фаза	Описание
	после того, как куча будет просканирована полностью. Она может повторяться несколько раз в процессе очистки, если объёма maintenance_work_mem (или, в случае автоочистки, autovacuum_work_mem, если он задан) оказывается недостаточно для сохранения всех найденных «мёртвых» кортежей.
`vacuuming heap`	Очистка кучи — `VACUUM` в настоящее время очищает кучу. Очистка кучи отличается от сканирования, так как она происходит после каждой операции очистки индексов. Если `heap_blks_scanned` меньше чем `heap_blks_total`, система вернётся к сканированию кучи после завершения этой фазы; в противном случае она начнёт уборку индексов.
`cleaning up indexes`	Уборка индексов — `VACUUM` в настоящее время производит уборку в индексах. Это происходит после завершения полного сканирования кучи и очистки индексов и кучи.
`truncating heap`	Усечение кучи — `VACUUM` в настоящее время усекает кучу, чтобы возвратить операционной системе объём пустых страниц в конце отношения. Это происходит после уборки индексов.
`performing final cleanup`	Выполнение окончательной очистки — `VACUUM` выполняет окончательную очистку. На этой стадии `VACUUM` очищает карту свободного пространства, обновляет статистику в `pg_class` и передаёт информацию системе накопительной статистики. После этой фазы `VACUUM` завершит свою работу.

26.4.6. Отслеживание выполнение базового копирования

Когда приложение pg_basebackup или подобное выполняет базовое копирование, представление `pg_stat_progress_basebackup` будет содержать по одной строке для каждого процесса-передатчика WAL, выполняющего в данный момент команду репликации `BASE_BACKUP` и передающего копируемые данные. Таблицы ниже показывают, какая информация будет отслеживаться, и поясняют, как её интерпретировать.

Таблица 26.47. Представление `pg_stat_progress_basebackup`

Тип столбца Описание
`pid integer` Идентификатор процесса-передатчика WAL.
`phase text` Текущая фаза обработки. См. Таблицу 26.48.
`backup_total bigint` Общий объём данных, который будет передан. Оценка этого объёма выдаётся на момент начала фазы `streaming database files` (передача файлов данных). Заметьте, что это лишь приблизительная оценка, так как база данных может измениться в фазе `streaming database files` и в резервную копию позже может быть добавлен WAL. Это значение устанавливается равным `backup_streamed`, когда фактически переданный объём начинает превышать рассчитанный предварительно. Когда расчёт оценки в pg_basebackup отключён (то есть передан параметр `--no-estimate-size`), это поле содержит `NULL`.
`backup_streamed bigint` Объём переданных данных. Этот показатель увеличивается только в фазе `streaming database files` (передача файлов данных) или `transferring wal files` (передача файлов WAL).
`tablespaces_total bigint` Общее число табличных пространств, которые будут переданы.

Тип столбца **Описание**
`tablespaces_streamed` `bigint` Число переданных табличных пространств. Этот счётчик увеличивается только в фазе `streaming database files` (передача файлов данных).

Таблица 26.48. Фазы базового копирования

Фаза	**Описание**
`initializing`	Процесс-передатчик WAL готовится начать копирование. Эта фаза должна быть очень быстрой.
`waiting for checkpoint to finish`	Процесс-передатчик WAL в настоящий момент выполняет `pg_backup_start`, чтобы подготовиться к получению базовой копии, и ждёт завершения контрольной точки для начала копирования.
`estimating backup size`	Процесс-передатчик WAL в настоящий момент оценивает общее количество файлов данных, которые будут передаваться при создании базовой копии.
`streaming database files`	Процесс-передатчик WAL в настоящий момент передаёт файлы данных в качестве содержимого базовой резервной копии.
`waiting for wal archiving to finish`	Процесс-передатчик WAL в настоящий момент выполняет `pg_backup_stop`, чтобы закончить копирование, и ждёт успешного завершения архивации всех файлов WAL, необходимых для базовой копии. Если при запуске pg_basebackup был указан параметр `--wal-method=none` или `--wal-method=stream`, резервное копирования заканчивается сразу после данной фазы.
`transferring wal files`	Процесс-передатчик WAL в настоящее время переносит все файлы WAL, заполненные во время копирования. Эта фаза следует за фазой `waiting for wal archiving to finish`, только если при запуске pg_basebackup указывался параметр `--wal-method=fetch`. По окончании этой фазы резервное копирование завершается.

26.5. Мониторинг использования диска

В данном разделе рассказывается о том, как отслеживать использование дискового пространства в СУБД Postgres Pro.

26.5.1. Определение использования диска

Для каждой таблицы создаётся первичный дисковый файл кучи (heap), в котором хранится большая часть данных. Если в таблице есть столбцы с потенциально большими значениями, то также может быть и ассоциированный с этой таблицей файл TOAST, который используется для хранения слишком больших значений, не умещающихся в главной таблице (см. Раздел 63.2). На каждую таблицу TOAST, если она существует, будет существовать один индекс. Также там могут быть и индексы, ассоциированные с базовой таблицей. Каждая таблица и индекс хранятся в отдельном дисковом файле — возможно более чем в одном файле, если размер этого файла превышает один гигабайт. Преобразования имён для этих файлов описываются в Разделе 63.1.

Вы можете осуществлять мониторинг дискового пространства тремя способами: используя SQL-функции, перечисленные в Таблице 9.100, используя модуль oid2name или просматривая системные каталоги вручную. Вышеупомянутые SQL-функции являются наиболее простыми и рекомендуются для использования. В продолжении этого раздела рассказывается, как просматривать системные каталоги.

Используя psql после недавнего применения команд VACUUM или ANALYZE, вы можете выполнить такой запрос, чтобы увидеть сколько дискового пространства использует какая-либо таблица:

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

Каждая страница обычно равна 8Кб. (Помните, что `relpages` обновляется только командами `VACUUM`, `ANALYZE`, и несколькими командами DDL, такими как `CREATE INDEX`). Путь к файлу представляет интерес, если вы хотите проанализировать непосредственно файл на диске.

Чтобы посмотреть пространство, используемое таблицами TOAST, используйте следующий запрос:

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

Вы можете легко посмотреть размеры индексов:

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

      relname      | relpages
-------------------+----------
 customer_id_index |       26
```

Также легко найти самые большие таблицы и индексы, используя эту информацию:

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

26.5.2. Ошибка переполнения диска

Основная цель мониторинга дисков для администратора БД — предотвратить их переполнение. Если переполнится диск с данными, повреждения данных не произойдёт, но могут не выполняться полезные действия. Если же переполнится диск, содержащий файлы WAL, это может привести к аварийному сбою сервера с последующим его отключением.

Если вы не можете освободить дополнительное пространство на диске, удалив какие-либо другие файлы, то можно перенести часть файлов базы данных на другие файловые системы с помощью создания табличных пространств. Подробности об этом смотрите в Разделе 21.6.

Подсказка

Некоторые файловые системы плохо работают, когда они почти или совсем заполнены, так что не ждите пока диск заполнится полностью, чтобы выполнить необходимые действия.

Если ваша система поддерживает дисковые квоты для пользователей, вы должны позаботиться о квоте, выделенной для пользователя, от имени которого запускается сервер СУБД. Если эта квота будет исчерпана, последствия будут столь же негативными, как если просто закончится свободное место на диске.

Глава 27. Надёжность и журнал предзаписи

В этой главе объясняется, как обеспечивать надёжность Postgres Pro, и представлены подробности о журнале предзаписи.

27.1. Надёжность

Надёжность — это важное свойство любой серьёзной СУБД и Postgres Pro делает всё возможное, чтобы гарантировать надёжность своего функционирования. Один из аспектов надёжности состоит в том, что все данные записываются с помощью подтверждённых транзакций, которые сохраняются в энергонезависимой области, которая защищена от потери питания, сбоев операционной системы и аппаратных отказов (разумеется, за исключением отказа самой энергонезависимой области). Успешная запись данных в постоянное место хранения (диск или эквивалентный носитель) обычно всё, что требуется. Фактически, даже если компьютер полностью вышел из строя, если диски выжили, то они могут быть переставлены в другой похожий компьютер и все подтверждённые транзакции останутся неповреждёнными.

Хотя периодическая запись данных на пластины диска может выглядеть как простая операция, это не так, потому что диски значительно медленнее, чем оперативная память и процессор, а также потому что между оперативной памятью и пластинами диска есть некоторые механизмы кеширования. Во-первых, есть буферный кеш операционной системы, который кеширует частые запросы к блокам диска и комбинирует запись на диск. К счастью, все операционные системы предоставляют приложениям способ принудительной записи из буферного кеша на диск и Postgres Pro использует эту возможность. (Смотрите параметр wal_sync_method который отвечает за то как это делается.)

Далее, кеширование также может осуществляться контроллером диска; в особенности это касается RAID-контроллеров. В некоторых случаях это кеширование работает в режиме *сквозной записи*, что означает, что запись осуществляется на диск как только приходят данные. В других случаях, возможна работа в режиме *отложенной записи*, что означает, что запись осуществляется некоторое время спустя. Такой режим кеширования может создавать риск для надёжности, потому что память контроллера диска непостоянна и будет потеряна в случае потери питания. Лучшие контроллеры имеют так называемую *батарею резервного питания* (Battery-Backup Unit, BBU), которая сохраняет кеш контроллера на батарее, если пропадёт системное питание. После возобновления питания, данные, оставшиеся в кеше контроллера, будут записаны на диски.

И наконец, многие диски имеют внутренний кеш. На каких-то дисках он работает в режиме сквозной записи, на других — в режиме отложенной записи. В последнем случае с кешем диска связаны те же риски потери данных, что и с кешем контроллера дисков. В частности, кеш отложенной записи, сбрасывающийся при потере питания, часто имеют диски IDE и SATA потребительского класса. Также зависимый от питания кеш отложенной записи имеют многие SSD-накопители.

Обычно, такое кеширование можно выключить; однако то, как это делается, различается для операционной системы и для типа диска:

- В Linux параметры дисков IDE и SATA могут быть получены с помощью команды `hdparm -I`; кеширование записи включено, если за строкой `Write cache` следует `*`. Для выключения кеширования записи может быть использована команда `hdparm -W 0`. Параметры SCSI-дисков могут быть получены с помощью утилиты *sdparm*. Используйте `sdparm --get=WCE`, чтобы проверить, включено ли кеширование записи, и `sdparm --clear=WCE`, чтобы выключить его.
- Во FreeBSD параметры IDE-дисков могут быть получены с помощью команды `camcontrol identify`, а кеширование записи выключается при помощи установки параметра `hw.ata.wc=0` в файле `/boot/loader.conf`; Для SCSI-дисков параметры могут быть получены, используя команду `camcontrol identify`, а кеширование записи изменяется при помощи утилиты `sdparm`.

- В Solaris кешированием записи на диск управляет команда `format -e`. (Использование файловой системы Solaris ZFS, при включённом кешировании записи на диск, является безопасным, потому что она использует собственные команды сброса кеша на диск.)
- В Windows, если параметр `wal_sync_method` установлен в `open_datasync` (по умолчанию), кеширование записи на диск может быть выключено снятием галочки `My Computer\Open\`*`диск`*`\Properties\Hardware\Properties\Policies\Enable write caching on the disk`. В качестве альтернативы, можно установить параметр `wal_sync_method` в значение `fdatasync` (только для NTFS) или `fsync`, что предотвращает кеширование записи.
- В macOS кеширование записи можно отключить, установив для параметра `wal_sync_method` значение `fsync_writethrough`.

Новые модели SATA-дисков (которые соответствуют стандарту ATAPI-6 или более позднему) предлагают команду сброса кеша на диск (`FLUSH CACHE EXT`), а SCSI-диски уже давно поддерживают похожую команду `SYNCHRONIZE CACHE`. Эти команды недоступны из Postgres Pro напрямую, но некоторые файловые системы (например, ZFS, ext4), могут использовать их для сброса данных из кеша на пластины диска при включённом режиме кеша сквозной записи. К сожалению, такие файловые системы ведут себя неоптимально при комбинировании с батареей резервного питания (BBU) дискового контроллера. В таких случаях, команда синхронизации принуждает сохранять все данные на диск из кеша контроллера, сводя преимущество BBU к нулю. Вы можете запустить модуль pg_test_fsync, чтобы увидеть, что вы попали в эту ситуацию. Если это так, преимущества производительности BBU могут быть восстановлены с помощью выключения барьеров записи для файловой системы или переконфигурирования контроллера диска, если это возможно. Если барьеры записи выключены, убедитесь, что батарея годная; при отказе батареи может произойти потеря данных. Есть надежда, что разработчики файловых систем и контроллеров дисков, в конце концов, устранят это неоптимальное поведение.

Когда операционная система отправляет запрос на запись к аппаратному обеспечению для хранения данных, она мало что может сделать, чтобы убедиться, что данные действительно сохранены в какой-либо энергонезависимой области. Скорее, это является зоной ответственности администратора, убедиться в целостности данных на всех компонентах хранения. Избегайте дисковых контроллеров, которые не имеют батарей резервного питания для кеширования записи. На уровне диска, запретите режим отложенной записи, если диск не может гарантировать, что данные будут записаны перед выключением. Если вы используете SSD, знайте, что многие из них по умолчанию не выполняют команды сброса кеша на диск. Вы можете протестировать надёжность поведения подсистемы ввода-вывода, используя *`diskchecker.pl`*.

Другой риск потери данных состоит в самой записи на пластины диска. Пластины диска разделяются на секторы, обычно по 512 байт каждый. Каждая операция физического чтения или записи обрабатывает целый сектор. Когда дисковый накопитель получает запрос на запись, он может соответствовать нескольким секторам по 512 байт (Postgres Pro обычно за один раз записывает 8192 байта или 16 секторов) и из-за отказа питания процесс записи может закончится неудачей в любое время, что означает, что некоторые из 512-байтовых секторов будут записаны, а некоторые нет. Чтобы защититься от таких сбоев, *перед* изменением фактической страницы на диске Postgres Pro периодически записывает полные образы страниц на постоянное устройство хранения WAL. Так во время восстановления после сбоя Postgres Pro может восстановить из WAL частично записанные страницы. Если файловая система защищена от частичной записи страниц (например, ZFS), отключить работу с образами страниц можно при помощи конфигурационного параметра full_page_writes. Батарея резервного питания (BBU) контроллера диска не защищает от частичной записи страниц, если не гарантируется, что данные записаны в BBU как полные (8 КБ) страницы.

Postgres Pro также защищает от некоторых видов повреждения данных на устройствах хранения, которые могут произойти из-за аппаратных ошибок или из-за дефектов поверхности с течением времени, например, при операциях чтения/записи во время сборки мусора.

- Каждая индивидуальная запись в WAL защищена с помощью контрольной суммы по алгоритму CRC-32C (32-bit), что позволяет судить о корректности данных в записи. Значение CRC устанавливается, когда мы пишем каждую запись WAL и проверяется в ходе восстановления после сбоя, восстановления из архива, и при репликации.

- Страницы данных в настоящее время не защищаются контрольными суммами по умолчанию, хотя полные образы страниц, записанные в WAL, будут защищены; смотрите initdb для деталей о включении в страницы данных информации о контрольных суммах.
- Для внутренних структур данных, таких как `pg_xact`, `pg_subtrans`, `pg_multixact`, `pg_serial`, `pg_notify`, `pg_stat`, `pg_snapshots` не ведётся расчёт контрольной суммы, равно как и для страниц, защищённых посредством полностраничной записи. Однако там, где такие структуры данных являются постоянными, записи WAL пишутся таким образом, чтобы после сбоя было возможно аккуратно повторить последние изменения, а эти записи WAL защищаются так же, как описано выше.
- Файлы каталога `pg_twophase` защищены с помощью контрольной суммы CRC-32C.
- Временные файлы данных, используемые в больших SQL-запросах для сортировки, материализации и промежуточных результатов, в настоящее время не защищаются контрольной суммой, а изменения в этих файлах не отражаются в WAL.

Postgres Pro не защищает от корректируемых ошибок памяти; предполагается, что вы будете работать с памятью, которая использует промышленный стандарт коррекции ошибок (ECC, Error Correcting Codes) или лучшую защиту.

27.2. Контрольные суммы данных

По умолчанию страницы данных не защищены контрольными суммами, но при необходимости для кластера можно включить контрольные суммы. В таком случае каждая страница данных будет содержать контрольную сумму, рассчитываемую при записи и проверяемую при каждом чтении страницы. Контрольными суммами защищены только страницы данных, но не внутренние структуры данных и временные файлы.

Контрольные суммы можно включить при инициализации кластера с помощью initdb. Также контрольные суммы можно включить или отключить позднее, в остановленном кластере. Включить или отключить контрольные суммы данных можно на уровне всего кластера, но не для отдельной базы данных или таблицы.

Текущее состояние контрольных сумм в кластере можно узнать, выполнив команду `SHOW data_checksums`, которая возвращает значение неизменяемой конфигурационной переменной data_checksums.

При попытке восстановить страницы после повреждения иногда нужно обойти защиту, обеспечиваемую контрольными суммами. Для этого можно временно установить параметр конфигурации ignore_checksum_failure.

27.2.1. Включение контрольных сумм в остановленном кластере

Чтобы проверить, включить или отключить контрольные суммы данных в остановленном кластере, можно использовать утилиту pg_checksums.

27.3. Журнал предзаписи (WAL)

Журнал предзаписи (WAL) — это стандартный метод обеспечения целостности данных. Детальное описание можно найти в большинстве книг (если не во всех) по обработке транзакций. Вкратце, основная идея WAL состоит в том, что изменения в файлах с данными (где находятся таблицы и индексы) должны записываться только после того, как эти изменения были занесены в WAL, т. е. после того как записи WAL, описывающие данные изменения, будут сохранены на постоянное устройство хранения. Если следовать этой процедуре, то записывать страницы данных на диск после подтверждения каждой транзакции нет необходимости, потому что мы знаем, что если случится сбой, то у нас будет возможность восстановить базу данных с помощью журнала: любые изменения, которые не были применены к страницам с данными, могут быть воссозданы из записей WAL. (Это называется восстановлением с воспроизведением или REDO.)

Подсказка

Поскольку WAL восстанавливает содержимое файлов базы данных, журналируемая файловая система не является необходимой для надёжного хранения файлов с данными или файлов WAL. Фактически, журналирование может снизить производительность, особенно если журналирование заставляет сохранять *данные* файловой системы на диск. К счастью, такое сохранение при журналировании часто можно отключить с помощью параметров монтирования файловой системы, например, `data=writeback` для файловой системы ext3 в Linux. С другой стороны, с журналируемыми файловыми системами увеличивается скорость загрузки после сбоя.

Результатом использования WAL является значительное уменьшение количества запросов записи на диск, потому что для гарантии, что транзакция подтверждена, в записи на диск нуждается только файл WAL, а не каждый файл данных, изменённый в результате транзакции. Файл WAL записывается последовательно и таким образом затраты на синхронизацию WAL намного меньше, чем затраты на запись страниц с данными. Это особенно справедливо для серверов, которые обрабатывают много маленьких транзакций, изменяющих разные части хранилища данных. Таким образом, когда сервер обрабатывает множество мелких конкурентных транзакций, для подтверждения многих транзакций достаточно одного вызова `fsync` на файл WAL.

WAL также делает возможным поддержку резервного копирования «на ходу» и восстановления на определённый момент времени, как описывается в Разделе 24.3. С помощью архивирования данных WAL поддерживается возврат к любому моменту времени, который доступен в данных WAL: мы просто устанавливаем предыдущую физическую резервную копию базы данных и воспроизводим WAL до нужного момента времени. Более того, физическая резервная копия не должна быть мгновенным снимком состояния баз данных — если она была сделана некоторое время назад, воспроизведение WAL за этот период исправит все внутренние несоответствия.

27.4. Асинхронное подтверждение транзакций

Асинхронная фиксация — это возможность завершать транзакции быстрее, ценой того, что в случае краха СУБД последние транзакции могут быть потеряны. Для многих приложений такой компромисс приемлем.

Как описано в предыдущей части, подтверждение транзакции обычно *синхронное*: сервер ждёт сохранения записей WAL транзакции в постоянном хранилище, прежде чем сообщить клиенту об успешном завершении. Таким образом, клиенту гарантируется, что транзакция, которую подтвердил сервер, будет защищена, даже если сразу после этого произойдёт крах сервера. Однако для коротких транзакций данная задержка будет основной составляющей общего времени транзакции. В режиме асинхронного подтверждения сервер сообщает об успешном завершении сразу, как только транзакция будет завершена логически, прежде чем сгенерированные записи WAL фактически будут записаны на диск. Это может значительно увеличить производительность при выполнении небольших транзакций.

Асинхронное подтверждение транзакций приводит к риску потери данных. Существует короткое окно между отчётом о завершении транзакции для клиента и временем, когда транзакция реально подтверждена (т. е. гарантируется, что она не будет потеряна в случае краха сервера). Таким образом, асинхронное подтверждение транзакций не должно использоваться, если клиент будет выполнять внешние действия, опираясь на предположение, что транзакция будет сохранена. Например, банк конечно не должен использовать асинхронное подтверждение для транзакций в банкоматах, выдающих наличные. Но во многих случаях, таких как журналирование событий, столь серьёзная гарантия сохранности данных не нужна.

Риск потери данных при использовании асинхронного подтверждения транзакций — это не риск повреждения данных. Если случился крах СУБД, она будет восстановлена путём воспроизведения WAL до последней записи, которая была записана на диск. Таким образом, будет восстановлено целостное состояние СУБД, но любые транзакции, которые ещё не были сохранены на диск, в этом состоянии не будут отражены. Чистый эффект будет заключаться в потере нескольких последних транзакций. Поскольку транзакции воспроизводятся в том же порядке, в котором подтверждались, воспроизведение не нарушает целостность — например, если транзакция «B» выполнила изменения, которые влияют на предыдущую транзакцию «A», то не может быть такого, что изменения, выполненные «A» были потеряны, а изменения, внесённые «B» сохранены.

Пользователь может выбрать режим подтверждения для каждой транзакции, так что возможен конкурентный запуск транзакций в синхронном и асинхронном режиме. Это позволяет достичь гибкого компромисса между производительностью и конечно надёжностью транзакций. Режим подтверждения транзакций управляется параметром synchronous_commit, который может быть изменён любым из способов, пригодным для установки параметров конфигурации. Режим, используемый для какой-либо конкретной транзакции, зависит от значения `synchronous_commit`, которое действует на момент начала этой транзакции.

Некоторые команды, например `DROP TABLE`, принудительно запускают синхронное подтверждение транзакции, независимо от значения `synchronous_commit`. Это сделано для того, чтобы иметь уверенность в целостности данных между файловой системой сервера и логическим состоянием базы данных. Команды, которые поддерживают двухфазное подтверждение транзакций, такие как `PREPARE TRANSACTION`, также всегда синхронные.

Если во время окна риска между асинхронным подтверждением транзакции и сохранением на диск записей WAL, происходит крах СУБД, то изменения, сделанные во время этой транзакции *будут* потеряны. Продолжительность окна риска ограничена, потому что фоновый процесс («WAL writer»), сохраняет не записанные записи WAL на диск каждые wal_writer_delay миллисекунд. Фактически, максимальная продолжительность окна риска составляет трёхкратное значение `wal_writer_delay`, потому что WAL writer разработан так, чтобы сразу сохранять целые страницы во время периодов занятости.

Внимание

Режим немедленного завершения работы (immediate) эквивалентен краху сервера и приведёт, таким образом, к потере всех не сохранённых асинхронных транзакций.

Асинхронное подтверждение транзакций предоставляет поведение, которое отличается от того, что соответствует установке параметра fsync = off. Настройка `fsync` касается всего сервера и может изменить поведение всех транзакций. Она выключает всю логику внутри Postgres Pro, которая пытается синхронизировать запись отдельных порций в базу данных и, таким образом, крах системы (обусловленный отказом аппаратного обеспечения или операционной системы, который не является сбоем самой СУБД Postgres Pro) может в результате привести к повреждению состояния базы данных. Во многих случаях, асинхронное подтверждение транзакций предоставляет лучшую производительность, чем то, что можно получить выключением `fsync`, но без риска повреждения данных.

commit_delay также выглядит очень похоже на асинхронное подтверждение транзакций, но по сути это является методом синхронного подтверждения транзакций (фактически, во время асинхронных транзакций `commit_delay` игнорируется). `commit_delay` приводит к задержке только перед тем, как синхронная транзакция пытается записать данные WAL на диск, в надежде, что одиночная запись, выполняемая на одну такую транзакцию, сможет также обслужить другие транзакции, которые подтверждаются приблизительно в это же время. Установку этого параметра можно рассматривать как способ увеличения промежутка времени, в течение которого транзакции группируются для единовременной записи на диск. Это распределяет стоимость записи между несколькими транзакциями.

27.5. Настройка WAL

Существует несколько конфигурационных параметров, относящихся к WAL, которые влияют на производительность СУБД. Далее рассказывается об их использовании. Общую информацию об установке параметров конфигурации сервера смотрите в Главе 18.

Контрольные точки— это точки в последовательности транзакций, в которых гарантируется, что файлы с данными и индексами были обновлены всей информацией, записанной перед контрольной точкой. Во время выполнения контрольной точки все «грязные» страницы данных, находящиеся в памяти, сбрасываются на диск, а в файле WAL создаётся специальная запись контрольной точки. (Сделанные изменения были перед этим записаны в файлы WAL.) В случае сбоя процедура восстановления ищет последнюю запись контрольной точки, чтобы определить эту точку в WAL (называемую записью REDO), от которой процедура должна начать операцию воспроизведения изменений. Любые изменения файлов данных до этой точки уже гарантированно находятся на диске. Таким образом, после контрольной точки сегменты WAL, которые предшествуют записи REDO, больше не нужны и могут быть удалены или направлены в циклическую перезапись. (Когда выполняется архивирование WAL, сегменты журнала должны быть заархивированы перед циклической перезаписью или удалением.)

Запись всех «грязных» страниц данных из памяти на диск, которая требуется для контрольной точки, может вызвать значительную нагрузку на дисковый ввод-вывод. По этой причине, активность записи по контрольной точке регулируется так, что ввод-вывод начинается при старте контрольной точки и завершается перед стартом следующей контрольной точки; это минимизирует потерю производительности во время прохождения контрольных точек.

Отдельный серверный процесс контрольных точек автоматически выполняет контрольные точки с заданной частотой. Контрольные точки производятся каждые checkpoint_timeout секунд либо при приближении к пределу max_wal_size, если это имеет место раньше. Значения по умолчанию: 5 минут и 1 Гбайт, соответственно. Если после предыдущей контрольной точки новые записи WAL не добавились, следующие контрольные точки будут пропущены, даже если проходит время `checkpoint_timeout`. (Если вы применяете архивацию WAL и хотите установить нижний предел для частоты архивации, чтобы ограничить потенциальную потерю данных, вам следует настраивать параметр archive_timeout, а не параметры контрольных точек.) Также можно выполнить контрольную точку принудительно, воспользовавшись SQL-командой `CHECKPOINT`.

Уменьшение значений `checkpoint_timeout` и/или `max_wal_size` приводит к учащению контрольных точек. Это позволяет ускорить восстановление после сбоя (поскольку для воспроизведения нужно меньше данных), но с другой стороны нужно учитывать дополнительную нагрузку, возникающую вследствие более частого сброса «грязных» страниц данных на диск. Если включён режим full_page_writes (по умолчанию это так), нужно учесть и ещё один фактор. Для обеспечения целостности страницы данных при первом изменении страницы данных после контрольной точки эта страница записывается в журнал целиком. В данном случае чем меньше интервал между контрольными точками, тем больше объём записи в WAL, так что это частично дискредитирует идею уменьшения интервала записи и в любом случае приводит к увеличению объёма обмена с диском.

Контрольные точки довольно дороги с точки зрения ресурсов: во-первых, потому что они требуют записи всех «грязных» буферов из памяти на диск, и во-вторых, потому что они создают дополнительный трафик WAL, о чём говорилось выше. Таким образом, будет благоразумным установить параметры контрольных точек так, чтобы они не выполнялись слишком часто. Для простой проверки параметров контрольной точки можно установить параметр checkpoint_warning. Если промежуток времени между контрольными точками будет меньше, чем количество секунд, заданное в `checkpoint_warning`, то в журнал сервера будет выдано сообщение с рекомендацией увеличить `max_wal_size`. Эпизодическое появление такого сообщения не является поводом для беспокойства. Но если оно появляется часто, необходимо увеличить значения параметров управления контрольными точками. Массовые операции, такие как `COPY` с большим объёмом данных, могут привести к появлению нескольких таких предупреждений, если значение `max_wal_size` не установлено достаточно большим.

Чтобы избежать «заваливания» системы ввода-вывода при резкой интенсивной записи страниц, запись «грязных» буферов во время контрольной точки растягивается на определённый период времени. Этот период управляется параметром checkpoint_completion_target, который задаётся как часть интервала между контрольными точками (настраиваемого параметром `checkpoint_timeout`). Скорость ввода-вывода подстраивается так, чтобы контрольная точка завершилась к моменту истечения заданной части от `checkpoint_timeout` секунд или до превышения `max_wal_size`, если оно имеет место раньше. Со значением 0.9, заданным по умолчанию, можно ожидать, что Postgres Pro завершит процедуру контрольной точки незадолго до следующей запланированной (примерно на 90% выполнения предыдущей контрольной точки). При этом процесс ввода-вывода растягивается насколько возможно, чтобы обеспечить равномерную нагрузку ввода-вывода в течение интервала между контрольными точками. Но с другой стороны, растягивание контрольных точек влияет на время восстановления, так как для восстановления нужно будет задействовать большее количество сегментов WAL. Если вы хотите оптимизировать длительность восстановления, вы можете уменьшить `checkpoint_timeout`, чтобы контрольные точки создавались чаще, а ввод-вывод при этом всё же растягивался на интервал между ними. Либо же можно уменьшить значение `checkpoint_completion_target`, но это не рекомендуется, поскольку в результате в определённые периоды ввод-вывод будет более интенсивным (во время создания контрольной точки) или менее интенсивным (по завершении контрольной точки, но до следующей запланированной контрольной точки). Хотя в `checkpoint_completion_target` можно задать значение вплоть до 1.0, лучше выбрать значение меньше (по крайней мере, не больше 0.9), так как при контрольных точках выполняются и некоторые другие операции, помимо записи «грязных» буферов. Со значением 1.0 контрольные точки, скорее всего, не будут завершаться вовремя, что приведёт к потере производительности из-за неожиданных колебаний требуемого количества сегментов WAL.

На платформах Linux и POSIX параметр checkpoint_flush_after позволяет принудительно сбрасывать на диск страницы ОС, записываемые во время контрольной точки, при накоплении заданного количества байт. Если его не настроить, эти страницы могут оставаться в кеше страниц ОС, что повлечёт затормаживание при выполнении `fsync` в конце контрольной точки. Этот параметр часто помогает уменьшить задержки транзакций, но может оказать и негативное влияние на производительность; особенно, когда объём нагрузки больше shared_buffers, но меньше кеша страниц в ОС.

Число файлов сегментов WAL в каталоге `pg_wal` зависит от `min_wal_size`, `max_wal_size` и объёма WAL, сгенерированного в предыдущих циклах контрольных точек. Когда старые файлы сегментов оказываются не нужны, они удаляются или перерабатываются (то есть переименовываются, чтобы стать будущими сегментами в нумерованной последовательности). Если вследствие кратковременного скачка интенсивности записи в WAL предел `max_wal_size` превышается, ненужные файлы сегментов будут удаляться, пока система не опустится ниже этого предела. Оставаясь ниже этого предела, система перерабатывает столько файлов WAL, сколько необходимо для покрытия ожидаемой потребности до следующей контрольной точки, и удаляет остальные. Эта оценка базируется на скользящем среднем числа файлов WAL, задействованных в предыдущих циклах контрольных точек. Скользящее среднее увеличивается немедленно, если фактическое использование превышает оценку, так что в нём в некоторой степени накапливается пиковое использование, а не среднее. Значение `min_wal_size` ограничивает снизу число файлов WAL, которые будут переработаны для будущего использования; такой объём WAL всегда будет перерабатываться, даже если система простаивает и оценка использования говорит, что нужен совсем небольшой WAL.

Вне зависимости от `max_wal_size`, последние файлы WAL в объёме wal_keep_size мегабайт и ещё один дополнительный файл WAL сохраняются в любом случае. Кроме того, если применяется архивация WAL, старые сегменты не могут быть удалены или переработаны, пока они не будут заархивированы. Если WAL архивируется медленнее, чем генерируется, либо если команда `archive_library` или библиотека `archive_library` постоянно даёт сбои, старые файлы WAL будут накапливаться в `pg_wal`, пока ситуация не будет разрешена. Медленно работающий или отказавший ведомый сервер, использующий слот репликации, даст тот же эффект (см. Подраздел 25.2.6). Аналогично, если включено создание сводок WAL, старые сегменты сохраняются до тех пор, пока не будут обобщены.

В режиме восстановления архива или горячего резерва сервер периодически выполняет *точки перезапуска*, которые похожи на контрольные точки в обычном режиме работы: сервер принудительно сбрасывает своё состояние на диск, обновляет файл `pg_control`, чтобы показать, что уже обработанные данные WAL не нужно сканировать снова, и затем перерабатывает все старые файлы сегментов журнала в каталоге `pg_wal`. Точки перезапуска не могут выполняться чаще, чем контрольные точки на ведущем сервере, так как они выполняются только в записях контрольных точек. Точка перезапуска может выполняться по расписанию или по внешнему запросу. Счётчик `restartpoints_timed` в представлении `pg_stat_checkpointer` подсчитывает точки, выполняемые по расписанию, а счётчик `restartpoints_req` — точки, выполняемые по запросу. Точка перезапуска по расписанию выполняется, когда достигается запись контрольной точки и после предыдущей точки перезапуска прошло не меньше checkpoint_timeout секунд или при неудавшейся предыдущей попытке выполнить точку перезапуска. В последнем случае следующая точка перезапуска по расписанию выполняется через 15 секунд. Точка перезапуска по запросу начинает выполняться по тем же причинам, что и контрольная точка, но в основном, если размер WAL может превысить max_wal_size. Однако из-за того, что на время выполнения точек перезапуска накладываются ограничения, `max_wal_size` часто превышается при восстановлении вплоть до объёма WAL, записываемого в цикле между контрольными точками. (Значение `max_wal_size` не является строгим ограничением, поэтому рекомендуется иметь достаточно объёмный диск, чтобы не остаться без свободного места.) Счётчик `restartpoints_done` представления `pg_stat_checkpointer` подсчитывает количество действительно выполненных точек перезапуска.

В некоторых случаях, когда размер WAL на ведущем сервере быстро увеличивается, например при добавлении большого числа строк командой `INSERT`, счётчик точек перезапуска по запросу `restartpoints_req` на резервном сервере может показывать взрывной рост. Это происходит, потому что запросы на создание новой точки перезапуска из-за разрастания WAL не могут быть выполнены, так как безопасная запись контрольной точки с последней точки перезапуска ещё не была воспроизведена на резервном сервере. Это нормальное поведение, которое не приводит к повышенному потреблению ресурсов системы. Из всех связанных с точками перезапуска счётчиков на существенное потребление ресурсов системы указывает только `restartpoints_done`.

Наиболее часто используются две связанные с WAL внутренние функции: `XLogInsertRecord` и `XLogFlush`. `XLogInsertRecord` применяется для добавления записи в буферы WAL в разделяемой памяти. Если места для новой записи недостаточно, `XLogInsertRecord` придётся записать (переместить в кеш ядра) несколько заполненных буферов WAL. Это нежелательно, так как `XLogInsertRecord` используется при каждом изменении в базе данных на низком уровне (например, при добавлении строки) в момент, когда установлена исключительная блокировка задействованных страниц данных, поэтому данная операция должна быть максимально быстрой. Что ещё хуже, запись буферов WAL может также повлечь создание нового сегмента WAL, что займёт ещё больше времени. Обычно буферы WAL должны записываться и сохраняться на диске в функции `XLogFlush`, которая вызывается, по большей части, при фиксировании транзакции, чтобы результаты транзакции сохранились в надёжном хранилище. В системах с интенсивной записью в WAL вызовы `XLogFlush` могут иметь место не так часто, чтобы `XLogInsertRecord` не приходилось производить запись. В таких системах следует увеличить число буферов WAL, изменив параметр wal_buffers. Когда включён режим full_page_writes и система очень сильно загружена, увеличение `wal_buffers` поможет сгладить скачки во времени ответа в период сразу после каждой контрольной точки.

Параметр commit_delay определяет, на сколько микросекунд будет засыпать ведущий процесс группы, записывающий в журнал, после получения блокировки в `XLogFlush`, пока подчинённые формируют очередь на запись. Во время этой задержки другие серверные процессы смогут добавлять записи в WAL буферы журнала, чтобы все эти записи сохранились на диск в результате одной операции синхронизации, которую выполнит ведущий. Ведущий процесс не засыпает, если отключён режим fsync, либо число сеансов с активными транзакциями меньше commit_siblings, так как маловероятно, что какой-либо другой сеанс зафиксирует транзакцию в ближайшее время. Заметьте, что на некоторых платформах, разрешение этого таймера сна составляет 10 миллисекунд, так что любое значение параметра `commit_delay` от 1 до 10000 микросекунд будет действовать одинаково. Кроме того, в некоторых системах состояние сна может продлиться несколько дольше, чем требует параметр.

Так как цель `commit_delay` состоит в том, чтобы позволить стоимости каждой операции синхронизации амортизироваться через параллельную фиксацию транзакций (потенциально за счёт задержки транзакции), необходимо определить количество той стоимости, прежде чем урегулирование сможет быть выбрано разумно. Чем выше стоимость, тем более эффективный будет `commit_delay` в увеличении пропускной способности транзакций в какой-то степени. Программа pg_test_fsync может использоваться, чтобы измерить среднее время в микросекундах, которое занимает одиночная работа сброса WAL на диск. Значение половины среднего времени сообщаемого программой рекомендуется в качестве отправной точки для использования значения в параметре `commit_delay` при оптимизации для конкретного объёма работы, и говорит о том, сколько нужно времени для синхронизации сброса единственной операции записи 8 Кбайт. Настройка параметра `commit_delay` особенно полезна в случае хранения WAL в хранилище с высокоскоростными дисками, такими как твердотельные накопители (SSD) или RAID-массивы с кешем записи и аварийным питанием на батарее; но это определённо должно тестироваться на репрезентативной рабочей нагрузке. Более высокие значения `commit_siblings` должны использоваться в таких случаях, тогда как меньшие значения `commit_siblings` часто полезны на носителях с большими задержками. Обратите внимание на то, что увеличение значения параметра `commit_delay` может увеличить задержку транзакции настолько, что пострадает общая производительность транзакций.

Даже если `commit_delay` равен нулю (значение по умолчанию), групповая фиксация всё равно может произойти, но группа будет состоять только из тех сеансов, которым понадобилось сбросить записи о фиксации на диск за то время, пока происходил предыдущий сброс. Чем больше сеансов, тем чаще это происходит даже при нулевом `commit_delay`, поэтому увеличение этого параметра может и не оказать заметного действия. Установка `commit_delay` имеет смысл в двух случаях: (1) когда несколько транзакций одновременно фиксируют изменения, (2) либо когда частота фиксаций ограничена пропускной способностью дисковой подсистемы. Однако при задержке из-за низкой скорости вращения диска эта настройка может оказаться полезной даже всего при двух сеансах.

Параметр wal_sync_method определяет, как Postgres Pro будет обращаться к ядру, чтобы принудительно сохранить WAL на диск. Все методы должны быть одинаковыми в плане надёжности, за исключением `fsync_writethrough`, который может иногда принудительно сбрасывать кеш диска, даже если другие методы не делают этого. Однако какой из них самый быстрый, во многом определяется платформой; вы можете протестировать скорость, используя модуль pg_test_fsync. Обратите внимание, что данный параметр не имеет значения, если `fsync` выключен.

Включение параметра конфигурации wal_debug (предоставляется, если Postgres Pro был скомпилирован с его поддержкой) приводит к тому, что все вызовы связанных с WAL функций `XLogInsertRecord` и `XLogFlush` протоколируются в журнале сервера. В будущем данный параметр может быть заменён более общим механизмом.

Для записи данных WAL на диск предназначены две внутренние функции: `XLogWrite` и `issue_xlog_fsync`. Когда параметр track_wal_io_timing включён, общее время, которое затратила функция `XLogWrite` на запись WAL на диск и которое затратила функция `issue_xlog_fsync` на синхронизацию с ФС, подсчитывается в pg_stat_wal как `wal_write_time` и `wal_sync_time` соответственно. Функцию `XLogWrite` обычно вызывают `XLogInsertRecord` (когда в буферах WAL нет места для новых записей), `XLogFlush` и процесс записи WAL, чтобы записать данные из буферов WAL на диск, а затем вызвать функцию `issue_xlog_fsync`. Функцию `issue_xlog_fsync` обычно вызывает `XLogWrite` для синхронизации файлов WAL. Если параметр `wal_sync_method` принимает значение `open_datasync` или `open_sync`, операция записи `XLogWrite` обеспечивает синхронизацию данных WAL, а `issue_xlog_fsync` не делает ничего. Если же `wal_sync_method` принимает значение `fdatasync`, `fsync` или `fsync_writethrough`, операция записи перемещает буферы WAL в кеш ядра, а `issue_xlog_fsync` синхронизирует с ФС. Вне зависимости от значения `track_wal_io_timing`, в полях `wal_write` и `wal_sync` представления `pg_stat_wal` отображается, сколько раз вызывалась функция `XLogWrite` для записи WAL и сколько раз вызывалась функция `issue_xlog_fsync` для синхронизации с ФС, соответственно.

Параметр recovery_prefetch позволяет уменьшить время ожидания ввода-вывода во время восстановления, указывая ядру начинать чтение дисковых блоков, которые ещё не в пуле буферов Postgres Pro, но вскоре потребуются. Количество параллельных потоков и объём предвыборки ограничиваются параметрами maintenance_io_concurrency и wal_decode_buffer_size соответственно. По умолчанию параметр `recovery_prefetch` имеет значение `try`, при котором эта функциональность включается в системах, поддерживающих `posix_fadvise`.

27.6. Внутреннее устройство WAL

WAL включается автоматически; от администратора не требуется никаких действий за исключением того, чтобы убедиться, что выполнены требования файлов WAL к месту на диске, и что выполнены все необходимые действия по тонкой настройке (см. Раздел 27.5).

Записи WAL добавляются в файлы WAL по мере поступления. Позицию добавления в журнал определяет значение LSN (Log Sequence Number, Последовательный номер в журнале), представляющее собой смещение в байтах внутри WAL, монотонно увеличивающееся с каждой новой записью. Значения LSN возвращаются с типом данных `pg_lsn`. Сравнивая эти значения, можно вычислить объём данных WAL между ними, так что они могут быть полезны для вычисления прогресса при репликации и восстановлении.

Файлы WAL хранятся в виде набора файлов сегментов в каталоге `pg_wal`, находящемся в каталоге данных. Эти файлы обычно имеют размер 16 Мбайт каждый (его можно изменить, передав initdb другое значение в `--wal-segsize`). Каждый файл сегмента разделяется на страницы, обычно по 8 Кбайт. Содержимое записи журнала зависит от типа события, которое сохраняется в журнале. Файлы сегментов имеют имена-номера, которые начинаются с `000000010000000000000001` и последовательно увеличиваются. Зацикливание этих номеров не предусмотрено, но для использования всех доступных номеров потребуется очень много времени.

Имеет смысл размещать WAL на другом диске, отличном от того, где находятся основные файлы базы данных. Для этого можно переместить каталог `pg_wal` в другое место (разумеется, когда сервер остановлен) и создать символическую ссылку из исходного места на перемещённый каталог.

Для WAL важно, чтобы запись в журнал выполнялась до изменений данных в базе. Но этот порядок могут нарушить дисковые устройства, которые ложно сообщают ядру об успешном завершении записи, хотя фактически они только выполнили кеширование данных и пока не сохранили их на диск. Сбой питания в такой ситуации может привести к неисправимому повреждению данных. Администраторы должны убедиться, что диски, где хранятся файлы WAL Postgres Pro, не выдают таких ложных сообщений ядру. (См. Раздел 27.1.)

После выполнения контрольной точки и сброса WAL позиция контрольной точки сохраняется в файл `pg_control`. Таким образом, при старте восстановления сервер сперва читает файл `pg_control` и затем запись контрольной точки; затем он выполняет операцию REDO, сканируя вперёд от позиции в WAL, обозначенной в записи контрольной точки. Поскольку полное содержимое страниц данных сохраняется в WAL в первой странице после контрольной точки (предполагается, что включён режим full_page_writes), все страницы, изменённые с момента контрольной точки, будут восстановлены в целостном состоянии.

В случае, если файл `pg_control` повреждён, мы должны поддерживать возможность сканирования существующих сегментов WAL в обратном порядке — от новых к старым — чтобы найти последнюю контрольную точку. Это пока не реализовано. `pg_control` является достаточно маленьким файлом (меньше, чем одна дисковая страница), который не должен попадать под проблему частичной записи, и на момент написания данной документации не было ни одного сообщения о сбоях СУБД исключительно из-за невозможности чтения самого файла `pg_control`. Таким образом, хотя теоретически это является слабым местом, на практике проблем с `pg_control` не обнаружено.

Глава 28. Логическая репликация

Логическая репликация — это метод репликации объектов данных и изменений в них, использующий репликационные идентификаторы (обычно это первичный ключ). Мы называем такую репликацию «логической», в отличие от физической, которая построена на точных адресах блоков и побайтовом копировании. Postgres Pro поддерживает оба механизма одновременно; см. Главу 25. Логическая репликация позволяет более детально управлять репликацией данных и аспектами безопасности.

В логической репликации используется модель *публикаций/подписок* с одним или несколькими *подписчиками*, которые подписываются на одну или несколько *публикаций* на *публикующем* узле. Подписчики получают данные из публикаций, на которые они подписаны, и могут затем повторно опубликовать данные для организации каскадной репликации или более сложных конфигураций.

Логическая репликация таблицы обычно начинается с создания снимка данных в публикуемой базе данных и копирования её подписчику. После этого изменения на стороне публикации передаются подписчику в реальном времени, когда они происходят. Подписчик применяет изменения в том же порядке, что и узел публикации, так что для публикаций в рамках одной подписки гарантируется транзакционная целостность. Этот метод репликации данных иногда называется транзакционной репликацией.

Типичные сценарии использования логической репликации:

- Передача подписчикам инкрементальных изменений в одной базе данных или подмножестве базы данных, когда они происходят.
- Срабатывание триггеров для отдельных изменений, когда их получает подписчик.
- Объединение нескольких баз данных в одну (например, для целей анализа).
- Репликация между разными основными версиями Postgres Pro.
- Репликация между экземплярами Postgres Pro на разных платформах (например, с Linux на Windows)
- Предоставление доступа к реплицированным данным другим группам пользователей.
- Разделение подмножества базы данных между несколькими базами данных.

База данных подписчика функционирует так же, как и любой другой экземпляр базы Postgres Pro, и может стать публикующей, если создать публикации в ней. Когда подписчик действует как исключительно читающее приложение, никаких конфликтов с одной подпиской не будет. Но они могут возникнуть, если в тот же набор таблиц производят запись какие-либо приложения или другие подписчики.

28.1. Публикация

Публикация может быть определена на любом ведущем сервере физической репликации. Сервер, на котором определяется публикация, называется *публикующим*. Публикация — это набор изменений, выделяемых в таблице или в группе таблиц (он также может называться набором репликации). Публикация существует только в одной базе данных.

Публикации отличаются от схем и они никак не влияют на доступ к таблице. Если требуется, каждую таблицу можно включить в несколько публикаций. В настоящее время публикации могут содержать только таблицы, при этом они могут содержать все таблицы в схеме. Объекты в них нужно добавлять явным образом, если только публикация не создана для всех таблиц (с указанием `ALL TABLES`).

Публикации могут ограничивать набор публикуемых изменений, выбирая любое сочетание операций из `INSERT`, `UPDATE`, `DELETE` и `TRUNCATE`, подобно тому как для разных типов событий могут срабатывать триггеры. По умолчанию реплицируются все типы операций. Эти ограничения публикаций распространяются только на операции DML, они не влияют на копирование при начальной синхронизации данных. (Фильтры строк не работают для операции `TRUNCATE`. См. Раздел 28.4).

Чтобы можно было реплицировать операции `UPDATE` и `DELETE`, в публикуемой таблице должен быть настроен *репликационный идентификатор*, чтобы идентифицировать соответствующие строки для изменения или удаления на стороне подписчика. По умолчанию это первичный ключ, если он создан. Также репликационным идентификатором можно назначить другой уникальный индекс (с некоторыми дополнительными условиями). Если в таблице нет подходящего ключа, в качестве репликационного идентификатора можно задать `FULL`, что будет означать, что ключом будет вся строка. Если выбран репликационный идентификатор `FULL`, для поиска строк на стороне подписчика могут использоваться индексы. Подходящими индексами будут нечастичные индексы-B-деревья или хеш-индексы, самое левое поле которых является столбцом (а не выражением), ссылающимся на столбец опубликованной таблицы. Эти ограничения на свойства неуникальных индексов соответствуют некоторым ограничениям, которые применяются к первичным ключам. Если нет подходящих индексов, поиск на стороне подписчика может быть очень неэффективным, поэтому репликационный идентификатор `FULL` следует использовать только в крайнем случае, если нет другого решения. Если на стороне публикации выбран репликационный идентификатор, отличный от `FULL`, то идентификатор, состоящий из того же или меньшего количества столбцов, также должен быть определён на стороне подписчика. Подробнее о назначении репликационного идентификатора рассказывается в `REPLICA IDENTITY`. Если в публикацию, в которой реплицируются операции `UPDATE` и `DELETE`, добавляется таблица без репликационного идентификатора, то последующие команды `UPDATE` и `DELETE` на стороне публикации вызовут ошибку. Команды `INSERT` могут выполняться вне зависимости от такого идентификатора.

У каждой публикации может быть множество подписчиков.

Публикация создаётся командой `CREATE PUBLICATION` и может быть впоследствии изменена или удалена с помощью соответствующих команд.

В публикации можно динамически добавлять или удалять отдельные таблицы, используя команду `ALTER PUBLICATION`. Операции `ADD TABLE` и `DROP TABLE` являются транзакционными, так что репликация таблицы будет начата или закончена с определённым снимком только после фиксации транзакции.

28.2. Подписка

Подписка — это принимающая сторона логической репликации. Узел, на котором определяется подписка, называется *подписчиком*. В свойствах подписки определяется подключение к другой базе данных и набор публикаций (из одной или нескольких), на которые подписчик хочет подписаться.

База данных подписчика работает так же, как и экземпляр любой другой базы Postgres Pro, и может быть публикующей для других баз, если в ней определены собственные подписки.

Узел подписчика может подписываться на несколько подписок, если требуется. В одной паре публикующий сервер/подписчик могут быть определены несколько подписок, но при этом нужно позаботиться о том, чтобы публикуемые объекты в разных подписках не перекрывались.

Изменения в каждой подписке будут приходить через один слот репликации (см. Подраздел 25.2.6). Дополнительные слоты репликации могут потребоваться для начальной синхронизации данных, уже существующих в таблицах, но будут удалены в конце синхронизации.

Подписка логической репликации может представлять собой ведомый узел для синхронной репликации (см. Подраздел 25.2.8). В этом случае именем ведомого узла по умолчанию будет имя подписки. Другое имя можно задать в свойстве `application_name` в строке подключения для данной подписки.

Подписки могут быть выгружены командой `pg_dump`, если её выполняет суперпользователь. В противном случае выдаётся предупреждение и подписки пропускаются, так как обычным пользователям не разрешено читать всю информацию о подписках из каталога `pg_subscription`.

Подписки добавляются командой `CREATE SUBSCRIPTION` и могут быть остановлены/возобновлены в любой момент командой `ALTER SUBSCRIPTION`, а также удалены командой `DROP SUBSCRIPTION`.

Когда подписка удаляется и пересоздаётся, информация о синхронизации теряется. Это означает, что после этого данные необходимо синхронизировать заново.

Определения схемы не реплицируются, а публикуемые таблицы должны существовать в базе подписчика. Объектами репликации могут быть только обычные таблицы. Так, например, нельзя произвести репликацию в представление.

Таблицы публикации сопоставляются с таблицами подписчика по полностью заданным именам таблиц. Репликация в таблицы с другими именами на стороне подписчика не поддерживается.

Столбцы таблиц также сопоставляются по именам. Порядок столбцов в таблице подписчика может отличаться от порядка столбцов в публикации. Также могут не совпадать типы столбцов; достаточно только возможности преобразования текстового представления данных в целевой тип. Например, данные столбца типа `integer` могут реплицироваться в столбец типа `bigint`. Целевая таблица может также содержать дополнительные столбцы, отсутствующие в публикуемой таблице. Такие столбцы будут заполнены значениями по умолчанию, заданными в определении целевой таблицы. Однако логическая репликация в двоичном формате имеет более строгие ограничения. За подробностями обратитесь к описанию параметра `binary` команды `CREATE SUBSCRIPTION`.

28.2.1. Управление слотами репликации

Как упоминалось ранее, каждая (активная) подписка получает изменения из слота репликации на удалённой (публикующей) стороне.

Дополнительные слоты синхронизации таблиц обычно существуют временно, создаются только для выполнения начальной синхронизации таблиц и автоматически удаляются, когда становятся не нужны. Эти слоты синхронизации таблиц называются так: «`pg_%u_sync_%u_%llu`» (параметры: *`oid`* подписки, *`relid`* таблицы, системный идентификатор *`sysid`*).

Обычно расположенный удалённо слот репликации автоматически создаётся при создании подписки командой `CREATE SUBSCRIPTION` подписка и автоматически удаляется при удалении подписки командой `DROP SUBSCRIPTION`. Однако в некоторых ситуациях может быть полезно или даже необходимо манипулировать подпиской и нижележащим слотом по отдельности. Например, возможны такие сценарии:

- При создании подписки слот репликации может уже существовать. В этом случае подписку можно создать с параметром `create_slot = false`, чтобы она была связана с существующим слотом.
- При создании подписки расположенный удалённо узел может быть недоступен или находиться в нерабочем состоянии. В этом случае подписку можно создать с указанием `connect = false`. При этом подключение к расположенному удалённо узлу не будет устанавливаться. Этот вариант использует pg_dump. Чтобы активировать такую подписку впоследствии, расположенный удалённо слот репликации нужно будет создать вручную.
- При ликвидации публикации может потребоваться сохранить слот репликации. Например, это полезно, когда нужно перенести базу данных подписчика на другой узел и активировать её там. В этом случае разорвите связь подписки со слотом, используя команду `ALTER SUBSCRIPTION`, прежде чем удалять подписку.
- При ликвидации подписки расположенный удалённо узел может быть недоступен. В этом случае разорвите связь подписки со слотом, используя команду `ALTER SUBSCRIPTION`, прежде чем пытаться удалять подписку. Если расположенный удалённо экземпляр базы данных прекратил существование, больше никакие действия не требуются. Если же такой экземпляр базы данных просто оказался недоступным, слот репликации (и все оставшиеся слоты синхронизации таблиц) нужно будет удалить вручную; в противном случае публикующий сервер(ы) продолжит сохранять WAL и может в конце концов заполнить всё место на диске. Такие случаи заслуживают самого серьёзного разбирательства.

28.2.2. Примеры: настройка логической репликации

Создайте несколько тестовых таблиц на публикующем сервере.

```
test_pub=# CREATE TABLE t1(a int, b text, PRIMARY KEY(a));
CREATE TABLE
test_pub=# CREATE TABLE t2(c int, d text, PRIMARY KEY(c));
CREATE TABLE
test_pub=# CREATE TABLE t3(e int, f text, PRIMARY KEY(e));
CREATE TABLE
```

Создайте такие же таблицы на стороне подписчика.

```
test_sub=# CREATE TABLE t1(a int, b text, PRIMARY KEY(a));
CREATE TABLE
test_sub=# CREATE TABLE t2(c int, d text, PRIMARY KEY(c));
CREATE TABLE
test_sub=# CREATE TABLE t3(e int, f text, PRIMARY KEY(e));
CREATE TABLE
```

Вставьте данные в таблицы на стороне публикующего сервера.

```
test_pub=# INSERT INTO t1 VALUES (1, 'one'), (2, 'two'), (3, 'three');
INSERT 0 3
test_pub=# INSERT INTO t2 VALUES (1, 'A'), (2, 'B'), (3, 'C');
INSERT 0 3
test_pub=# INSERT INTO t3 VALUES (1, 'i'), (2, 'ii'), (3, 'iii');
INSERT 0 3
```

Создайте публикации для таблиц. Для публикаций `pub2` и `pub3a` параметром `publish` отключаются некоторые операции, а публикация `pub3b` создаётся с фильтром строк (см. Раздел 28.4).

```
test_pub=# CREATE PUBLICATION pub1 FOR TABLE t1;
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION pub2 FOR TABLE t2 WITH (publish = 'truncate');
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION pub3a FOR TABLE t3 WITH (publish = 'truncate');
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION pub3b FOR TABLE t3 WHERE (e > 5);
CREATE PUBLICATION
```

Создайте подписки на публикации. Подписка `sub3` оформляется сразу на две публикации, `pub3a` и `pub3b`. Для всех подписок будут по умолчанию скопированы начальные данные.

```
test_sub=# CREATE SUBSCRIPTION sub1
test_sub-# CONNECTION 'host=localhost dbname=test_pub application_name=sub1'
test_sub-# PUBLICATION pub1;
CREATE SUBSCRIPTION
test_sub=# CREATE SUBSCRIPTION sub2
test_sub-# CONNECTION 'host=localhost dbname=test_pub application_name=sub2'
test_sub-# PUBLICATION pub2;
CREATE SUBSCRIPTION
test_sub=# CREATE SUBSCRIPTION sub3
test_sub-# CONNECTION 'host=localhost dbname=test_pub application_name=sub3'
test_sub-# PUBLICATION pub3a, pub3b;
CREATE SUBSCRIPTION
```

Вы можете убедиться в том, что начальные данные таблицы копируются независимо от того, какие операции исключены параметром `publish`.

```
test_sub=# SELECT * FROM t1;
```

```
 a |   b
---+-------
 1 | one
 2 | two
 3 | three
(3 rows)

test_sub=# SELECT * FROM t2;
 c | d
---+---
 1 | A
 2 | B
 3 | C
(3 rows)
```

Кроме того, поскольку при копировании исходных данных игнорируется исключение операций, определяемое в `publish`, и у публикации `pub3a` нет фильтра строк, скопированная таблица `t3` содержит все строки, при том что они не соответствуют фильтру строк публикации `pub3b`.

```
test_sub=# SELECT * FROM t3;
 e |  f
---+-----
 1 | i
 2 | ii
 3 | iii
(3 rows)
```

Вставьте дополнительные данные в таблицы на стороне публикации.

```
test_pub=# INSERT INTO t1 VALUES (4, 'four'), (5, 'five'), (6, 'six');
INSERT 0 3
test_pub=# INSERT INTO t2 VALUES (4, 'D'), (5, 'E'), (6, 'F');
INSERT 0 3
test_pub=# INSERT INTO t3 VALUES (4, 'iv'), (5, 'v'), (6, 'vi');
INSERT 0 3
```

Теперь на стороне публикации данные выглядят так:

```
test_pub=# SELECT * FROM t1;
 a |   b
---+-------
 1 | one
 2 | two
 3 | three
 4 | four
 5 | five
 6 | six
(6 rows)

test_pub=# SELECT * FROM t2;
 c | d
---+---
 1 | A
 2 | B
 3 | C
 4 | D
 5 | E
 6 | F
(6 rows)
```

```
test_pub=# SELECT * FROM t3;
 e |  f
---+-----
 1 | i
 2 | ii
 3 | iii
 4 | iv
 5 | v
 6 | vi
(6 rows)
```

Вы можете заметить, что в процессе обычной репликации учитывается список операций, определённый в `publish`. Вследствие этого через публикации `pub2` и `pub3a` не реплицируются операции `INSERT`. Кроме того, для публикации `pub3b` реплицируются только данные, соответствующие фильтру строк `pub3b`. Теперь данные на стороне подписчика выглядят так:

```
test_sub=# SELECT * FROM t1;
 a |   b
---+-------
 1 | one
 2 | two
 3 | three
 4 | four
 5 | five
 6 | six
(6 rows)

test_sub=# SELECT * FROM t2;
 c | d
---+---
 1 | A
 2 | B
 3 | C
(3 rows)

test_sub=# SELECT * FROM t3;
 e |  f
---+-----
 1 | i
 2 | ii
 3 | iii
 6 | vi
(4 rows)
```

28.2.3. Примеры: отложенное создание слота репликации

В некоторых случаях (например, Подраздел 28.2.1), если слот репликации не был создан удалённо автоматически, пользователю необходимо создать его вручную, прежде чем подписку можно будет активировать. Шаги по созданию слота и активации подписки показаны в следующих примерах. В этих примерах указан стандартный модуль вывода логического декодирования (`pgoutput`), который используется встроенной логической репликацией.

Сначала создайте публикацию, которая будет использоваться для примеров.

```
test_pub=# CREATE PUBLICATION pub1 FOR ALL TABLES;
CREATE PUBLICATION
```

Пример 1: В подписке указано `connect = false`.

- Создайте подписку.

```
test_sub=# CREATE SUBSCRIPTION sub1
test_sub-# CONNECTION 'host=localhost dbname=test_pub'
test_sub-# PUBLICATION pub1
test_sub-# WITH (connect=false);
ПРЕДУПРЕЖДЕНИЕ:  подписка создана, но не подключена
ПОДСКАЗКА:  Чтобы начать репликацию, вы должны вручную создать слот репликации,
 включить подписку, а затем обновить её.
CREATE SUBSCRIPTION
```

- На публикующем сервере вручную создайте слот. Поскольку имя не было указано при выполнении `CREATE SUBSCRIPTION`, имя создаваемого слота совпадает с именем подписки, например «sub1».

```
test_pub=# SELECT * FROM pg_create_logical_replication_slot('sub1', 'pgoutput');
 slot_name |    lsn
-----------+-----------
 sub1      | 0/19404D0
(1 row)
```

- На подписчике выполните активацию подписки. После этого таблицы `pub1` начнут реплицироваться.

```
test_sub=# ALTER SUBSCRIPTION sub1 ENABLE;
ALTER SUBSCRIPTION
test_sub=# ALTER SUBSCRIPTION sub1 REFRESH PUBLICATION;
ALTER SUBSCRIPTION
```

Пример 2: В подписке указано `connect = false`, но также указан параметр `slot_name`.

- Создайте подписку.

```
test_sub=# CREATE SUBSCRIPTION sub1
test_sub-# CONNECTION 'host=localhost dbname=test_pub'
test_sub-# PUBLICATION pub1
test_sub-# WITH (connect=false, slot_name='myslot');
ПРЕДУПРЕЖДЕНИЕ:  подписка создана, но не подключена
ПОДСКАЗКА:  Чтобы начать репликацию, вы должны вручную создать слот репликации,
 включить подписку, а затем обновить её.
CREATE SUBSCRIPTION
```

- На публикующем сервере вручную создайте слот с тем же именем, которое использовалось при выполнении `CREATE SUBSCRIPTION`, например «myslot».

```
test_pub=# SELECT * FROM pg_create_logical_replication_slot('myslot', 'pgoutput');
 slot_name |    lsn
-----------+-----------
 myslot    | 0/19059A0
(1 row)
```

- На подписчике остальные шаги активации подписки такие же, как указано выше.

```
test_sub=# ALTER SUBSCRIPTION sub1 ENABLE;
ALTER SUBSCRIPTION
test_sub=# ALTER SUBSCRIPTION sub1 REFRESH PUBLICATION;
ALTER SUBSCRIPTION
```

Пример 3: В подписке указано `slot_name = NONE`

- Создайте подписку. Если `slot_name = NONE`, также необходимо указать `enabled = false` и `create_slot = false`.

```
test_sub=# CREATE SUBSCRIPTION sub1
test_sub-# CONNECTION 'host=localhost dbname=test_pub'
```

```
test_sub-# PUBLICATION pub1
test_sub-# WITH (slot_name=NONE, enabled=false, create_slot=false);
CREATE SUBSCRIPTION
```

- На публикующем сервере вручную создайте слот с любым именем, например «myslot».

```
test_pub=# SELECT * FROM pg_create_logical_replication_slot('myslot', 'pgoutput');
 slot_name |    lsn
-----------+-----------
 myslot    | 0/1905930
(1 row)
```

- На подписчике свяжите подписку с только что созданным именем слота.

```
test_sub=# ALTER SUBSCRIPTION sub1 SET (slot_name='myslot');
ALTER SUBSCRIPTION
```

- Остальные шаги активации подписки такие же, как указано выше.

```
test_sub=# ALTER SUBSCRIPTION sub1 ENABLE;
ALTER SUBSCRIPTION
test_sub=# ALTER SUBSCRIPTION sub1 REFRESH PUBLICATION;
ALTER SUBSCRIPTION
```

28.3. Отработка отказа логической репликации

Чтобы подписчики могли продолжить репликацию данных с публикующего узла даже в случае отказа узла, необходим физический резервный сервер, связанный с публикующим узлом. Логические слоты главного сервера, соответствующие подпискам, можно синхронизировать с резервным сервером, указывая при создании подписок параметр `failover = true`. За подробностями обратитесь к Подразделу 47.2.3. Включение параметра `failover` обеспечивает бесшовный переход таких подписок после повышения резерва. Они могут продолжить получать публикации с нового главного сервера.

Поскольку синхронизация слотов логической репликации происходит асинхронно, необходимо удостовериться, что слоты репликации синхронизировались с резервным сервером до начала отработки отказа. Это можно сделать, настроив параметр конфигурации `synchronized_standby_slots`.

Чтобы убедиться в готовности резервного сервера к возможному отказу для определённого подписчика, проверьте, что все слоты логической репликации, требующиеся подписчику, синхронизированы с резервным сервером с помощью этих команд:

1. Чтобы определить, какие слоты логической репликации нужно синхронизировать с резервным сервером, который планируется повысить, выполните приведённый ниже SQL-запрос на подписчике. Этот запрос возвращает слоты, связанные с подписками, на которых включён режим отработки отказа.

```
test_sub=# SELECT
               array_agg(quote_literal(s.subslotname)) AS slots
           FROM  pg_subscription s
           WHERE s.subfailover AND
                 s.subslotname IS NOT NULL;
 slots
-------
 {'sub1','sub2','sub3'}
(1 row)
```

2. Чтобы определить, какие слоты логической репликации нужно синхронизировать с резервным сервером, который планируется повысить, выполните приведённый ниже SQL-запрос на подписчике. Этот запрос необходимо запустить для каждой базы данных, где есть подписки с включённой отработкой отказа. Обратите внимание, что слот синхронизации таблиц необходимо синхронизировать с резервным сервером, только если завершено создание копии табли-

цы (см. Раздел 51.57). В других случаях такую синхронизацию проверять не нужно, поскольку слоты либо будут удалены, либо воссозданы на новом главном сервере.

```
test_sub=# SELECT
               array_agg(quote_literal(slot_name)) AS slots
           FROM
           (
               SELECT CONCAT('pg_', srsubid, '_sync_', srrelid, '_',
 ctl.system_identifier) AS slot_name
               FROM pg_control_system() ctl, pg_subscription_rel r, pg_subscription
 s
               WHERE r.srsubstate = 'f' AND s.oid = r.srsubid AND s.subfailover
           );
 slots
-------
 {'pg_16394_sync_16385_7394666715149055164'}
(1 row)
```

3. Проверьте, что указанные слоты логической репликации существуют на резервном сервере и готовы к отработке отказа.

```
test_standby=# SELECT slot_name, (synced AND NOT temporary AND invalidation_reason
 IS NULL) AS failover_ready
               FROM pg_replication_slots
               WHERE slot_name IN
                   ('sub1','sub2','sub3',
 'pg_16394_sync_16385_7394666715149055164');
  slot_name                                  | failover_ready
--------------------------------------------+----------------
  sub1                                       | t
  sub2                                       | t
  sub3                                       | t
  pg_16394_sync_16385_7394666715149055164   | t
(4 rows)
```

Если слоты присутствуют на резервном сервере и готовы к отработке отказа (запрос выше вернул true для `failover_ready`), то существующие подписки продолжат работать с новым главным сервером.

Первые два шага вышеуказанной процедуры выполняются подписчиком PostgreSQL. Рекомендуется пройти эту инструкцию на каждом узле подписчика, который будет обслуживаться назначенным резервным сервером после переключения, что сформирует полный список слотов репликации. Этот список можно проверить на третьем шаге, чтобы убедиться в готовности аварийного переключения. Подписчики, не относящиеся к PostgreSQL, могут использовать другие методы, чтобы определить, какие слоты репликации используется в их подписках.

В некоторых случаях, например при плановом переключении, необходимо убедиться, что все подписчики, относящиеся и не относящиеся к PostgreSQL, смогут продолжить репликацию после переключения на заданный резервный сервер. В таких случаях используйте следующий SQL-запрос вместо первых двух шагов, чтобы определить, какие слоты репликации на ведущем сервере необходимо синхронизировать с резервным, который будет повышаться. Этот запрос возвращает все необходимые для этого слоты репликации, связанные с подписками, участвующими в переключении.

```
/* primary # */ SELECT array_agg(quote_literal(r.slot_name)) AS slots
               FROM pg_replication_slots r
               WHERE r.failover AND NOT r.temporary;
 slots
-------
 {'sub1','sub2','sub3', 'pg_16394_sync_16385_7394666715149055164'}
(1 row)
```

28.4. Фильтры строк

По умолчанию подписчикам передаются все данные из всех опубликованных таблиц. Однако множество реплицируемых данных можно ограничить, используя *фильтр строк*. Потребность использовать фильтры строк может быть вызвана соображениями безопасности, производительности или требованиями к поведению. Если для опубликованной таблицы устанавливается фильтр строк, строка реплицируется, только если её данные удовлетворяют выражению этого фильтра. Это позволяет организовать частичную репликацию данных набора таблиц. Определяются фильтры строк на уровне отдельных таблиц. Для каждой публикуемой таблицы, данные которой требуется отфильтровать, нужно добавить предложение `WHERE` после её имени, при этом выражение условия `WHERE` нужно заключить в круглые скобки. За подробностями обратитесь к CREATE PUBLICATION.

28.4.1. Правила фильтров строк

Фильтры строк применяются *перед* публикацией изменений. Если результатом выражения фильтра для строки является `false` или `NULL`, эта строка не реплицируется. Выражение условия `WHERE` вычисляется от имени роли, которая используется для подключения репликации (т. е. роли, указанной в предложении `CONNECTION` CREATE SUBSCRIPTION). На команду `TRUNCATE` фильтры строк не распространяются.

28.4.2. Ограничения выражений

В предложении `WHERE` допускаются только простые выражения, в которых нельзя использовать определяемые пользователем функции, операторы, типы и параметры сортировки, а также ссылки на системные столбцы и непостоянные встроенные функции.

Если через публикацию реплицируются операции `UPDATE` или `DELETE`, предложение `WHERE` фильтра строк должно содержать только те столбцы, которые входят в идентификатор реплики (см. `REPLICA IDENTITY`). Если же через публикацию реплицируются только операции `INSERT`, в фильтре строк `WHERE` можно использовать любой столбец.

28.4.3. Преобразования UPDATE

При выполнении каждой операции `UPDATE` выражение фильтра строк вычисляется и для старой, и для новой строки (т. е. проверяются данные до и после изменения). Если оба результата положительные (`true`), изменение `UPDATE` реплицируется. Если оба результата отрицательные (`false`), изменение не реплицируется. Если же выражению фильтра удовлетворяет только одна строка, старая или новая, операция `UPDATE` преобразуется в `INSERT` или `DELETE`, чтобы не нарушилась согласованность данных. Строка на стороне подписки должна соответствовать выражению фильтра строк, определённого на публикующем сервере.

Если старая строка соответствует выражению фильтра строк (она была передана подписчику), а новая строка — нет, то для обеспечения согласованности данных старую строку нужно удалить у подписчика. Таким образом, `UPDATE` преобразуется в `DELETE`.

Если старая строка не соответствует выражению фильтра строк (она не была передана подписчику), а новая строка соответствует, то для обеспечения согласованности данных новую строку нужно добавить в таблицу подписчика. Таким образом, `UPDATE` преобразуется в `INSERT`.

Производимые преобразования показаны в Таблице 28.1.

Таблица 28.1. Краткая схема преобразования `UPDATE`

Старая строка	Новая строка	Преобразование
не соответствует	не соответствует	не реплицировать
не соответствует	соответствует	`INSERT`
соответствует	не соответствует	`DELETE`

Старая строка	**Новая строка**	**Преобразование**
соответствует	соответствует	`UPDATE`

28.4.4. Секционированные таблицы

Если публикация содержит секционированную таблицу, параметр публикации `publish_via_partition_root` определяет, какой фильтр строк использовать. Если `publish_via_partition_root` имеет значение `true`, используется фильтр строк *корневой секционированной таблицы*. В противном случае, если `publish_via_partition_root` имеет значение `false` (по умолчанию), используются фильтры строк каждой *секции*.

28.4.5. Начальная синхронизация данных

Если подписка требует копирования уже существующих данных, а публикация содержит предложения `WHERE`, подписчику копируются только данные, удовлетворяющие выражениям фильтров строк.

Если подписка связана с несколькими публикациями, в которых одна таблица опубликована с разными предложениями `WHERE`, будут скопированы строки, удовлетворяющие *любому* из фильтров. За подробностями обратитесь к Подразделу 28.4.6.

Предупреждение

Так как при начальной синхронизации данных не учитывается параметр `publish`, когда копируются существующие данные таблиц, подписчику могут быть переданы строки, которые не реплицировались бы при выполнении операций DML. Примеры приведены в Подразделе 28.8.1 и Подразделе 28.2.2.

Примечание

Если на стороне подписчика используется версия ниже 15, при копировании существующих данных фильтры строк не обрабатываются, даже если они определены в публикации. Это связано с тем, что старые версии могут копировать только все данные таблицы.

28.4.6. Объединение нескольких фильтров строк

Если подписка связана с несколькими публикациями, в которых одна и та же таблица опубликована с разными фильтрами строк (для одной и той же операции `publish`), эти выражения объединяются вместе, так что реплицироваться будут строки, удовлетворяющие *любому* из выражений. Это означает, что все остальные фильтры строк для той же таблицы становятся избыточными, если:

- У одной из публикаций нет фильтра строк.
- Одна из публикаций была создана с предложением `FOR ALL TABLES`. Это предложение не позволяет использовать фильтры строк.
- Одна из публикаций была создана с предложением `FOR TABLES IN SCHEMA` и целевая таблица принадлежит указанной схеме. Это предложение не позволяет использовать фильтры строк.

28.4.7. Примеры

Создайте несколько таблиц, которые будут использоваться в следующих примерах.

```
test_pub=# CREATE TABLE t1(a int, b int, c text, PRIMARY KEY(a,c));
CREATE TABLE
test_pub=# CREATE TABLE t2(d int, e int, f int, PRIMARY KEY(d));
CREATE TABLE
test_pub=# CREATE TABLE t3(g int, h int, i int, PRIMARY KEY(g));
```

```
CREATE TABLE
```

Создайте несколько публикаций. Публикация `p1` содержит одну таблицу (`t1`), и для неё определяется фильтр строк. Публикация `p2` содержит две таблицы, и фильтр строк определён для `t2`, но отсутствует для `t1`. Публикация `p3` содержит две таблицы, и фильтр строк определён для обеих.

```
test_pub=# CREATE PUBLICATION p1 FOR TABLE t1 WHERE (a > 5 AND c = 'NSW');
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION p2 FOR TABLE t1, t2 WHERE (e = 99);
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION p3 FOR TABLE t2 WHERE (d = 10), t3 WHERE (g = 10);
CREATE PUBLICATION
```

Выражения фильтров строк для каждой публикации можно просмотреть в `psql`.

```
test_pub=# \dRp+
                               Publication p1
  Owner   | All tables | Inserts | Updates | Deletes | Truncates | Via root
----------+------------+---------+---------+---------+-----------+----------
 postgres | f          | t       | t       | t       | t         | f
Tables:
    "public.t1" WHERE ((a > 5) AND (c = 'NSW'::text))

                               Publication p2
  Owner   | All tables | Inserts | Updates | Deletes | Truncates | Via root
----------+------------+---------+---------+---------+-----------+----------
 postgres | f          | t       | t       | t       | t         | f
Tables:
    "public.t1"
    "public.t2" WHERE (e = 99)

                               Publication p3
  Owner   | All tables | Inserts | Updates | Deletes | Truncates | Via root
----------+------------+---------+---------+---------+-----------+----------
 postgres | f          | t       | t       | t       | t         | f
Tables:
    "public.t2" WHERE (d = 10)
    "public.t3" WHERE (g = 10)
```

Также в `psql` можно просмотреть выражения фильтров строк для каждой таблицы. Обратите внимание: таблица `t1` является членом двух публикаций, но имеет фильтр строк только в `p1`, а таблица `t2` является членом двух публикаций и имеет в них разные фильтры строк.

```
test_pub=# \d t1
                 Table "public.t1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 a      | integer |           | not null |
 b      | integer |           |          |
 c      | text    |           | not null |
Indexes:
    "t1_pkey" PRIMARY KEY, btree (a, c)
Publications:
    "p1" WHERE ((a > 5) AND (c = 'NSW'::text))
    "p2"

test_pub=# \d t2
                 Table "public.t2"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
```

```
 d      | integer |           | not null |
 e      | integer |           |          |
 f      | integer |           |          |
Indexes:
    "t2_pkey" PRIMARY KEY, btree (d)
Publications:
    "p2" WHERE (e = 99)
    "p3" WHERE (d = 10)

test_pub=# \d t3
                 Table "public.t3"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 g      | integer |           | not null |
 h      | integer |           |          |
 i      | integer |           |          |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (g)
Publications:
    "p3" WHERE (g = 10)
```

Создайте на сервере подписчика таблицу `t1` с тем же определением, что и на публикующем сервере, а также создайте подписку `s1`, связанную с публикацией `p1`.

```
test_sub=# CREATE TABLE t1(a int, b int, c text, PRIMARY KEY(a,c));
CREATE TABLE
test_sub=# CREATE SUBSCRIPTION s1
test_sub-# CONNECTION 'host=localhost dbname=test_pub application_name=s1'
test_sub-# PUBLICATION p1;
CREATE SUBSCRIPTION
```

Вставьте несколько строк. Реплицироваться будут только строки, соответствующие предложению `t1 WHERE` публикации `p1`.

```
test_pub=# INSERT INTO t1 VALUES (2, 102, 'NSW');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (3, 103, 'QLD');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (4, 104, 'VIC');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (5, 105, 'ACT');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (6, 106, 'NSW');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (7, 107, 'NT');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (8, 108, 'QLD');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (9, 109, 'NSW');
INSERT 0 1

test_pub=# SELECT * FROM t1;
 a |  b  |  c
---+-----+-----
 2 | 102 | NSW
 3 | 103 | QLD
 4 | 104 | VIC
 5 | 105 | ACT
 6 | 106 | NSW
```

```
 7 | 107 | NT
 8 | 108 | QLD
 9 | 109 | NSW
(8 rows)
```

```
test_sub=# SELECT * FROM t1;
 a |  b  |  c
---+-----+-----
 6 | 106 | NSW
 9 | 109 | NSW
(2 rows)
```

Измените некоторые данные, в которых и старое, и новое значения строки соответствуют условию `t1 WHERE` публикации `p1`. Операция `UPDATE` будет реплицировать изменения как обычно.

```
test_pub=# UPDATE t1 SET b = 999 WHERE a = 6;
UPDATE 1

test_pub=# SELECT * FROM t1;
 a |  b  |  c
---+-----+-----
 2 | 102 | NSW
 3 | 103 | QLD
 4 | 104 | VIC
 5 | 105 | ACT
 7 | 107 | NT
 8 | 108 | QLD
 9 | 109 | NSW
 6 | 999 | NSW
(8 rows)
```

```
test_sub=# SELECT * FROM t1;
 a |  b  |  c
---+-----+-----
 9 | 109 | NSW
 6 | 999 | NSW
(2 rows)
```

Измените некоторые данные, в которых старые значения строк не соответствуют условию `t1 WHERE` публикации `p1`, а новые значения строк — соответствуют. Операция `UPDATE` будет преобразована в `INSERT` и реплицируется в таком виде. Посмотрите на новую строку в таблице подписчика.

```
test_pub=# UPDATE t1 SET a = 555 WHERE a = 2;
UPDATE 1

test_pub=# SELECT * FROM t1;
  a  |  b  |  c
-----+-----+-----
   3 | 103 | QLD
   4 | 104 | VIC
   5 | 105 | ACT
   7 | 107 | NT
   8 | 108 | QLD
   9 | 109 | NSW
   6 | 999 | NSW
 555 | 102 | NSW
(8 rows)
```

```
test_sub=# SELECT * FROM t1;
  a  |  b  |  c
-----+-----+-----
```

```
   9 | 109 | NSW
   6 | 999 | NSW
 555 | 102 | NSW
(3 rows)
```

Измените некоторые данные, в которых старые значения строк соответствуют условию `t1 WHERE` публикации `p1`, а новые значения — не соответствуют. Операция `UPDATE` будет преобразована в `DELETE` и реплицируется в таком виде. Обратите внимание, строка была удалена из таблицы подписчика.

```
test_pub=# UPDATE t1 SET c = 'VIC' WHERE a = 9;
UPDATE 1

test_pub=# SELECT * FROM t1;
  a  |  b  |  c
-----+-----+-----
   3 | 103 | QLD
   4 | 104 | VIC
   5 | 105 | ACT
   7 | 107 | NT
   8 | 108 | QLD
   6 | 999 | NSW
 555 | 102 | NSW
   9 | 109 | VIC
(8 rows)
```

```
test_sub=# SELECT * FROM t1;
  a  |  b  |  c
-----+-----+-----
   6 | 999 | NSW
 555 | 102 | NSW
(2 rows)
```

Следующие примеры показывают, как в зависимости от параметра публикации `publish_via_partition_root` для секционированных таблиц используется фильтр строк родительской или дочерней таблицы.

Создайте секционированную таблицу на публикующем сервере.

```
test_pub=# CREATE TABLE parent(a int PRIMARY KEY) PARTITION BY RANGE(a);
CREATE TABLE
test_pub=# CREATE TABLE child PARTITION OF parent DEFAULT;
CREATE TABLE
```

Создайте такую же конфигурацию таблиц на подписчике.

```
test_sub=# CREATE TABLE parent(a int PRIMARY KEY) PARTITION BY RANGE(a);
CREATE TABLE
test_sub=# CREATE TABLE child PARTITION OF parent DEFAULT;
CREATE TABLE
```

Создайте публикацию `p4`, а затем подпишитесь на неё. Параметр публикации `publish_via_partition_root` имеет значение true. При этом определяются фильтры строк для секционированной таблицы (`parent`) и для её секции (`child`).

```
test_pub=# CREATE PUBLICATION p4 FOR TABLE parent WHERE (a < 5), child WHERE (a >= 5)
test_pub-# WITH (publish_via_partition_root=true);
CREATE PUBLICATION
```

```
test_sub=# CREATE SUBSCRIPTION s4
test_sub-# CONNECTION 'host=localhost dbname=test_pub application_name=s4'
test_sub-# PUBLICATION p4;
```

```
CREATE SUBSCRIPTION
```

Вставьте некоторые значения непосредственно в таблицы `parent` и `child`. При их репликации будет использоваться фильтр строк `parent` (так как `publish_via_partition_root` имеет значение true).

```
test_pub=# INSERT INTO parent VALUES (2), (4), (6);
INSERT 0 3
test_pub=# INSERT INTO child VALUES (3), (5), (7);
INSERT 0 3

test_pub=# SELECT * FROM parent ORDER BY a;
 a
---
 2
 3
 4
 5
 6
 7
(6 rows)
```

```
test_sub=# SELECT * FROM parent ORDER BY a;
 a
---
 2
 3
 4
(3 rows)
```

Повторите ту же проверку с другим значением `publish_via_partition_root`. Теперь параметр публикации `publish_via_partition_root` будет иметь значение false, а фильтр строк будет определён в секции (`child`).

```
test_pub=# DROP PUBLICATION p4;
DROP PUBLICATION
test_pub=# CREATE PUBLICATION p4 FOR TABLE parent, child WHERE (a >= 5)
test_pub-# WITH (publish_via_partition_root=false);
CREATE PUBLICATION
```

```
test_sub=# ALTER SUBSCRIPTION s4 REFRESH PUBLICATION;
ALTER SUBSCRIPTION
```

Выполните на публикующем сервере те же операции INSERT. Они реплицируются с использованием фильтра строк в таблице `child` (так как `publish_via_partition_root` имеет значение false).

```
test_pub=# TRUNCATE parent;
TRUNCATE TABLE
test_pub=# INSERT INTO parent VALUES (2), (4), (6);
INSERT 0 3
test_pub=# INSERT INTO child VALUES (3), (5), (7);
INSERT 0 3

test_pub=# SELECT * FROM parent ORDER BY a;
 a
---
 2
 3
 4
 5
 6
```

```
 7
(6 rows)
```

```
test_sub=# SELECT * FROM child ORDER BY a;
 a
---
 5
 6
 7
(3 rows)
```

28.5. Списки столбцов

Для каждой публикации можно дополнительно задать, какие столбцы каждой таблицы реплицируются для подписчиков. При этом в таблице на стороне подписчика должны быть как минимум все публикуемые столбцы. Если список столбов не задан, реплицируются все столбцы публикуемой таблицы. Подробнее синтаксис рассматривается в описании CREATE PUBLICATION.

Ограничение множества столбцов может быть продиктовано соображениями производительности или требованиями к поведению. Однако из соображений безопасности использовать это не следует: злоумышленник на стороне подписчика может получить данные из столбцов, которые не были опубликованы явно. Когда важно обеспечить безопасность, надо применять защиту на стороне публикации.

Если список столбцов не задан, все столбцы, добавляемые в таблицу позднее, будут реплицироваться автоматически. Это означает, что наличие списка, включающего все столбцы, не равнозначно отсутствию списка столбцов.

Список столбцов может содержать только простые ссылки на столбцы. Порядок столбцов в этом списке не сохраняется.

Указать список столбцов в публикации, которая также публикует все таблицы в схеме (`FOR TABLES IN SCHEMA`), нельзя.

Для секционированных таблиц параметр публикации `publish_via_partition_root` определяет, какой список столбцов использовать. Если `publish_via_partition_root` имеет значение `true`, используется список столбцов корневой секционированной таблицы. В противном случае, если `publish_via_partition_root` имеет значение `false` (по умолчанию), используется список столбцов каждой секции.

Если через публикацию реплицируются операции `UPDATE` или `DELETE`, список столбцов должен содержать столбцы, которые входят в идентификатор реплики для данной таблицы (см. `REPLICA IDENTITY`). Если же через публикацию реплицируются только операции `INSERT`, столбцы идентификатора реплики в списке столбцов могут отсутствовать.

Для команды `TRUNCATE` списки столбцов не имеют значения.

Во время начальной синхронизации данных копируются только публикуемые столбцы. Однако если на стороне подписчика используется версия ниже 15, при начальной синхронизации копируются все столбцы в таблице, то есть любые списки столбцов игнорируются.

Внимание: объединение списков столбцов из нескольких публикаций

В настоящее время не поддерживаются подписки, связанные с несколькими публикациями, в которых одна таблица опубликована с разными списками столбцов. Команда CREATE SUBSCRIPTION не допускает создание таких подписок, но это всё равно может произойти при добавлении или изменении списков столбцов на стороне публикации после создания подписки.

Это означает, что изменение списков столбцов таблиц в публикациях, у которых уже есть подписчики, может привести к возникновению ошибок на стороне подписчика.

Если подписка затронута этой проблемой, единственный способ возобновить репликацию — изменить один из списков столбцов на стороне публикации, чтобы все они совпадали; а затем либо заново создать подписку, либо использовать `ALTER SUBSCRIPTION ... DROP PUBLICATION`, чтобы удалить одну из проблемных публикаций и добавить её снова.

28.5.1. Примеры

Создайте таблицу `t1`, которая будет использоваться в следующем примере.

```
test_pub=# CREATE TABLE t1(id int, a text, b text, c text, d text, e text, PRIMARY
 KEY(id));
CREATE TABLE
```

Создайте публикацию `p1`, в которой для таблицы `t1` определяется список столбцов, уменьшающий количество реплицируемых столбцов. Заметьте, что порядок имён столбцов в этом списке не имеет значения.

```
test_pub=# CREATE PUBLICATION p1 FOR TABLE t1 (id, b, a, d);
CREATE PUBLICATION
```

Списки столбцов для каждой публикации можно просмотреть в `psql`.

```
test_pub=# \dRp+
                               Publication p1
  Owner   | All tables | Inserts | Updates | Deletes | Truncates | Via root
----------+------------+---------+---------+---------+-----------+----------
 postgres | f          | t       | t       | t       | t         | f
Tables:
    "public.t1" (id, a, b, d)
```

Списки столбцов для каждой таблицы можно просмотреть в `psql`.

```
test_pub=# \d t1
                 Table "public.t1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 a      | text    |           |          |
 b      | text    |           |          |
 c      | text    |           |          |
 d      | text    |           |          |
 e      | text    |           |          |
Indexes:
    "t1_pkey" PRIMARY KEY, btree (id)
Publications:
    "p1" (id, a, b, d)
```

Создайте на сервере подписчика таблицу `t1`, которая теперь должна содержать лишь подмножество столбцов, существующих в публикуемой таблице `t1`, а также создайте публикацию `s1`, связанную с публикацией `p1`.

```
test_sub=# CREATE TABLE t1(id int, b text, a text, d text, PRIMARY KEY(id));
CREATE TABLE
test_sub=# CREATE SUBSCRIPTION s1
test_sub-# CONNECTION 'host=localhost dbname=test_pub application_name=s1'
test_sub-# PUBLICATION p1;
CREATE SUBSCRIPTION
```

Добавьте на стороне публикации в таблицу `t1` несколько строк.

```
test_pub=# INSERT INTO t1 VALUES(1, 'a-1', 'b-1', 'c-1', 'd-1', 'e-1');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES(2, 'a-2', 'b-2', 'c-2', 'd-2', 'e-2');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES(3, 'a-3', 'b-3', 'c-3', 'd-3', 'e-3');
INSERT 0 1
test_pub=# SELECT * FROM t1 ORDER BY id;
 id |  a  |  b  |  c  |  d  |  e
----+-----+-----+-----+-----+-----
  1 | a-1 | b-1 | c-1 | d-1 | e-1
  2 | a-2 | b-2 | c-2 | d-2 | e-2
  3 | a-3 | b-3 | c-3 | d-3 | e-3
(3 rows)
```

Как можно видеть, реплицировались только данные из списка столбцов в публикации `p1`.

```
test_sub=# SELECT * FROM t1 ORDER BY id;
 id |  b  |  a  |  d
----+-----+-----+-----
  1 | b-1 | a-1 | d-1
  2 | b-2 | a-2 | d-2
  3 | b-3 | a-3 | d-3
(3 rows)
```

28.6. Конфликты

Логическая репликация работает подобно обычным операциям DML в том смысле, что данные будут изменены, даже если они независимо изменялись на стороне подписчика. Если входящие данные нарушат какие-либо ограничения, репликация остановится. Эта ситуация называется *конфликтом*. При репликации операций `UPDATE` или `DELETE` отсутствие данных не вызывает конфликта, так что такие операции просто пропускаются.

Операции логической репликации выполняются с правами роли, которой принадлежит подписка. Поэтому при отсутствии необходимых прав доступа к целевым таблицам возникнут конфликты репликации, как и при включённой для целевых таблиц защите на уровне строк, ограничивающей владельца подписки. При этом не имеет значения, существует ли политика, которая бы запрещала реплицируемые операции `INSERT`, `UPDATE`, `DELETE` или `TRUNCATE`. Ограничение, связанное с защитой на уровне строк, может быть убрано в следующих версиях Postgres Pro.

В случае конфликта выдаётся ошибка и репликация останавливается; разрешить возникшую проблему пользователь должен вручную. Подробности конфликта можно найти в журнале сервера-подписчика.

Разрешить конфликт можно, изменив данные или разрешения на стороне подписчика, чтобы они не конфликтовали с поступающими изменениями, или пропустив транзакцию, конфликтующую с существующими данными. Когда в случае конфликта выдаётся ошибка, репликация останавливается, а рабочий процесс логической репликации выводит в журнал сервера подписчика сообщение следующего вида:

```
ERROR:  duplicate key value violates unique constraint "test_pkey"
DETAIL:  Key (c)=(1) already exists.
CONTEXT:  processing remote data for replication origin "pg_16395" during "INSERT" for
 replication target relation "public.test" in transaction 725 finished at 0/14C0378
```

В этом сообщении можно увидеть LSN транзакции, содержащей изменение, которое нарушает ограничение, и имя источника данных репликации (в данном случае LSN 0/14C0378 и `pg_16395`, соответственно). Транзакцию, вызвавшую конфликт, можно пропустить, выполнив команду `ALTER SUBSCRIPTION ... SKIP` с LSN её завершения (то есть LSN 0/14C0378). В качестве LSN завершения транзакции может задаваться LSN, с которым транзакция была зафиксирована или подготовлена на сервере публикации. Конфликтующую транзакцию также можно пропустить, вызвав функцию `pg_replication_origin_advance()`. Прежде чем вызывать эту функцию, нужно либо временно отключить подписку, выполнив `ALTER SUBSCRIPTION ... DISABLE`, либо использовать подписку с параметром `disable_on_error`. Затем можно вызвать функцию `pg_replication_origin_advance()`, передав ей *node_name* (то есть `pg_16395`) и LSN, следующий за LSN завершения (то есть 0/14C0379). Текущие позиции источников репликации можно увидеть в системном представлении `pg_replication_origin_status`. Обратите внимание: когда пропускается вся транзакция, пропускаются все её изменения, в том числе не нарушающие никаких ограничений. В результате состояние подписчика легко может оказаться несогласованным.

Если для `streaming` установлен режим `parallel`, LSN завершения неудачных транзакций может не регистрироваться. В этом случае может потребоваться включить или отключить режим потоковой передачи (изменить значение на `on` или `off`) и снова вызвать те же конфликты, чтобы LSN завершения неудачной транзакции был записан в журнал сервера. Подробная информация об использовании LSN завершения находится в описании `ALTER SUBSCRIPTION... SKIP`.

28.7. Ограничения

Логическая репликация в настоящее время имеет ограничения и недостатки, описанные ниже. Они могут быть устранены в будущих выпусках.

- Схема базы данных и команды DDL не реплицируются. Изначальную схему можно скопировать, воспользовавшись командой `pg_dump --schema-only`. Последующие изменения схемы необходимо будет синхронизировать вручную. (Заметьте, однако, что схемы не обязательно должны быть абсолютно идентичными на обеих сторонах репликации.) Если определения схемы в исходной базе данных меняются, логическая репликация работает надёжно — когда данные после изменения схемы прибывают на сторону подписчика, но не вписываются в схему его таблиц, выдаётся ошибка, требующая обновления схемы. Во многих случаях возникновение таких ошибок можно предупредить, сначала применяя дополняющие изменения на подписчике.
- Данные последовательностей не реплицируются. Данные в столбцах serial или столбцах идентификации, выдаваемые последовательностями, конечно, будут реплицированы в составе таблицы, но сама последовательность на подписчике будет сохранять стартовое значение. Если подписчик используется в качестве базы только для чтения, обычно это не является проблемой. Если же, однако, предусматривается возможность переключения на базу подписчика некоторым образом, текущие значения в этих последовательностях нужно будет обновить, либо скопировав текущие данные из базы публикации (вероятно, с применением `pg_dump`), либо выбрав достаточно большие значения из самих таблиц.
- Репликация команд `TRUNCATE` поддерживается, но опустошение групп таблиц, соединённых внешними ключами, стоит выполнять с осторожностью. При репликации действия `TRUNCATE` подписчик опустошит ту же группу таблиц, которая была опустошена на публикующем сервере (явным образом или в результате `CASCADE`), исключая таблицы, не входящие в подписку. Данная операция завершится корректно, если все затронутые таблицы включены в одну и ту же подписку. Если же некоторые таблицы, подлежащие опустошению на подписчике, связаны по внешнему ключу с таблицами, не входящими в данную подписку, операция опустошения на сервере-подписчике завершится ошибкой.
- Большие объекты (см. Главу 33) не реплицируются. Это ограничение нельзя обойти никак, кроме как хранить данные в обычных таблицах.
- Репликация поддерживается только для таблиц, включая секционированные. При попытке реплицировать отношения другого рода, например, представления, матпредставления или сторонние таблицы, будет выдана ошибка.
- При репликации между секционированными таблицами изменения по умолчанию фактически реплицируются из конечных секций на стороне публикации, поэтому соответствующие целевые таблицы должны существовать и на стороне подписчика. Они могут быть тоже конечными секциями, могут дополнительно разбиваться на секции или могут быть даже независимыми таблицами. На стороне публикации также может быть указано, что изменения бу-

дут реплицироваться как произошедшие в корневой секционированной таблице (с её именем и схемой), а не в той конечной секции, где они фактически имели место (см. параметр `publish_via_partition_root` команды `CREATE PUBLICATION`).

- При использовании характеристики `REPLICA IDENTITY FULL` для опубликованных таблиц важно помнить, что к подписчикам нельзя применить операции `UPDATE` и `DELETE`, если в таблицах есть атрибуты с типами данных (например, `point` или `box`), для которых отсутствует класс операторов индекса B-дерева или хеш-индекса по умолчанию. Однако это ограничение можно обойти, определив в таблице первичный ключ или репликационный идентификатор.

28.8. Архитектура

Логическая репликация начинается с копирования снимка данных в базе данных публикации. По завершении этой операции изменения на стороне публикации передаются подписчику в реальном времени, когда они происходят. Подписчик применяет изменения в том же порядке, в каком они вносятся на узле публикации, так что для публикаций в рамках одной подписки гарантируется транзакционная целостность.

Логическая репликация построена по схеме, подобной физической потоковой репликации (см. Подраздел 25.2.5). Она реализуется процессами `walsender` (передачи WAL) и `apply` (применения). Процесс walsender запускает логическое декодирование (описанное в Главе 47) WAL и загружает стандартный модуль вывода логического декодирования (`pgoutput`). Этот модуль преобразует изменения, считываемые из WAL, в протокол логической репликации (см. Раздел 53.5) и отфильтровывает данные согласно спецификации публикации. Затем данные последовательно передаются по протоколу логической репликации рабочему процессу применения изменений, который сопоставляет данные с логическими таблицами и применяет отдельные изменения по мере их поступления, сохраняя транзакционный порядок.

Процесс применения изменений в базе данных подписчика всегда выполняется со значением `session_replication_role` равным `replica`. Это означает, что по умолчанию триггеры и правила не будут срабатывать на подписчике. При желании пользователи могут включить триггеры и правила для таблицы, используя команду `ALTER TABLE` с предложениями `ENABLE TRIGGER` и `ENABLE RULE`.

Процесс применения логической репликации в настоящее время вызывает только триггеры уровня строк, но не триггеры операторов. Однако начальная синхронизация таблицы реализована как команда `COPY` и поэтому вызывает триггеры для `INSERT` и уровня строк, и уровня оператора.

28.8.1. Начальный снимок

Начальные данные существующих таблиц в подписке помещаются в снимок и копируются в параллельном экземпляре процесса применения особого вида. Этот процесс создаёт собственный слот репликации и производит копирование существующих данных. Как только копирование будет закончено, содержимое таблицы станет видимым для других серверных процессов. Когда существующие данные будут скопированы, этот рабочий процесс переходит в режим синхронизации, в котором таблица приводится в синхронизированное состояние для основного процесса применения, то есть передаёт все изменения, произошедшие во время начального копирования данных, используя стандартную логическую репликацию. На этом этапе синхронизации изменения применяются и фиксируются в том же порядке, что и на стороне публикации. По завершении синхронизации управление репликацией этой таблицы возвращается главному процессу, который продолжает репликацию в обычном режиме.

> **Примечание**
>
> Параметр публикации `publish` влияет только на то, какие DML-операции будут реплицироваться. Он не влияет на копирование существующих данных таблиц при начальной синхронизации данных.

28.9. Мониторинг

Так как логическая репликация построена по схеме, подобной физической потоковой репликации, мониторинг публикующего узла подобен мониторингу ведущего сервера при физической репликации (см. Подраздел 25.2.5.2).

Информацию о подписке для мониторинга можно получить в представлении `pg_stat_subscription`. Это представление содержит по одной строке для каждого рабочего процесса подписчика. В зависимости от состояния подписки, с ней может быть связано ноль или более активных рабочих процессов.

Обычно для включённой подписки работает только один процесс применения. Для отключённой или нарушенной подписки это представление будет содержать ноль строк. Если выполняется начальная синхронизация данных для каких-либо таблиц, для этих таблиц будут показаны дополнительные рабочие процессы, производящие синхронизацию. Более того, если транзакция `streaming` применяется параллельно, могут существовать дополнительные параллельные процессы применения.

28.10. Безопасность

Роль, используемая для подключения репликации, должна иметь атрибут `REPLICATION` (или быть суперпользователем). Если у роли отсутствуют свойства `SUPERUSER` и `BYPASSRLS`, при репликации могут выполняться политики защиты строк, определённые на стороне публикации. Если эта роль не может доверять владельцам всех таблиц, добавьте в строку подключения `options=-crow_security=off`; если владелец таблицы добавит политику защиты строк позже, при таком значении параметра репликация остановится, но политика выполняться не будет. Доступ для этой роли должен быть настроен в `pg_hba.conf`, и эта роль также должна иметь атрибут `LOGIN`.

Чтобы иметь возможность скопировать исходные данные таблицы, роль, используемая для соединения репликации, должна иметь право `SELECT` в публикуемой таблице (или быть суперпользователем).

Чтобы создать публикацию, пользователь должен иметь право `CREATE` в базе данных.

Чтобы добавлять таблицы в публикацию, пользователь должен иметь права владельца для этих таблиц. Добавлять в публикацию все таблицы схемы может только суперпользователь. Также только суперпользователям разрешено создавать публикации, публикующие все таблицы или все таблицы в схеме автоматически.

На данный момент прав для публикаций нет. Любая подписка (имеющая возможность подключения) может получить доступ к любой публикации. Таким образом, если необходимо скрыть некоторую информацию от определённых подписчиков, например, используя фильтры строк или списки столбцов или не добавляя всю таблицу в публикацию, имейте в виду, что та же информация может раскрываться в других публикациях в той же базе данных. В будущем в Postgres Pro могут быть добавлены права для публикаций, чтобы обеспечить более детальный контроль доступа.

Чтобы создать подписку, пользователь должен иметь права роли `pg_create_subscription`, а также права `CREATE` для базы данных.

Процесс применения подписки на уровне сеанса будет выполняться с правами владельца подписки. Однако при выполнении операции вставки, изменения, удаления или усечения в конкретной таблице происходит переключение роли на владельца таблицы и выполнит операцию с правами владельца таблицы. Это означает, что владелец подписки должен иметь право `SET ROLE` для каждой роли, которой принадлежит реплицируемая таблица.

Если подписка сконфигурирована с `run_as_owner = true`, переключение пользователей не произойдёт. Вместо этого все операции будут выполняться с правами владельца подписки. В этом случае владельцу подписки нужны только права `SELECT`, `INSERT`, `UPDATE` и `DELETE` из целевой таблицы, но не `SET ROLE` для владельца таблицы. Однако это также означает, что любой пользователь, владеющий таблицей, в которую реплицируются данные, может выполнять произвольный код с правами владельца подписки. Например, это можно сделать, добавив триггер к одной из принадлежащих этому пользователю таблиц. Поскольку обычно нежелательно позволять одной роли свободно присваивать права другой, не рекомендуется использовать этот параметр, кроме случаев, когда безопасность пользователей в базе данных не вызывает сомнений.

На публикующем сервере права проверяются только один раз при установлении подключения для репликации и не перепроверяются при чтении каждой записи изменения.

На стороне подписки права владельца подписки перепроверяются для каждой транзакции, когда она применяется. Если рабочий процесс осуществляет применение транзакции, когда в параллельной транзакции меняется владелец подписки, применение текущей транзакции продолжается с правами старого владельца.

28.11. Параметры конфигурации

Логическая репликация требует установки нескольких параметров конфигурации. Большинство параметров актуальны только на одной стороне репликации. Однако параметр `max_replication_slots` используется как на публикующем сервере, так и на подписчике, но имеет разное значение.

28.11.1. Публикующие серверы

Для `wal_level` должно быть установлено значение `logical`.

Для параметра `max_replication_slots` должно быть задано число не меньше ожидаемого числа подписчиков плюс некоторый резерв для синхронизации таблиц.

В `max_wal_senders` должно быть значение как минимум равное `max_replication_slots` плюс число возможных физических реплик, работающих одновременно.

В работе передатчика данных логической репликации также учитывается параметр `wal_sender_timeout`.

28.11.2. Подписчики

Для параметра `max_replication_slots` должно быть задано значение не меньше числа подписок, на которые будет подписываться данный подписчик, плюс некоторый резерв для синхронизации таблиц.

Для `max_logical_replication_workers` должно быть установлено значение, равное как минимум количеству подписок (для ведущих процессов применения), плюс некоторый резерв для рабочих процессов синхронизации таблиц и рабочих процессов параллельного применения.

Кроме того, может потребоваться изменить `max_worker_processes`, чтобы это число включало дополнительные рабочие процессы для репликации (как минимум `max_logical_replication_workers` + `1`). Обратите внимание, что некоторые расширения и параллельные запросы также занимают слоты из числа `max_worker_processes`.

Параметр `max_sync_workers_per_subscription` управляет степенью распараллеливания копирования начальных данных в процессе инициализации подписки или при добавлении новых таблиц.

`max_parallel_apply_workers_per_subscription` управляет степенью распараллеливания для потоковой передачи незавершённых транзакций с параметром подписки `streaming = parallel`.

На рабочие процессы логической репликации также влияют параметры `wal_receiver_timeout`, `wal_receiver_status_interval` и `wal_retrieve_retry_interval`.

28.12. Быстрая настройка

Сначала установите параметры конфигурации в `postgresql.conf`:

```
wal_level = logical
```

Другие необходимые параметры по умолчанию имеют значения, подходящие для базовой настройки.

В файл `pg_hba.conf` необходимо внести изменения, чтобы разрешить репликацию (конкретные значения будут зависеть от фактической конфигурации сети и имени пользователя, с которым вы будете подключаться):

```
host     all     repuser     0.0.0.0/0     md5
```

Затем в базе данных публикации выполните:

```
CREATE PUBLICATION mypub FOR TABLE users, departments;
```

И в базе данных подписчика:

```
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=foo host=bar user=repuser' PUBLICATION
 mypub;
```

Показанная выше команда запустит процесс репликации, который вначале синхронизирует исходное содержимого таблиц `users` и `departments`, а затем начнёт перенос инкрементальных изменений в этих таблицах.

Глава 29. JIT-компиляция

В этой главе рассказывается о том, что такое JIT-компиляция, и как можно настроить её в Postgres Pro.

29.1. Что такое JIT-компиляция?

JIT-компиляция (Just-In-Time compilation, Компиляция «точно в нужное время») — это производимая во время выполнения процедура преобразования интерпретируемого варианта исполнения программы в программу на языке процессора. Например, вместо использования универсального кода, способного вычислять произвольные SQL-выражения, для вычисления конкретного условия `WHERE a.col = 3` можно сгенерировать функцию, предназначенную именно для этого выражения, которую сможет выполнять непосредственно процессор, так что она будет выполняться быстрее.

В Postgres Pro поддержка JIT-компиляции с использованием *LLVM* включается при сборке Postgres Pro с ключом `--with-llvm`.

29.1.1. Операции, ускоряемые с применением JIT

В настоящее время реализация JIT в Postgres Pro поддерживает ускорение вычисления выражений и преобразования кортежей. В будущем могут быть ускорены и некоторые другие операции.

Вычисление выражений производится при обработке предложений `WHERE`, целевых списков, агрегатов и проекций. Оно может быть ускорено в результате генерирования кода, предназначенного для каждого конкретного случая.

Преобразование кортежей — процедура перевода кортежа с диска (см. Подраздел 63.6.1) в развёрнутое представление в памяти. Его можно ускорить, создав функции, предназначенные для определённой структуры таблицы и количества извлекаемых столбцов.

29.1.2. Встраивание

СУБД Postgres Pro очень гибка и позволяет определять новые типы данных, функции, операторы и другие объекты базы данных; см. Главу 36. В действительности встроенные объекты реализуются практически теми же механизмами. С этой гибкостью связаны некоторые издержки, например, сопутствующие вызовам функций (см. Раздел 36.3). Для сокращения этих издержек JIT-компиляция может встраивать тела маленьких функций в код выражений, использующих их. Это позволяет соптимизировать значительный процент подобных издержек.

29.1.3. Оптимизация

В LLVM поддерживается оптимизация сгенерированного кода. Некоторые оптимизации обходятся достаточно дёшево и могут выполняться при использовании JIT в любом случае, тогда как другие оправданы только при длительных запросах. Подробнее об оптимизации рассказывается в *https://llvm.org/docs/Passes.html#transform-passes*.

29.2. Когда применять JIT?

JIT-компиляция имеет смысл в первую очередь для длительных запросов, нагружающих процессор. Например, такой характер обычно имеют аналитические запросы. Для быстрых запросов накладные расходы, связанные с выполнением JIT-компиляции, часто будут превышать выигрыш от их ускорения.

Решение об использовании JIT-компиляции принимается на основании общей стоимости запроса (см. Главу 65 и Подраздел 18.7.2). Стоимость запроса сравнивается со значением jit_above_cost, и если она оказывается больше, производится JIT-компиляция. Затем принимаются ещё два решения. Во-первых, если его стоимость превышает и значение jit_inline_above_cost, тела небольших функций и операторов, фигурирующих в запросе, будут встраиваться в вызывающий код. Во-вторых, если стоимость запроса превышает значение jit_optimize_above_cost, при генерации кода задействуются дорогостоящие оптимизации для улучшения сгенерированного кода. Обе эти операции увеличивают накладные расходы JIT, но могут значительно сократить время выполнения запроса.

Эти решения принимаются на основе стоимости во время планирования, а не исполнения запроса. Это означает, что в случае использования подготовленных операторов и общего плана (см. Раздел «Примечания») принятие решений зависит от параметров конфигурации, действующих во время подготовки запроса, а не во время выполнения.

Примечание

Если параметр jit имеет значение `off` или сервер не поддерживает JIT (например, потому что он был скомпилирован без `--with-llvm`), JIT-компиляция выполняться не будет, даже если она была бы выгодна, исходя из описанных выше критериев. Присвоенное параметру jit значение `off` учитывается и во время планирования, и во время выполнения запросов.

EXPLAIN позволяет определить, используется ли JIT-компиляция. Например, так выглядит запрос, не использующий JIT:

```
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                 QUERY PLAN
-------------------------------------------------------------------
-------------------------------------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=0.303..0.303 rows=1
 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual
 time=0.017..0.111 rows=356 loops=1)
 Planning Time: 0.116 ms
 Execution Time: 0.365 ms
(4 rows)
```

Учитывая стоимость планирования, отказ от использования JIT вполне обоснован, так как стоимость JIT-компиляции оказалась бы больше, чем выигрыш от оптимизации. Если уменьшить ограничение стоимости, JIT будет использоваться:

```
=# SET jit_above_cost = 10;
SET
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                 QUERY PLAN
-------------------------------------------------------------------
-------------------------------------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual time=6.049..6.049 rows=1
 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4) (actual
 time=0.019..0.052 rows=356 loops=1)
 Planning Time: 0.133 ms
 JIT:
   Functions: 3
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 1.259 ms, (Deform 0.000 ms), Inlining 0.000 ms, Optimization
 0.797 ms, Emission 5.048 ms, Total 7.104 ms
 Execution Time: 7.416 ms
```

В данном случае видно, что JIT используется, но встраивание (Inlining) и дорогостоящие оптимизации (Optimization) не выполнялись. Чтобы их включить, нужно было также уменьшить jit_inline_above_cost и jit_optimize_above_cost.

29.3. Конфигурация

Переменная конфигурации jit определяет, возможно ли использование JIT-компиляции. Если она включена, переменные jit_above_cost, jit_inline_above_cost и jit_optimize_above_cost определяют, будет ли выполняться JIT-компиляция для запросов, и насколько ресурсоёмкой она может быть.

Параметр jit_provider определяет, какая реализация JIT-компиляции должна использоваться. Изменять его может потребоваться лишь в редких случаях. См. Подраздел 29.4.2.

Для целей отладки и разработки предназначен ряд дополнительных параметров конфигурации, описанных в Разделе 18.18.

29.4. Расширяемость

29.4.1. Поддержка встраивания кода для расширений

Если вы хотите компилировать расширения Postgres Pro с поддержкой встраивания JIT, настройте окружение для разработки как описано в Подразделе 16.1.5.

Механизм JIT в Postgres Pro может встраивать в код реализации функций (типа `C` и `internal`), а также операторов, использующих эти функции. Чтобы это встраивание выполнялось для функций в расширениях, должно быть доступно определение этих функций. При сборке с использованием PGXS расширения для сервера, скомпилированного с JIT-поддержкой LLVM, соответствующие файлы будут собираться и устанавливаться автоматически.

Соответствующие файлы должны устанавливаться в `$pkglibdir/bitcode/$extension/`, а информация о них должна вноситься в `$pkglibdir/bitcode/$extension.index.bc`, где `$pkglibdir` — каталог, который выдаёт команда `pg_config --pkglibdir`, а `$extension` — базовое имя разделяемой библиотеки данного расширения.

Примечание

Для функций, встроенных в Postgres Pro, двоичный код устанавливается в `$pkglibdir/bitcode/postgres`.

29.4.2. Подключаемые провайдеры JIT

Postgres Pro предоставляет реализацию JIT на базе LLVM. Интерфейс JIT предусматривает динамическое подключение провайдера и позволяет менять провайдер без перекомпиляции (хотя в настоящее время при сборке предоставляется поддержка встраивания только для LLVM). Провайдер выбирается параметром jit_provider.

29.4.2.1. Интерфейс провайдера JIT

Провайдер JIT загружается путём динамической загрузки заданной разделяемой библиотеки. Для поиска её используется обычный путь поиска библиотек. Чтобы предоставить требуемые функции-обработчики провайдера JIT и показать, что библиотека является реализацией провайдера JIT, она должна содержать функцию, имеющую в C имя `_PG_jit_provider_init`. Этой функции передаётся структура, которую нужно заполнить указателями на функции-обработчики определённых действий.

```
struct JitProviderCallbacks
{
    JitProviderResetAfterErrorCB reset_after_error;
    JitProviderReleaseContextCB release_context;
    JitProviderCompileExprCB compile_expr;
};
```

```
extern void _PG_jit_provider_init(JitProviderCallbacks *cb);
```

Глава 30. Улучшенная безопасность

В этой главе описываются механизмы улучшенной безопасности, имеющиеся в сертифицированной редакции Postgres Pro Standard. Кроме того, Postgres Pro Standard предоставляет следующие встроенные средства безопасности:

- Расширение pg_proaudit, позволяющее отслеживать различные события, связанные с безопасностью.
- Расширенные политики аутентификации, которые позволяют ограничивать использование базы данных и устанавливать требования к паролям, назначая профили ролям. Более подробно о профилях рассказывается в CREATE PROFILE. Все профили, имеющиеся в вашем кластере, показываются в каталоге `pg_profile`.

30.1. Очистка памяти

В зависимости от параметров конфигурации очистки памяти Postgres Pro Standard может автоматически перезаписывать данные нулевыми байтами перед удалением. В этом разделе описывается, как очищаются данные различных типов. По умолчанию все параметры очистки памяти включены.

30.1.1. Удаление файлов из внешней памяти

Когда вы выполняете команды SQL, удаляющие файлы, соответствующее дисковое пространство возвращается операционной системе. К таким командам относятся:

- Команды, удаляющие объекты БД:
  - `DROP TABLE`
  - `DROP TEMPORARY TABLE`
  - `DROP MATERIALIZED VIEW`
  - `DROP INDEX`
  - `TRUNCATE`
  - `DROP DATABASE`
  - `DROP SCHEMA`
- Команды, пересоздающие объекты:
  - `VACUUM FULL`
  - `REINDEX`
  - `ALTER TABLE ADD COLUMN` (со значением по умолчанию)
  - `ALTER TABLE ALTER COLUMN TYPE`

Если параметр конфигурации wipe_file_on_delete включён, файлы перед удалением сначала заполняются нулевыми байтами.

> **Примечание**
>
> Команда `ALTER TABLE DROP COLUMN` не пересоздаёт файл. Все данные, содержащиеся в удаляемом столбце, остаются внутри страниц, хотя к ним и нельзя обращаться с помощью команд SQL. Чтобы физически удалить эти данные из файла, выполните `VACUUM FULL` после удаления столбца.

30.1.2. Очистка страниц

В соответствии с моделью MVCC (Multiversion Concurrency Control, Многоверсионное управление конкурентным доступом) при удалении строк (командой `DELETE`) данные, содержащиеся в этих строках, помечаются как удалённые, но физически не удаляются. При модификации строк (командой `UPDATE`) сначала удаляется старая версия строки, а затем вставляется новая, так что предыдущее содержимое так же сохраняется в странице.

Чтобы механизм MVCC работал корректно, Postgres Pro сохраняет удалённые данные в страницах до тех пор, пока эта версия может быть востребована какими-либо активными снимками. Если строка задействуется в индексах, то в страницах индексов так же сохраняются ссылки на строки, которые были удалены, но ещё не освобождены. Когда версия строки не востребована никакими снимками, её можно удалить, произведя процедуру `VACUUM`. При этом все ссылки на эту версию будут удалены и из индексов. Однако это удаление не подразумевает физическое уничтожение: в обычном режиме соответствующее место в странице помечается как свободное и может использоваться для размещения другой строки.

Для предотвращения доступа к удалённым версиям строк необходимо включить параметр wipe_heaptuple_on_delete в файле конфигурации `postgresql.conf`. При этом процесс `VACUUM` будет не только помечать место в страницах как свободное, но и заполнять его нулевыми байтами.

30.1.3. Очистка блоков памяти в ОЗУ

Во время работы сервера оперативная память (в ОЗУ) постоянно выделяется и освобождается. Postgres Pro использует собственную подсистему управления памятью, основанную на контекстах, в которой память всегда выделяется в одном из вложенных контекстов. При удалении контекста освобождается вся память, выделенная в нём и в соответствующих вложенных контекстах. Этот подход позволяет избежать утечек памяти. Тем не менее выделением и освобождением памяти в конце концов занимается операционная система. В обычных условиях освобождаемая область ОЗУ возвращается операционной системе и может быть выделена другому процессу.

Чтобы из освобождаемой области ОЗУ удалялись все данные, включите следующие параметры в файле конфигурации `postgresql.conf`:

- wipe_memctx_on_free — этот параметр включает заполнение нулевыми байтами освобождаемой памяти в каком-либо контексте.
- wipe_mem_on_free — этот параметр включает заполнение нулевыми байтами освобождаемой памяти, не принадлежащей никакому контексту. Хотя Postgres Pro всегда выделяет память в контекстах, вы можете воспользоваться этим параметром для большей уверенности.

30.1.4. Очистка файлов WAL

Журнал предзаписи (Write-Ahead Log, WAL) представляет собой стандартный способ обеспечения целостности данных, позволяющий осуществлять резервное копирование, восстановление на момент времени и репликацию данных между серверами. Концептуальной сущностью WAL является то, что изменения в каталогах данных (где располагаются таблицы и индексы) должны производиться только после фиксации изменений в журнале, то есть после того, как записи журнала, описывающие изменения, окажутся в постоянном хранилище. Таким образом WAL может содержать важные данные, которые необходимо защищать.

Сервер Postgres Pro всегда сохраняет определённое количество сегментов WAL в каталоге `$PGDATA/pg_wal`. Минимальный объём сохраняемых сегментов WAL определяется параметром `min_wal_size`. В случае большой нагрузки размер сегментов WAL может увеличиться до значения `max_wal_size` и даже несколько его превысить. Пока размер журнала WAL на диске остаётся больше `min_wal_size`, старые файлы сегментов WAL удаляются при освобождении. В противном случае сегменты WAL перезаписываются.

Для недопущения неавторизованного доступа к освобождённым сегментам WAL необходимо включить параметр wipe_xlog_on_free в файле конфигурации `postgresql.conf`. В этом случае сегмент WAL будет заполнен нулевыми байтами перед удалением или повторным использованием.

Более подробно конфигурация и использование WAL описывается в Главе 27.

30.2. Проверки целостности

30.2.1. Обзор

В Postgres Pro Standard входит утилита `pg_integrity_check`, предоставляющая следующие возможности для проверки целостности:

- Подсчёт и проверка контрольных сумм по требованию
- Встроенная проверка контрольных сумм при запуске сервера

`pg_integrity_check` может отслеживать контрольные суммы неизменяемых и дополнительных файлов, а также таблиц системных каталогов.

30.2.1.1. Неизменяемые файлы

К отслеживаемым неизменяемым файлам относятся исполняемые файлы, библиотеки и другие файлы, которые никогда не должны изменяться. Контрольные суммы таких файлов определяются в файле конфигурации `/opt/pgpro/std-17/share/security/system.conf`.

Файл `system.conf` включён в состав дистрибутива Postgres Pro Standard. Для каждого экземпляра Postgres Pro Standard существует один отдельный файл `system.conf`. Каждая строка в `system.conf` соответствует одному отслеживаемому объекту и содержит два поля: контрольную сумму, состоящую из 40 шестнадцатеричных цифр, и относительный путь к файлу. Эти значения разделяются тремя символами: пробел, минус, пробел. Контрольные суммы вычисляются и записываются в этот файл во время установки Postgres Pro Standard. При их расчёте учитывается и содержимое файла, и его атрибуты. Поэтому, например, при изменении прав доступа к файлу его контрольная сумма изменится.

30.2.1.2. Дополнительные файлы

Дополнительными файлами считаются файлы, которые могут быть изменены администратором баз данных. Контрольные суммы дополнительных файлов сохраняются в файлах конфигурации, находящихся в каталоге `share/security`. Для каждого кластера необходимо иметь отдельный файл конфигурации для дополнительных файлов. При этом действует следующее соглашение об именовании: путь к каталогу данных кластера (`PGDATA`), в котором косая черта заменяется подчёркиванием, дополняется окончанием `.user.conf`. Например, для кластера с каталогом данных `/var/lib/pgpro/std-17/data` файл конфигурации будет называться `_var_lib_pgpro_std-17_data.user.conf`.

Каждая строка в файле конфигурации соответствует одному отслеживаемому объекту и содержит два поля: контрольную сумму, состоящую из 40 шестнадцатеричных цифр, и относительный путь к файлу. Поля разделяются тремя символами: пробел, минус, пробел.

При расчёте контрольных сумм учитывается и содержимое, и атрибуты файлов. Поэтому, например, при изменении прав доступа к файлу его контрольная сумма изменится.

Чтобы настроить проверку контрольных сумм дополнительных файлов, администратор баз данных должен сделать следующее:

1. Создать файл конфигурации для каждого кластера, согласно описанному выше соглашению об именовании.
2. В созданном файле конфигурации перечислить все отслеживаемые дополнительные файлы. В качестве контрольной суммы можно задать любые 40 шестнадцатеричных цифр, например нули.
3. Выполнить следующую команду, чтобы пересчитать контрольные суммы, указав путь к каталогу данных вашего кластера:

```
pg_integrity_check -u -o -D /var/lib/pgpro/std-17/data
```

> **Примечание**
>
> Администратор также может, запустив эту команду, получить примерный файл конфигурации, а затем отредактировать его должным образом.

30.2.1.3. Таблицы системных каталогов

Таблицы системных каталогов представляют собой избранный набор данных, относящихся к экземпляру Postgres Pro Standard. Управление контрольными суммами таблиц системных каталогов осуществляется с помощью соответствующих файлов конфигурации для каждой базы данных в кластере. Проверку целостности таблиц системных каталогов можно настроить для нескольких баз данных путём создания отдельных файлов конфигурации с конкретными значениями контрольных сумм.

Каждая строка в файле конфигурации соответствует одному отслеживаемому объекту и содержит два поля: контрольную сумму, состоящую из 40 шестнадцатеричных цифр, и относительный путь к файлу. Поля разделяются тремя символами: пробел, минус, пробел.

Чтобы настроить проверку контрольных сумм избранных данных, для каждой БД в кластере администратор баз данных должен сделать следующее:

1. Создайте файл конфигурации для выбранной базы данных. Например, `*имя_базы_данных*-catalog.conf`.
2. В созданном файле конфигурации перечислить SQL-запросы, которые будут возвращать отслеживаемые данные. В качестве контрольной суммы можно задать любые 40 шестнадцатеричных цифр, например нули.
3. Выполнить следующую команду, чтобы пересчитать контрольные суммы, указав параметры подключения к вашей базе данных:

```
pg_integrity_check -c -o -C полный-путь-к-файлу-конфигурации -d postgres -h localhost -
p 5432 -U postgres
```

Чтобы проверить таблицы системного каталога, запустите `pg_integrity_check` для каждой базы данных, подлежащей проверке целостности.

> **Примечание**
>
> Администратор также может, запустив эту команду, получить примерный файл конфигурации, а затем отредактировать его должным образом.

30.2.2. Проверка целостности при запуске сервера

Контрольные суммы неизменяемых файлов всегда проверяются при запуске сервера Postgres Pro Standard. Если запуск сервера остановлен из-за несовпадения контрольных сумм, администратор должен разобраться в проблеме, разрешить её и перезапустить сервер.

Таблицы системных каталогов и дополнительные файлы при запуске сервера не проверяются. Вы можете проверить их вручную, выполнив `pg_integrity_check` после запуска сервера.

30.2.3. Планирование проверок целостности

Если вы используете систему семейства Linux, вы можете запланировать периодические проверки целостности, используя демон `cron`. Для этого измените файл `/etc/crontab`, добавив в него строки, определяющие частоту запуска утилиты `pg_integrity_check`. Файл `/etc/crontab` представляет собой системный файл, содержащий все инструкции для демона `cron`. Чтобы получить подробное описание формата файла `crontab` в вашей системе на базе Linux, выполните команду:

```
man 5 crontab
```

Примеры

Следующий пример иллюстрирует организацию проверок целостности в операционной системе Rosa SX:

```
# Каталог данных кластера
PGDATA = /var/lib/pgpro/std-17/data
# Файл журнала pg_integrity_check
LOG = /opt/pgpro/std-17/share/security/log
# Проверять неизменяемые файлы ежедневно в 00:05
5 0 * * *       root   /opt/pgpro/std-17/bin/pg_integrity_check -s >> $LOG
# Запускать проверки целостности в 14:15 в первый день месяца
15 14 1 * *     root   /opt/pgpro/std-17/bin/pg_integrity_check -s >> $LOG
# Запускать проверки целостности в 22.00 по выходным
0 22 * * 1-5    root   /opt/pgpro/std-17/bin/pg_integrity_check -s >> $LOG
#  Запускать проверки целостности ежедневно в 00:23, 2:23, 4:23 ...
23 0-23/2 * * * root   /opt/pgpro/std-17/bin/pg_integrity_check -s >> $LOG
#  Запускать проверки целостности в 4:05 по воскресеньям
5 4 * * sun     root   /opt/pgpro/std-17/bin/pg_integrity_check -s >> $LOG
```

30.3. Разграничение прав между привилегированными пользователями СУБД

Конфигурация всех СУБД на основе PostgreSQL включает роль суперпользователя, применяемую для начальной загрузки базы данных. Однако дальнейшее использование этой роли для рутинных операций с СУБД несёт в себе риски для информационной безопасности. Поскольку учётная запись суперпользователя обладает широким набором прав в системе, она становится мишенью киберпреступников, что может привести к ряду неблагоприятных последствий, таких как:

- Несанкционированный доступ к конфиденциальной информации.
- Утечка данных.
- Внесение опасных изменений в конфигурацию СУБД.
- Сбои в работе СУБД.

В качестве возможной меры противодействия подобным угрозам можно рассмотреть реализацию принципа разграничения полномочий с уменьшением числа операций, которые требуют применения прав суперпользователя. Это необходимо, чтобы исключить риски, связанные как с доверием к самому суперпользователю, так и на случай доступа к его учётной записи злоумышленника.

30.3.1. Обзор

Postgres Professional разработала и реализовала модель разделения обязанностей суперпользователя между двумя дополнительными административными ролями — администратором СУБД и администратором базы данных (Администратор БД). Кроме того, данная модель улучшает механизмы защиты от самостоятельного повышения привилегий и аудита действий всех пользователей.

Администратор СУБД отвечает за следующие действия:

- Управление сервером.
- Настройка репликации данных и резервного копирования.
- Создание баз данных.
- Настройка соединений с базой данных Postgres Pro.
- Назначение администраторов БД.

Администратор БД отвечает за следующие действия:

- Резервное копирование отдельной БД.

- Создание таблиц и других объектов в рамках отдельной БД.
- Создание пользователей БД.
- Выдача пользователям прав доступа.

После делегирования большинства рутинных задач суперпользователя новым контролируемым администраторам организация может отказаться от активного использования этой роли, несущей большое количество рисков. Для этого администратору инфраструктуры достаточно внести в файл `pg_hba.conf` строки, запрещающие соединения для суперпользователя, а также заблокировать возможность изменения этого файла другими администраторами.

Если привилегии суперпользователя необходимо временно вернуть для решения некоторой редкой сложной задачи, безопасно это можно сделать в технологическом режиме (single mode) с привлечением администратора инфраструктуры. Для перевода СУБД в технологический режим её необходимо остановить и перезапустить командой:

```
postgres --single -D DBMS_data_directory  other_options DB_name
```

Обратитесь к инструкции по установке дополнительно поставляемых модулей без участия суперпользователя в Разделе 16.2.

Расширение pg_proaudit, позволяющее регистрировать различные события, связанные с безопасностью, описано в Разделе F.39.

30.3.2. Создание дополнительных администраторов

30.3.2.1. Создание роли администратора СУБД

Данный раздел описывает создание роли и пользователя `PGPRO_DBMS_ADMIN` с соответствующими правами.

Данный скрипт создаёт роль `PGPRO_DBMS_ADMIN` и пользователя с этой ролью. Его запускает суперпользователь `postgres`. Обратите внимание, что скрипт использует следующие предопределённые роли:

- `pg_create_tablespace` позволяет выполнять команду `CREATE TABLESPACE` без прав суперпользователя.
- `pg_manage_profiles` позволяет выполнять команды `CREATE PROFILE`, `ALTER PROFILE` и `DROP PROFILE` без прав суперпользователя.

```
$ psql postgres
CREATE ROLE PGPRO_DBMS_ADMIN WITH CREATEDB CREATEROLE INHERIT LOGIN REPLICATION;
REVOKE ALL ON DATABASE postgres FROM PUBLIC;
GRANT CONNECT ON DATABASE postgres TO PGPRO_DBMS_ADMIN;
CREATE USER dbms_admin WITH CREATEDB CREATEROLE INHERIT LOGIN REPLICATION; -- Не
 забудьте установить первичный пароль
GRANT PGPRO_DBMS_ADMIN TO dbms_admin;
GRANT pg_read_all_settings TO PGPRO_DBMS_ADMIN WITH ADMIN OPTION;
GRANT pg_read_all_stats TO PGPRO_DBMS_ADMIN WITH ADMIN OPTION;
GRANT pg_stat_scan_tables TO PGPRO_DBMS_ADMIN WITH ADMIN OPTION;
GRANT pg_monitor TO PGPRO_DBMS_ADMIN WITH ADMIN OPTION;
GRANT pg_signal_backend TO PGPRO_DBMS_ADMIN WITH ADMIN OPTION;
GRANT pg_checkpoint TO PGPRO_DBMS_ADMIN WITH ADMIN OPTION;
GRANT pg_create_tablespace TO PGPRO_DBMS_ADMIN WITH ADMIN OPTION;
GRANT pg_manage_profiles TO PGPRO_DBMS_ADMIN WITH ADMIN OPTION;
EXIT;
```

Разрешение для роли `PGPRO_DBMS_ADMIN` действий по управлению конфигурацией, а также записи в журнал и резервного копирования/восстановление выдаётся следующей командой от имени суперпользователя `postgres`.

```
$ psql postgres
GRANT EXECUTE ON FUNCTION pg_reload_conf TO PGPRO_DBMS_ADMIN;
GRANT EXECUTE ON FUNCTION pg_rotate_logfile TO PGPRO_DBMS_ADMIN;
GRANT EXECUTE ON FUNCTION pg_create_restore_point TO PGPRO_DBMS_ADMIN;
GRANT EXECUTE ON FUNCTION pg_backup_start TO PGPRO_DBMS_ADMIN;
GRANT EXECUTE ON FUNCTION pg_backup_stop TO PGPRO_DBMS_ADMIN;
GRANT EXECUTE ON FUNCTION pg_switch_wal TO PGPRO_DBMS_ADMIN;
GRANT EXECUTE ON FUNCTION pg_promote TO PGPRO_DBMS_ADMIN;
GRANT EXECUTE ON FUNCTION pg_wal_replay_pause TO PGPRO_DBMS_ADMIN;
GRANT EXECUTE ON FUNCTION pg_wal_replay_resume TO PGPRO_DBMS_ADMIN;
EXIT;
```

30.3.2.2. Создание роли администратора БД

В рамках одной роли `PGPRO_DB_DBNAME_ADMIN` для одной БД может быть несколько пользователей. Также допускается, чтобы один пользователь включался в несколько разных ролей `PGPRO_DB_DBNAME_ADMIN`, если по должностным инструкциям он имеет право администрировать несколько разных БД.

Данный раздел описывает создание базы данных на примере БД под названием `DB1`, роли администратора БД для неё и пользователя `DB1_ADMIN`, наделённого соответствующими правами. На этом этапе для всех этих действий достаточно прав ранее созданного пользователя с ролью `PGPRO_DBMS_ADMIN`.

Данный скрипт создаёт базу данных `DB1`, роль `PGPRO_DB_1_ADMIN` и пользователя с этой ролью. Его запускает администратор СУБД:

```
$ psql postgres -U dbms_admin
SET ROLE PGPRO_DBMS_ADMIN;
CREATE ROLE PGPRO_DB_1_ADMIN WITH CREATEROLE INHERIT;
CREATE USER db1_admin WITH CREATEROLE INHERIT; -- Не забудьте установить первичный
 пароль
GRANT PGPRO_DB_1_ADMIN TO db1_admin;
GRANT PGPRO_DB_1_ADMIN TO PGPRO_DBMS_ADMIN;
GRANT pg_read_all_settings TO PGPRO_DB_1_ADMIN;
GRANT pg_read_all_stats TO PGPRO_DB_1_ADMIN;
GRANT pg_stat_scan_tables TO PGPRO_DB_1_ADMIN;
GRANT pg_monitor TO PGPRO_DB_1_ADMIN;
CREATE DATABASE db1 OWNER PGPRO_DB_1_ADMIN;
REVOKE CONNECT, TEMPORARY ON DATABASE db1 FROM PUBLIC;
```

30.3.3. Закрытие доступа для суперпользователя

Чтобы запретить соединение для суперпользователя, администратор инфраструктуры вносит в `pg_hba.conf` следующие строки:

```
TYPE     DATABASE  USER       ADDRESS       METHOD
local    all       postgres                 reject
host     all       postgres   127.0.0.1/32  reject
host     all       postgres   ::1/128       reject
```

Администратор инфраструктуры `root` запрещает пользователю `postgres` изменять файл `pg_hba.conf`:

```
# chown root pg_hba.conf
# chmod 640 pg_hba.conf
```

В результате пользователь `postgres` не может самостоятельно вернуть себе возможность соединения с СУБД:

```
# ls -lh pg_hba.conf
-rw-r----- 1 root postgres .......... pg_hba.conf
```

Чтобы новые ограничения вступили в силу, требуется перезапуск сервера СУБД:

```
# pg_ctl restart
```

Глава 31. Устранение неполадок

Postgres Pro предоставляет возможность выгружать состояние серверного процесса путём включения параметра конфигурации crash_info, что может быть полезно для диагностики и отладки. Затем можно сгенерировать файл состояния одним из следующих способов:

- Отправив сигнал `40` (также называемый сигналом выгрузки для диагностики):

  ```
  kill -40 идентификатор_процесса
  ```

- Используя функцию `pg_diagdump()`:

  ```
  SELECT pg_diagdump(идентификатор_процесса);
  ```

Здесь *`идентификатор_процесса`* — идентификатор серверного процесса, для которого нужно выгрузить состояние.

В результате Postgres Pro выгрузит состояние в файл в каталоге `$PGDATA/crash_info` по умолчанию или в каталоге, указанном в параметре конфигурации crash_info_location. Имя файла генерируется по следующему шаблону: `crash_`*`ид_файла`*`_pid`*`ид_процесса`*`.state`. Источники данных для выгрузки состояния можно задать в параметре конфигурации crash_info_dump.

В примере ниже показано, как создать и просмотреть файл состояния для серверного процесса с PID `23111`:

```
-- Создание файла состояния
SELECT pg_diagdump(23111);

-- Исследование каталога crash_info и его содержимого
SELECT pg_ls_dir('crash_info');

-- Чтение содержимого файла состояния
SELECT pg_read_file('crash_info/crash_1722943138419104_pid23111.state');
```

Часть IV. Клиентские интерфейсы

В этой части документации описываются клиентские программные интерфейсы, включённые в дистрибутив Postgres Pro. Все включённые в неё главы можно читать по отдельности. Существует множество других программных интерфейсов со своей документацией (некоторые наиболее популярные перечислены в Приложении J). Для изучения этой части нужно уметь работать с базой данных, используя SQL (см. Часть II), и, конечно же, знать выбранный язык программирования.

Глава 32. libpq — библиотека для языка C

libpq — это интерфейс Postgres Pro для программирования приложений на языке C. Библиотека libpq содержит набор функций, используя которые клиентские программы могут передавать запросы серверу Postgres Pro и принимать результаты этих запросов.

libpq также является базовым механизмом для нескольких других прикладных интерфейсов Postgres Pro, включая те, что написаны для C++, Perl, Python, Tcl и ECPG. Поэтому некоторые аспекты поведения libpq будут важны для вас, если вы используете один из этих пакетов. В частности, Раздел 32.15, Раздел 32.16 и Раздел 32.19 описывают поведение, видимое пользователю любого приложения, использующего libpq.

В конце этой главы приведены короткие программы (Раздел 32.22), показывающие, как использовать libpq в своих программах.

Клиентские программы, которые используют libpq, должны включать заголовочный файл `libpq-fe.h`и должны компоноваться с библиотекой libpq.

32.1. Функции управления подключением к базе данных

Следующие функции имеют дело с созданием подключения к серверу Postgres Pro. Прикладная программа может иметь несколько подключений к серверу, открытых одновременно. (Одна из причин этого заключается в необходимости доступа к более чем одной базе данных.) Каждое соединение представляется объектом `PGconn`, который можно получить от функции `PQconnectdb`, `PQconnectdbParams` или `PQsetdbLogin`. Обратите внимание, что эти функции всегда возвращают ненулевой указатель на объект, кроме случая, когда не остаётся свободной памяти даже для объекта `PGconn`. Прежде чем передавать запросы через объект подключения, следует вызвать функцию `PQstatus` для проверки возвращаемого значения в случае успешного подключения.

Предупреждение

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления доступных им для записи схем из пути поиска (`search_path`). Для этого можно присвоить параметру с ключом `options` значение `-csearch_path=`. Также можно выполнить `PQexec(`*`соединение`*`, "SELECT pg_catalog.set_config('search_path', '', false)")` после подключения. Это касается не только psql, но и любых других интерфейсов для выполнения произвольных SQL-команд.

Предупреждение

В системе Unix создание дочернего процесса на основе процесса, уже имеющего открытые подключения с помощью libpq, может привести к непредсказуемым результатам, потому что родительский и дочерний процессы совместно используют одни и те же сокеты и ресурсы операционной системы. По этой причине подобный подход не рекомендуется. Однако использование системного вызова `exec` из дочернего процесса для загрузки нового исполняемого файла является безопасным.

`PQconnectdbParams`

Создаёт новое подключение к серверу баз данных.

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
```

```
                                int expand_dbname);
```

Эта функция открывает новое соединение с базой данных, используя параметры, содержащиеся в двух массивах, завершающихся символом `NULL`. Первый из них, `keywords`, определяется как массив строк, каждая из которых представляет собой ключевое слово. Второй, `values`, даёт значение для каждого ключевого слова. В отличие от функции `PQsetdbLogin`, описываемой ниже, её набор параметров может быть расширен без изменения сигнатуры функции, поэтому при разработке новых приложений предпочтительнее использовать данную функцию (или её неблокирующие аналоги `PQconnectStartParams` и `PQconnectPoll`).

Ключевые слова-параметры, распознаваемые в настоящее время, приведены в Подразделе 32.1.2.

Передаваемые массивы могут быть пустыми (в этом случае будут использоваться все параметры по умолчанию) или содержать одно или несколько имён/значений параметров. При этом они должны быть одинаковой длины. Просмотр массивов завершается, как только в массиве `keywords` встречается `NULL`. Если элемент массива `values`, соответствующий отличному от `NULL` элементу `keywords`, содержит пустую строку или NULL, такой параметр пропускается и просматривается следующая пара элементов массива.

Когда `expand_dbname` имеет ненулевое значение, первый параметр *`dbname`* может содержать *строку подключения*. В этом случае она «разворачивается» в отдельные параметры подключения, извлечённые из этой строки. Значение считается строкой подключения, а не просто именем базы данных, если оно содержит знак равно (=) или начинается с обозначения схемы URI. (Подробнее форматы строк подключения описаны в Подразделе 32.1.1.) Таким способом обрабатывается только первое вхождение *`dbname`*, следующие параметры *`dbname`* будут восприниматься как просто имя базы данных.

Как правило, массивы параметров обрабатываются от начала к концу. Если какой-либо параметр указывается неоднократно, использоваться будет последнее значение (отличное от `NULL` и непустое). Это справедливо, в частности, и тогда, когда ключевое слово, заданное в строке подключения, конфликтует с заданным в массиве `keywords`. Таким образом, программист может по своему усмотрению решить, будут ли значения в массиве переопределять значения, заданными в строке подключения, или переопределяться ими. Элементы массива, предшествующие развёрнутому значению *`dbname`*, могут быть переопределены значениями в строке подключения, которые в свою очередь переопределяются элементами массива, следующими после *`dbname`* (и в этом случае речь идёт о непустых значениях).

После разбора всех элементов массива и развёрнутой строки подключения (если она задана), параметры подключения, которые остались незаданными, получают значения по умолчанию. Если незаданному параметру соответствует установленная переменная окружения (см. Раздел 32.15), будет использоваться её значение. Если такая переменная не задана, для параметра будет использоваться встроенное значение по умолчанию.

`PQconnectdb`

Создаёт новое подключение к серверу баз данных.

```
PGconn *PQconnectdb(const char *conninfo);
```

Эта функция открывает новое соединение с базой данных, используя параметры, полученные из строки `conninfo`.

Передаваемая строка может быть пустой. В этом случае используются все параметры по умолчанию. Она также может содержать одно или более значений параметров, разделённых пробелами, или URI. За подробностями обратитесь к Подразделу 32.1.1.

`PQsetdbLogin`

Создаёт новое подключение к серверу баз данных.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

Это предшественница функции `PQconnectdb` с фиксированным набором параметров. Она имеет такую же функциональность, за исключением того, что отсутствующие параметры всегда принимают значения по умолчанию. Подставьте `NULL` или пустую строку в качестве любого из фиксированных параметров, которые должны принять значения по умолчанию.

Если параметр *`dbName`* содержит знак `=` или имеет допустимый префикс URI для подключения, то он воспринимается в качестве строки *`conninfo`* точно таким же образом, как если бы он был передан функции `PQconnectdb`, а оставшиеся параметры применяются, как указано для `PQconnectdbParams`.

`pgtty` больше не используется, и любое переданное значение будет проигнорировано.

`PQsetdb`

Создаёт новое подключение к серверу баз данных.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

Это макрос, который вызывает `PQsetdbLogin` с нулевыми указателями в качестве значений параметров *`login`* и *`pwd`*. Обеспечивает обратную совместимость с очень старыми программами.

`PQconnectStartParams`
`PQconnectStart`
`PQconnectPoll`

Создают подключение к серверу баз данных неблокирующим способом.

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Три эти функции используются для того, чтобы открыть подключение к серверу баз данных таким образом, чтобы поток исполнения вашего приложения не был заблокирован при выполнении удалённой операции ввода-вывода в процессе подключения. Суть этого подхода в том, чтобы ожидание завершения операций ввода-вывода могло происходить в главном цикле приложения, а не внутри функций `PQconnectdbParams` или `PQconnectdb`, с тем, чтобы приложение могло управлять этой операцией параллельно с другой работой.

С помощью функции `PQconnectStartParams` подключение к базе данных выполняется, используя параметры, взятые из массивов `keywords` и `values`, а управление осуществляется с помощью `expand_dbname`, как описано выше для `PQconnectdbParams`.

С помощью функции `PQconnectStart` подключение к базе данных выполняется, используя параметры, взятые из строки `conninfo`, как описано выше для `PQconnectdb`.

Ни `PQconnectStartParams`, ни `PQconnectStart`, ни `PQconnectPoll` не заблокируются до тех пор, пока выполняется ряд ограничений:

- Параметр `hostaddr` должен использоваться так, чтобы для разрешения заданного имени не требовалось выполнять запросы DNS. Подробнее этот параметр описан в Подразделе 32.1.2.
- Если вы вызываете `PQtrace`, сделайте так, чтобы поток, в который выводится трассировочная информация, не заблокировался.
- Перед вызовом `PQconnectPoll` вы должны перевести сокет в соответствующее состояние, как описано ниже.

Чтобы начать неблокирующий запрос на подключение, вызовите `PQconnectStart` или `PQconnectStartParams`. Если результатом будет null, значит libpq не смогла выделить память для новой структуры `PGconn`. В противном случае возвращается действительный указатель `PGconn` (хотя он ещё не представляет установленное подключение к базе данных). Затем вызовите `PQstatus(conn)`. Если результатом будет `CONNECTION_BAD`, значит попытка подключения уже не будет успешной, возможно, из-за неверных параметров.

Если вызов `PQconnectStart` или `PQconnectStartParams` оказался успешным, теперь нужно опросить libpq для продолжения процедуры подключения. Вызовите `PQsocket(conn)` для получения дескриптора нижележащего сокета, через который устанавливается соединение. (Внимание: этот сокет может меняться от вызова к вызову `PQconnectPoll`.) Организуйте цикл таким образом: если `PQconnectPoll(conn)` при последнем вызове возвращает `PGRES_POLLING_READING`, ожидайте, пока сокет не окажется готовым для чтения (это покажет функция `select()`, `poll()` или подобная системная функция). Обратите внимание, что, если функция `PQsocketPoll` доступна в системе, она помогает уменьшить шаблонность кода путём абстрагирования функций `select(2)` или `poll(2)`. Затем снова вызовите `PQconnectPoll(conn)`. Если же `PQconnectPoll(conn)` при последнем вызове возвратила `PGRES_POLLING_WRITING`, дождитесь готовности сокета к записи, а затем снова вызовите `PQconnectPoll(conn)`. На первой итерации, то есть когда вы ещё не вызывали `PQconnectPoll`, реализуйте то же поведение, что и после получения `PGRES_POLLING_WRITING`. Продолжайте этот цикл, пока `PQconnectPoll(conn)` не выдаст значение `PGRES_POLLING_FAILED`, сигнализирующее об ошибке при установлении соединения, или `PGRES_POLLING_OK`, показывающее, что соединение установлено успешно.

В любое время в процессе подключения его состояние можно проверить, вызвав `PQstatus`. Если этот вызов возвратит `CONNECTION_BAD`, значит, процедура подключения завершилась сбоем; если вызов возвратит `CONNECTION_OK`, значит, соединение готово. Оба эти состояния можно определить на основе возвращаемого значения функции `PQconnectPoll`, описанной выше. Другие состояния могут также иметь место в течение (и только в течение) асинхронной процедуры подключения. Они показывают текущую стадию процедуры подключения и могут быть полезны, например, для предоставления обратной связи пользователю. Вот эти состояния:

`CONNECTION_STARTED`

Ожидание, пока соединение будет установлено.

`CONNECTION_MADE`

Соединение установлено; ожидание отправки.

`CONNECTION_AWAITING_RESPONSE`

Ожидание ответа от сервера.

`CONNECTION_AUTH_OK`

Аутентификация получена; ожидание завершения запуска серверной части.

`CONNECTION_SSL_STARTUP`

Согласование SSL-шифрования.

`CONNECTION_GSS_STARTUP`

Согласование GSS-шифрования.

`CONNECTION_CHECK_WRITABLE`

Проверка, можно ли через подключение выполнять пишущие транзакции.

`CONNECTION_CHECK_STANDBY`

Проверка, происходит ли подключение к серверу, находящемуся в режиме резерва.

`CONNECTION_CONSUME`

Прочтение всех оставшихся ответных сообщений через подключение.

Заметьте, что, хотя эти константы и сохранятся (для поддержания совместимости), приложение никогда не должно полагаться на то, что они появятся в каком-то конкретном порядке или вообще появятся, а также на то, что состояние всегда примет одно из этих документированных значений. Приложение может сделать что-то наподобие:

```
switch(PQstatus(conn))
{
        case CONNECTION_STARTED:
            feedback = "Подключение...";
            break;

        case CONNECTION_MADE:
            feedback = "Подключён к серверу...";
            break;
.
.
.
        default:
            feedback = "Подключение...";
}
```

Параметр подключения `connect_timeout` игнорируется, когда используется `PQconnectPoll`; именно приложение отвечает за принятие решения о том, является ли истекшее время чрезмерным. В противном случае вызов `PQconnectStart` с последующим вызовом `PQconnectPoll` в цикле будут эквивалентны вызову `PQconnectdb`.

Заметьте, что когда функция `PQconnectStart` или `PQconnectStartParams` возвращает ненулевой указатель, вы должны вызвать `PQfinish`, закончив его использование, чтобы освободить полученную структуру и все связанные с ней блоки памяти. Это нужно сделать, даже если попытка подключения оказалась неудачной или подключение не потребовалось.

`PQsocketPoll`

Опросить дескриптор нижележащего сокета, полученный с помощью `PQsocket`. Основное предназначение данной функции — перебор последовательностей соединений, как описано в документации `PQconnectStartParams`.

```
typedef pg_int64 pg_usec_time_t;

int PQsocketPoll(int sock, int forRead, int forWrite,
                 pg_usec_time_t end_time);
```

Данная функция опрашивает файловый дескриптор, по выбору — с тайм-аутом. При ненулевом значении параметра *`forRead`* функция завершит работу, когда сокет будет готов для чтения. При ненулевом значении параметра *`forWrite`* функция завершит работу, когда сокет будет готов для записи.

Параметр *`end_time`* задаёт тайм-аут — количество микросекунд, по истечении которого ожидание прекращается, в формате Unix (то есть `time_t`, умноженное на 1 миллион). Тайм-аут не ограничен при значении `-1` у параметра *`end_time`*. Тайм-аут завершается мгновенно (без блокировок) при значении `0` у параметра *`end_time`* (или любого значения до настоящего времени). Значения тайм-аута можно рассчитать, добавляя желаемое число микросекунд к результату работы функции `PQgetCurrentTimeUSec`. Обратите внимание, что другие системные вызовы выдают менее точные значения в микросекундах, поэтому реальная задержка может отличаться.

Функция возвращает значение больше `0` при соблюдении указанных условий, `0` при наступлении тайм-аута и `-1` при ошибке. Ошибку можно увидеть при проверке значения `errno(3)`. Если оба значения *`forRead`* и *`forWrite`* равны нулю, функция сразу выдаёт сообщение о наступлении тайм-аута.

В зависимости от платформы, функция `PQsocketPoll` реализована через `poll(2)` или `select(2)`. Более подробно информация представлена в описании `POLLIN` и `POLLOUT` функции `poll(2)`, а также параметров *`readfds`* и *`writefds`* функции `select(2)`.

`PQconndefaults`

Возвращает значения по умолчанию для параметров подключения.

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* Ключевое слово для данного параметра */
    char   *envvar;    /* Имя альтернативной переменной окружения */
    char   *compiled;  /* Альтернативное значение по умолчанию, назначенное при
 компиляции */
    char   *val;       /* Текущее значение параметра или NULL */
    char   *label;     /* Обозначение этого поля в диалоге подключения */
    char   *dispchar;  /* Показывает, как отображать это поле
                          в диалоге подключения. Значения следующие:
                          ""        Отображать введённое значение "как есть"
                          "*"       Поле пароля — скрывать значение
                          "D"       Параметр отладки — не показывать по умолчанию */
    int     dispsize;  /* Размер поля в символах для диалога */
} PQconninfoOption;
```

Возвращает массив параметров подключения. Он может использоваться для определения всех возможных параметров `PQconnectdb` и их текущих значений по умолчанию. Возвращаемое значение указывает на массив структур `PQconninfoOption`, который завершается элементом, имеющим нулевой указатель `keyword`. Если выделить память не удалось, то возвращается нулевой указатель. Обратите внимание, что текущие значения по умолчанию (поля `val`) будут зависеть от переменных среды и другого контекста. Отсутствующий или неверный файл служб будет просто проигнорирован. Вызывающие функции должны рассматривать данные параметров подключения как данные только для чтения.

После обработки массива параметров освободите память, передав его функции `PQconninfoFree`. Если этого не делать, то при каждом вызове функции `PQconndefaults` будет теряться небольшой объём памяти.

`PQconninfo`

Возвращает параметры подключения, используемые действующим соединением.

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

Возвращает массив параметров подключения. Он может использоваться для определения всех возможных параметров `PQconnectdb` и значений, которые были использованы для подключения к серверу. Возвращаемое значение указывает на массив структур `PQconninfoOption`, который завершается элементом, имеющим нулевой указатель `keyword`. Все замечания, приведённые выше для `PQconndefaults`, также справедливы и для результата `PQconninfo`.

`PQconninfoParse`

Возвращает разобранные параметры подключения, переданные в строке подключения.

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

Разбирает строку подключения и возвращает результирующие параметры в виде массива; возвращает `NULL`, если возникают проблемы при разборе строки подключения. Эту функцию можно использовать для извлечения параметров функции `PQconnectdb` из предоставленной строки подключения. Возвращаемое значение указывает на массив структур `PQconninfoOption`, который завершается элементом, имеющим нулевой указатель `keyword`.

Все разрешённые параметры будут присутствовать в результирующем массиве, но `PQconninfoOption` для любого параметра, не присутствующего в строке подключения, будет иметь значение `NULL` в поле `val`; значения по умолчанию не подставляются.

Если `errmsg` не равно `NULL`, тогда в случае успеха `*errmsg` присваивается `NULL`, а в противном случае -- адрес строки сообщения об ошибке, объясняющего проблему. Память для этой строки выделяет функция `malloc`. (Также возможна ситуация, когда `*errmsg` будет установлено в `NULL`, и при этом функция возвращает `NULL`. Это указывает на нехватку памяти.)

После обработки массива параметров освободите память, передав его функции `PQconninfoFree`. Если этого не делать, некоторое количество памяти будет теряться при каждом вызове `PQconninfoParse`. Если же произошла ошибка и `errmsg`, указывающий на строку сообщения об ошибке, не равен `NULL`, не забудьте освободить память с этой строкой, вызвав `PQfreemem`.

`PQfinish`

Закрывает соединение с сервером. Также освобождает память, используемую объектом `PGconn`.

```
void PQfinish(PGconn *conn);
```

Обратите внимание, что даже если попытка подключения к серверу потерпела неудачу (как показывает `PQstatus`), приложение все равно должно вызвать `PQfinish`, чтобы освободить память, используемую объектом `PGconn`. Указатель `PGconn` не должен использоваться повторно после того, как была вызвана функция `PQfinish`.

`PQreset`

Переустанавливает канал связи с сервером.

```
void PQreset(PGconn *conn);
```

Эта функция закроет подключение к серверу, а потом попытается установить новое подключение, используя все те же параметры, которые использовались прежде. Это может быть полезным для восстановления после ошибки, если работающее соединение было разорвано.

`PQresetStart`
`PQresetPoll`

Переустанавливает канал связи с сервером неблокирующим способом.

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Эти функции закроют подключение к серверу, а потом попытаются установить новое подключение, используя все те же параметры, которые использовались прежде. Это может быть полезным для восстановления после ошибки, если работающее соединение оказалось потерянным. Они отличаются от `PQreset` (см. выше) тем, что действуют неблокирующим способом. На эти функции налагаются те же ограничения, что и на `PQconnectStartParams`, `PQconnectStart` и `PQconnectPoll`.

Чтобы произвести переподключение, вызовите `PQresetStart`. Если она возвратит 0, значит при переподключении произошла ошибка. Если она возвратит 1, опросите результат операции, используя `PQresetPoll`, таким же образом, как это делается с помощью `PQconnectPoll` при обычном установления соединения.

`PQpingParams`

`PQpingParams` сообщает состояние сервера. Она принимает параметры подключения, идентичные тем, что получает функция `PQconnectdbParams`, описанная выше. Нет необходимости предоставлять корректные имя пользователя, пароль или имя базы данных, чтобы получить состояние сервера. Но если будут предоставлены некорректные значения, сервер зафиксирует в журнале неудачную попытку подключения.

```
PGPing PQpingParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);
```

Функция возвращает одно из следующих значений:

`PQPING_OK`

Сервер работает и, по-видимому, принимает подключения.

`PQPING_REJECT`

Сервер работает, но находится в состоянии, которое запрещает подключения (запуск, завершение работы или восстановление после аварийного отказа).

`PQPING_NO_RESPONSE`

Контакт с сервером не удался. Это может указывать на то, что сервер не запущен или что-то не в порядке с параметрами данного подключения (например, неверный номер порта), или имеет место проблема с возможностью соединения по сети (например, брандмауэр блокирует запрос на подключение).

`PQPING_NO_ATTEMPT`

Никакой попытки установить контакт с сервером сделано не было, поскольку предоставленные параметры были явно некорректными, или имела место какая-то проблема на стороне клиента (например, нехватка памяти).

`PQping`

`PQping` сообщает состояние сервера. Она принимает параметры подключения, идентичные тем, что получает функция `PQconnectdb`, описанная выше. Нет необходимости предоставлять корректные имя пользователя, пароль или имя базы данных, чтобы получить состояние сервера. Но если будут предоставлены некорректные значения, сервер зафиксирует в журнале неудачную попытку подключения.

```
PGPing PQping(const char *conninfo);
```

Возвращаемые значения такие же, как и для `PQpingParams`.

`PQsetSSLKeyPassHook_OpenSSL`

`PQsetSSLKeyPassHook_OpenSSL` позволяет приложению переопределить реализованный в `libpq` стандартный вариант обработки файлов с зашифрованными ключами клиентских сертификатов, используя sslpassword или интерактивное приглашение.

```
void PQsetSSLKeyPassHook_OpenSSL(PQsslKeyPassHook_OpenSSL_type hook);
```

Приложение передаёт указатель на функцию-обработчик со следующей сигнатурой:

```
int callback_fn(char *buf, int size, PGconn *conn);
```

Эту функцию `libpq` будет вызывать *вместо* своего стандартного обработчика `PQdefaultSSLKeyPassHook_OpenSSL`. Данная функция должна получить пароль для ключа и скопировать его в результирующий буфер `buf` размера `size`. Строка в `buf` должна завершаться нулём. В результате эта функция должна выдать длину пароля, сохранённого в `buf`, не считая завершающего нуля. В случае ошибки она должна установить `buf[0] = '\0'` и выдать 0.

Если пользователь задал размещение ключа явно, заданный путь будет передан при вызове этого обработчика в `conn->sslkey`. Это поле будет пустым, если используется путь к ключу по умолчанию. Что касается ключей, специфичных для модулей OpenSSL, для них модули могут по своему усмотрению получать пароль через стандартный обработчик OpenSSL или через свой собственный.

Пользовательский обработчик может полностью переопределить функцию `PQdefaultSSLKeyPassHook_OpenSSL`, либо делегировать ей необрабатываемые им случаи, либо сначала вызывать её и предпринимать какие-то другие действия, если она возвратит 0.

Этот обработчик *не должен* нарушать обычный ход выполнения, выбрасывая исключения, вызывая `longjmp(...)` и т. п. Он должен завершиться нормально.

`PQgetSSLKeyPassHook_OpenSSL`

`PQgetSSLKeyPassHook_OpenSSL` возвращает текущий обработчик пароля для ключа клиентского сертификата либо `NULL`, если такой обработчик не установлен.

```
PQsslKeyPassHook_OpenSSL_type PQgetSSLKeyPassHook_OpenSSL(void);
```

32.1.1. Строки параметров подключения

Ряд функций libpq разбирают заданную пользователем строку для извлечения параметров подключения. Есть два принятых формата этих строк: простая строка ключ/значение и URI. Строки URI в основном соответствуют *RFC 3986*, но могут содержать также строки подключения с несколькими узлами, как описано ниже.

32.1.1.1. Строки параметров подключения вида «ключ/значение»

Согласно формату ключ/значение, установка каждого параметра выполняется в форме `ключ = значение`, с пробелами между параметрами. Пробелы вокруг знака равенства не являются обязательными. Для записи пустого значения или значения, содержащего пробелы, заключите его в одинарные кавычки, например, `keyword = 'a value'`. Одинарные кавычки и символы обратной косой черты внутри значения нужно обязательно экранировать с помощью символа обратной косой черты, т. е., `\'` и `\\`.

Пример:

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

Ключевые слова-параметры, распознаваемые в настоящее время, приведены в Подразделе 32.1.2.

32.1.1.2. URI для подключения

Основная форма URI подключения:

```
postgresql://[пользователь@][сервер][/база_данных][?указание_параметра]

где пользователь:

имя_пользователя[:пароль]

и сервер:
```

```
[узел][:порт][,...]
```

и *указание_параметра*:

```
имя=значение[&...]
```

В качестве обозначения схемы URI может использоваться `postgresql://` или `postgres://`. Остальные части URI являются необязательными. В следующих примерах показан допустимый синтаксис URI:

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
postgresql://host1:123,host2:456/somedb?target_session_attrs=any&application_name=myapp
```

Значения, которые обычно задаются в иерархической части URI, также можно задать в именованных параметрах. Например:

```
postgresql:///mydb?host=localhost&port=5433
```

Все именованные параметры должны соответствовать ключевым словам, перечисленным в Подраздел 32.1.2, за исключением того, что вхождения `ssl=true` заменяются на `sslmode=require` для совместимости с URI-строками JDBC.

Если URI подключения содержит в какой-либо из частей символы со специальным значением, он должен кодироваться в *формате с процентами*. Например, так выглядит URI, в котором знак равно (=) заменён на `%3D`, а знак пробела — на `%20`:

```
postgresql://user@localhost:5433/mydb?options=-c%20synchronous_commit%3Doff
```

Сервер можно представить либо сетевым именем, либо IP-адресом. При использовании протокола IPv6 нужно заключить адрес в квадратные скобки:

```
postgresql://[2001:db8::1234]/database
```

Часть `host` интерпретируется в соответствии с описанием параметра host. Так, если эта часть строки пуста или выглядит как абсолютный путь, выбирается соединение через Unix-сокеты, в противном случае инициируется соединение по TCP/IP. Учтите однако, что символ косой черты в иерархической части URI является зарезервированным. Поэтому, чтобы указать нестандартный каталог Unix-сокета, нужно поступить одним из двух способов: не задавать эту часть в URI и указать сервер в именованном параметре либо закодировать путь в части `host` с процентами:

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

В одном URI можно задать несколько компонентов узлов, каждый с необязательным указанием порта. URI вида `postgresql://host1:port1,host2:port2,host3:port3/` равнозначно строке подключения вида `host=host1,host2,host3 port=port1,port2,port3`. Как описывается ниже, эти узлы будут перебираться по очереди, пока не будет установлено подключение.

32.1.1.3. Указание нескольких узлов

В строке подключения можно задать несколько узлов, к которым клиент будет пытаться подключиться в заданном порядке. Параметры `host`, `hostaddr` и `port` в формате ключ/значение принимают списки значений, разделённых запятыми. В каждом определяемом параметре должно содержаться одинаковое число элементов, чтобы, например, первый элемент `hostaddr` соответствовал первому элементу `host`, второй — второму `host` и так далее. Исключение составляет `port` — если этот параметр содержит только один элемент, он применяется ко всем узлам.

В формате URI внутри компонента `host` можно указать несколько пар `host:port`, разделённых запятыми.

В любом формате одно имя узла может переводиться в несколько сетевых адресов. Например, часто бывает, что один узел имеет и адрес IPv4, и адрес IPv6.

Когда задаются несколько узлов или когда одно имя узла переводится в несколько адресов, все узлы и адреса перебираются по порядку, пока подключение не будет установлено. Если ни один из адресов не будет доступен, произойдёт сбой подключения. Если подключение устанавливается успешно, но происходит ошибка аутентификации, остальные узлы в списке не перебираются.

Если используется файл паролей, в нём можно задать разные пароли для разных узлов. Все остальные параметры подключения будут одинаковыми для всех узлов; например, нельзя задать для разных узлов различные имена пользователей.

32.1.2. Ключевые слова-параметры

Ключевые слова-параметры, распознаваемые в настоящее время, следующие:

`host`

Имя узла для подключения. Если это имя выглядит как указание абсолютного пути, выбирается подключение через Unix-сокет, а не через TCP/IP, и данное значение определяет имя каталога, содержащего файл сокета. (В Unix-системах абсолютный путь начинается с косой черты, а в Windows абсолютные пути также могут начинаться с буквы диска.) Если имя узла начинается с `@`, оно воспринимается как Unix-сокет в абстрактном пространстве имён (в настоящее время поддерживается в Linux и Windows). По умолчанию, если параметр `host` отсутствует или пуст, выполняется подключение к Unix-сокету в `/tmp` (или в том каталоге сокетов, который был задан при сборке Postgres Pro). В Windows по умолчанию выполняется подключение к `localhost`.

Также принимается разделённый запятыми список имён узлов; при этом данные имена будут перебираться по порядку. Для пустых элементов списка применяется поведение по умолчанию, описанное выше. За подробностями обратитесь к Подразделу 32.1.1.3.

`hostaddr`

Числовой IP-адрес узла для подключения. Он должен быть представлен в стандартном формате адресов IPv4, например, `172.28.40.9`. Если ваша машина поддерживает IPv6, вы можете использовать и адреса IPv6. Если в этом параметре передана непустая строка, для подключения всегда используется TCP/IP. В отсутствие этого параметра целевой IP-адрес будет получен из имени, заданного в параметре `host`, либо если в `host` задан IP-адрес, будет использоваться непосредственно он.

Использование `hostaddr` позволяет приложению обойтись без разрешения имён, что может быть важно для приложений с жёсткими временными ограничениями. Однако имя узла требуется для методов аутентификации GSSAPI или SSPI, а также для проверки полномочий на основе SSL-сертификатов в режиме `verify-full`. Применяются следующие правила:

- Если адрес `host` задаётся без `hostaddr`, осуществляется разрешение имени. (При использовании `PQconnectPoll` разрешение производится, когда `PQconnectPoll` рассматривает это имя в первый раз, и может заблокировать `PQconnectPoll` на неопределённое время.)
- Если указан `hostaddr`, а `host` не указан, тогда значение `hostaddr` даёт сетевой адрес сервера. Попытка подключения завершится неудачей, если метод аутентификации требует наличия имени узла.
- Если указаны как `host`, так и `hostaddr`, тогда значение `hostaddr` даёт сетевой адрес сервера, а значение `host` игнорируется, если только метод аутентификации его не потребует. В таком случае оно будет использоваться в качестве имени узла.

Заметьте, что аутентификация может завершиться неудачей, если `host` не является именем сервера с сетевым адресом `hostaddr`. Также заметьте, что когда указывается и `host`, и `hostaddr`, соединение в файле паролей идентифицируется по значению `host` (см. Раздел 32.16).

Также принимается разделённый запятыми список значений `hostaddr`, при этом данные узлы будут перебираться по порядку. Вместо пустого элемента в этом списке будет подставлено имя соответствующего узла или, если и оно не определено, имя узла по умолчанию. За подробностями обратитесь к Подразделу 32.1.1.3.

Если не указаны ни имя узла, ни его адрес, libpq будет производить подключение, используя локальный Unix-сокет; в Windows она будет пытаться подключиться к `localhost`.

`port`

Номер порта, к которому нужно подключаться на сервере, либо расширение имени файла сокета для соединений через Unix-сокеты. Если в параметрах `host` или `hostaddr` задано несколько серверов, в данном параметре может задаваться через запятую список портов такой же длины, либо может указываться один номер порта для всех узлов. Пустая строка или пустой элемент в списке через запятую воспринимается как номер порта по умолчанию, установленный при сборке Postgres Pro.

`dbname`

Имя базы данных. По умолчанию оно совпадает с именем пользователя. В определённых контекстах это значение проверяется на соответствие расширенным форматам; см. Подраздел 32.1.1 для получения подробной информации.

`user`

Имя пользователя Postgres Pro, используемое для подключения. По умолчанию используется то же имя, которое имеет в операционной системе пользователь, от лица которого выполняется приложение.

`password`

Пароль, используемый в случае, когда сервер требует аутентификации по паролю.

`passfile`

Задаёт имя файла, в котором будут храниться пароли (см. Раздел 32.16). По умолчанию это `~/.pgpass` или `%APPDATA%\postgresql\pgpass.conf` в Microsoft Windows. (Отсутствие этого файла не считается ошибкой.)

`require_auth`

Указывает метод аутентификации, который клиент требует от сервера. Если сервер не использует требуемый метод для аутентификации клиента или если обмен сообщениями аутентификации со стороны сервера не завершён, соединение не будет установлено. Также в качестве значения можно указать список методов, разделённых запятыми, из которых сервер должен использовать только один, чтобы соединение было успешным. По умолчанию принимается любой метод аутентификации, и сервер даже может вообще не выполнять её.

Методы можно указать с оператором отрицания `!`, и в этом случае сервер *не* должен пытаться использовать указанный метод; допускается любой другой метод, и сервер может вообще не аутентифицировать клиента. Если указан список методов, разделённых запятыми, сервер не может использовать *любой* из перечисленных методов с оператором отрицания. Формы с отрицанием и без не могут указываться вместе.

Последняя особенность метода `none` требует, чтобы сервер не использовал аутентификацию. (Его также можно указать с оператором отрицания, чтобы потребовать некоторую форму аутентификации.)

Могут быть указаны следующие методы:

`password`

Сервер должен запрашивать аутентификацию по паролю, заданному открытым текстом.

`md5`

Сервер должен запрашивать аутентификацию по паролю, защищённому MD5.

`gss`

Сервер должен либо запросить подтверждение связи Kerberos через GSSAPI, либо установить канал с GSS шифрованием (см. также gssencmode).

`sspi`

Сервер должен запрашивать аутентификацию Windows SSPI.

`scram-sha-256`

Сервер должен успешно завершить обмен сообщениями по схеме аутентификации SCRAM-SHA-256 с клиентом.

`none`

Сервер не должен запрашивать у клиента обмен по схеме аутентификации. (Это не запрещает аутентификацию сертификата клиента через TLS или аутентификацию GSS с шифрованием данных на транспортном уровне.)

`channel_binding`

Этот параметр определяет режим использования связывания каналов клиентом. Вариант `require` означает, что для соединения должно задействоваться связывание каналов, `prefer` — клиент будет выбирать связывание, если оно поддерживается, а `disable` предотвращает использование этого механизма. По умолчанию подразумевается вариант `prefer`, если Postgres Pro собирается с поддержкой SSL; в противном случае — `disable`.

Механизм связывания каналов позволяет серверу подтвердить свою подлинность клиенту. Он работает только для подключений поверх SSL с серверами Postgres Pro версии 11 и новее, когда используется метод аутентификации `SCRAM`.

`connect_timeout`

Максимальное время ожидания подключения (задаётся десятичным целым числом, например: `10`). При нуле, отрицательном или не заданном значении ожидание будет бесконечным. Этот тайм-аут применяется для каждого отдельного IP-адреса или имени сервера. Например, если вы зададите адреса двух серверов и значение `connect_timeout` будет равно 5, тайм-аут при неудачной попытке подключения к каждому серверу истечёт через 5 секунд, а общее время ожидания подключения может достигать 10 секунд.

`client_encoding`

Этим устанавливается конфигурационный параметр `client_encoding` для данного подключения. В дополнение к значениям, которые принимает соответствующий параметр сервера, вы можете использовать значение `auto`. В этом случае правильная кодировка определяется на основе текущей локали на стороне клиента (в системах Unix это переменная системного окружения `LC_CTYPE`).

`options`

Задаёт параметры командной строки, которые будут отправлены серверу при установлении соединения. Например, значение `-c geqo=off` или `--geqo=off` установит для параметра сеанса `geqo` значение `off`. Пробелы в этой строке считаются разделяющими аргументы командной строки, если только перед ними не стоит обратная косая черта (`\`); чтобы записать собственно обратную косую черту, её нужно продублировать (`\\`). Подробное описание возможных параметров можно найти в Главе 18.

`application_name`

Устанавливает значение для конфигурационного параметра application_name.

`fallback_application_name`

Устанавливает альтернативное значение для конфигурационного параметра application_name. Это значение будет использоваться, если для параметра `application_name` не было передано никакого значения с помощью параметров подключения или переменной системного окружения `PGAPPNAME`. Задание альтернативного имени полезно для универсальных программ-утилит, которые желают установить имя приложения по умолчанию, но позволяют пользователю изменить его.

`keepalives`

Управляет использованием сообщений keepalive протокола TCP на стороне клиента. Значение по умолчанию равно 1, что означает использование сообщений. Вы можете изменить его на 0, если эти сообщения не нужны. Для соединений, установленных через Unix-сокеты, этот параметр игнорируется.

`keepalives_idle`

Управляет длительностью периода отсутствия активности, выраженного числом секунд, по истечении которого TCP должен отправить сообщение keepalive серверу. При значении 0 действует системная величина. Этот параметр игнорируется для соединений, установленных через Unix-сокеты, или если сообщения keepalive отключены. Он поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPIDLE` или равнозначный, и в Windows; в других системах он не оказывает влияния.

`keepalives_interval`

Задаёт количество секунд, по прошествии которых сообщение keepalive протокола TCP, получение которого не подтверждено сервером, должно быть отправлено повторно. При значении 0 действует системная величина. Этот параметр игнорируется для соединений, установленных через Unix-сокеты, или если сообщения keepalive отключены. Он поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPINTVL` или равнозначный, и в Windows; в других системах он не оказывает влияния.

`keepalives_count`

Задаёт количество сообщений keepalive протокола TCP, которые могут быть потеряны, прежде чем соединение клиента с сервером будет признано неработающим. Нулевое значение этого параметра указывает, что будет использоваться системное значение по умолчанию. Этот параметр игнорируется для соединений, установленных через Unix-сокеты, или если сообщения keepalive отключены. Он поддерживается только в системах, воспринимающих параметр сокета `TCP_KEEPCNT` или равнозначный; в других системах он не оказывает влияния.

`tcp_user_timeout`

Управляет длительностью интервала (в миллисекундах), в течение которого данные могут оставаться неподтверждёнными, прежде чем соединение будет принудительно закрыто. При значении 0 действует системная величина. Этот параметр игнорируется для соединений, установленных через Unix-сокеты. Он поддерживается только в системах, воспринимающих параметр сокета `TCP_USER_TIMEOUT`; в других системах он не оказывает влияния.

`replication`

Этот параметр определяет, должно ли подключение использовать протокол репликации вместо обычного протокола. Этот вариант используют внутри механизмы репликации Postgres Pro и такие средства, как pg_basebackup, но он может также применяться и сторонними приложениями. За подробным описанием протокола репликации обратитесь к Разделу 53.4.

Поддерживаются следующие значения этого параметра, без учёта регистра:

`true`, `on`, `yes`, `1`

Подключение осуществляется в режиме физической репликации.

`database`

Подключение осуществляется в режиме логической репликации, целевая база данных задаётся параметром `dbname`.

`false`, `off`, `no`, `0`

Подключение выполняется в обычном режиме; это поведение по умолчанию.

В режиме физической или логической репликации может использоваться только простой протокол запросов.

`gssencmode`

Этот параметр определяет, будет ли согласовываться с сервером защищённое GSS соединение по протоколу TCP/IP, и если да, то в какой очередности. Всего предусмотрено три режима:

`disable`

пытаться установить только соединение без шифрования GSSAPI

`prefer` (по умолчанию)

если уже имеются учётные данные GSSAPI (в кеше учётных данных), сначала попытаться установить соединение с шифрованием GSSAPI; если эта попытка не удастся или подходящих данных нет, попробовать соединение без шифрования GSSAPI. Это значение по умолчанию, если Postgres Pro был скомпилирован с поддержкой GSSAPI.

`require`

пытаться установить только соединение с шифрованием GSSAPI

`sslmode` игнорируется при использовании Unix-сокетов. Если Postgres Pro скомпилирован без поддержки GSSAPI, использование варианта `require` приведёт к ошибке, в то время как параметр `prefer` будет принят, но libpq в действительности не будет пытаться установить зашифрованное GSSAPI соединение.

`sslmode`

Этот параметр определяет, будет ли согласовываться с сервером защищённое SSL-соединение по протоколу TCP/IP, и если да, то в какой очередности. Всего предусмотрено шесть режимов:

`disable`

следует пытаться установить только соединение без использования SSL

`allow`

сначала следует попытаться установить соединение без использования SSL; если попытка будет неудачной, нужно попытаться установить SSL-соединение

`prefer` (по умолчанию)

сначала следует попытаться установить SSL-соединение; если попытка будет неудачной, нужно попытаться установить соединение без использования SSL

`require`

следует попытаться установить только SSL-соединение. Если присутствует файл корневого центра сертификации, то нужно верифицировать сертификат таким же способом, как будто был указан параметр `verify-ca`

`verify-ca`

следует попытаться установить только SSL-соединение, при этом проконтролировать, чтобы сертификат сервера был выпущен доверенным центром сертификации (CA) `verify-full`

следует попытаться установить только SSL-соединение, при этом проконтролировать, чтобы сертификат сервера был выпущен доверенным центром сертификации (CA) и чтобы имя запрошенного сервера соответствовало имени в сертификате

В Разделе 32.19 приведено подробное описание работы этих режимов.

`sslmode` игнорируется при использовании Unix-сокетов. Если Postgres Pro скомпилирован без поддержки SSL, использование параметров `require`, `verify-ca` или `verify-full` приведёт к ошибке, в то время как параметры `allow` и `prefer` будут приняты, но libpq в действительности не будет пытаться установить SSL-соединение.

Заметьте, что при возможности использовать шифрование GSSAPI, оно будет предпочитаться шифрованию SSL вне зависимости от значения `sslmode`. Чтобы принудительно использовать SSL в окружении, где имеется рабочая инфраструктура GSSAPI (например, сервер Kerberos), дополнительно установите для `gssencmode` значение `disable`.

`requiressl`

Использовать этот параметр не рекомендуется, в качестве замены предлагается установить `sslmode`.

Если установлено значение 1, то требуется SSL-соединение с сервером (это эквивалентно `sslmode` `require`). libpq в таком случае откажется подключаться, если сервер не принимает SSL-соединений. Если установлено значение 0 (по умолчанию), тогда libpq будет согласовывать тип подключения с сервером (эквивалентно `sslmode` `prefer`). Этот параметр доступен, если только Postgres Pro скомпилирован с поддержкой SSL.

`sslnegotiation`

Данный параметр управляет согласованием шифрования по протоколу SSL с сервером (при использовании SSL). В режиме по умолчанию `postgres` клиент узнаёт у сервера, поддерживает ли тот SSL. В режиме `direct` клиент начинает стандартный обмен сообщениями по SSL сразу же после установления соединения по TCP/IP. Самым гибким является традиционный протокол согласований Postgres Pro, обладающий разными серверными конфигурациями. Если известно, что сервер поддерживает прямые подключения по SSL, то этот способ уменьшает количество дополнительных обращений, тем самым снижая задержку соединения, и позволяет использовать сетевое оборудование с поддержкой SSL, совместимое с другими протоколами. Режим `direct` появился в 17-й версии Postgres Pro.

`postgres`

запустить протокол согласования Postgres Pro. Это режим по умолчанию, если параметр не задан.

`direct`

начать обмен сообщениями по SSL сразу же после установления соединения по TCP/IP. Данный режим возможен только при значении sslmode=require или более строгом значении, поскольку значение с менее строгой проверкой может привести к возврату на незашифрованную аутентификацию, если сервер не поддерживает прямой обмен сообщениями по SSL.

`sslcompression`

Если установлено значение 1, данные, передаваемые через SSL-соединения, будут сжиматься. Если установлено значение 0 (по умолчанию), сжатие будет отключено. Этот параметр игнорируется, если установлено подключение без SSL.

Сжатие SSL в настоящее время считается небезопасным, и использовать его уже не рекомендуется. В OpenSSL 1.1.0 сжатие отключено по умолчанию, к тому же во многих дистрибутивах оно отключается и с более ранними версиями. А если сервер не поддерживает сжатие, включение этого параметра не окажет никакого влияния. В Postgres Pro 14 сжатие полностью отключено на сервере.

Если вопросы безопасности не стоят на первом месте, сжатие может ускорить передачу данных, когда узким местом является сеть. Отключение сжатия может улучшить время отклика и пропускную способность, если ограничивающим фактором является производительность CPU.

`sslcert`

Этот параметр предписывает имя файла для SSL-сертификата клиента, заменяющего файл по умолчанию `~/.postgresql/postgresql.crt`. Этот параметр игнорируется, если SSL-подключение не выполнено.

`sslkey`

Этот параметр предписывает местоположение секретного ключа, используемого для сертификата клиента. Он может либо указывать имя файла, которое будет использоваться вместо имени по умолчанию `~/.postgresql/postgresql.key`, либо он может указывать ключ, полученный от внешнего «криптомодуля» (криптомодули — это загружаемые модули OpenSSL). Спецификация внешнего криптомодуля должна состоять из имени модуля и ключевого идентификатора, зависящего от конкретного модуля, разделённых двоеточием. Этот параметр игнорируется, если SSL-подключение не выполнено.

`sslpassword`

Этот параметр задаёт пароль для секретного ключа, указанного в `sslkey`, что позволяет хранить на диске закрытые ключи клиентских сертификатов в зашифрованном виде, даже когда применять интерактивный ввод пароля непрактично.

Когда для этого параметра задаётся непустое значение, приглашение OpenSSL `Enter PEM pass phrase:` (Введите пароль для PEM:) не выводится. По умолчанию это приглашение выдаётся, если в `libpq` передаётся зашифрованный ключ клиентского сертификата.

Если ключ не зашифрован, этот параметр игнорируется. Данный параметр не действует для ключей, которыми оперируют дополнительные модули OpenSSL, если только модуль не пользуется представленной в OpenSSL функцией-обработчиком запроса пароля.

Для этого параметра нет соответствующей переменной окружения, а также не реализована возможность задать его в `.pgpass`. Однако его можно задать в определении свойств подключения в файле служб. Пользователям, которых не устраивает такой простой способ задания пароля, следует использовать специальные модули и средства OpenSSL, например PKCS#11 или криптографические USB-устройства.

`sslcertmode`

Этот параметр определяет, может ли сертификат клиента быть отправлен серверу и должен ли сервер запрашивать его. Всего предусмотрено три режима:

`disable`

Сертификат клиента не отправляется, даже если он доступен (расположен по умолчанию или указан в sslcert).

`allow` (по умолчанию)

Сертификат может быть отправлен, если сервер запрашивает его и клиент может его отправить.

`require`

Сервер *должен* запрашивать сертификат. Установить подключение не удастся, если клиент не отправляет сертификат, а сервер всё равно успешно аутентифицирует клиента.

Примечание

Установка `sslcertmode=require` не повышает безопасность, поскольку нет гарантии, что сервер правильно проверяет сертификат. Серверы Postgres Pro обычно запрашивают сертификаты TLS от клиентов независимо от того, проверяют они их или нет. Этот параметр может быть полезен при устранении неполадок в более сложных схемах TLS.

`sslrootcert`

Этот параметр указывает имя файла, содержащего SSL-сертификаты, выданные Центром сертификации (CA). Если файл существует, сертификат сервера будет проверен на предмет его подписания одним из этих центров. Имя по умолчанию — `~/.postgresql/root.crt`.

Вместо этого можно указать специальное значение `system`, и в этом случае будут загружены доверенные сертификаты корневых ЦС, предоставляемые реализацией SSL. Точное расположение этих корневых сертификатов зависит от реализации SSL и платформы. В частности, для OpenSSL расположение можно уточнить в переменных окружения `SSL_CERT_DIR` и `SSL_CERT_FILE`.

Примечание

При использовании `sslrootcert=system` значение `sslmode`, заданное по умолчанию, меняется на `verify-full`, и любое значение с менее строгой проверкой приведёт к ошибке. В большинстве случаев нетрудно получить доверенный для системы сертификат по имени управляемого узла, что делает `verify-ca` и все более слабые режимы бесполезными.

Если указать файл сертификата с именем `system`, оно будет распознаваться как особое значение `system`, поэтому при наличии файла с таким именем следует задать альтернативный путь, например `sslrootcert=./system`.

`sslcrl`

Этот параметр указывает имя файла, содержащего список отозванных серверных сертификатов (CRL) для SSL. Сертификаты, перечисленные в этом файле, если он существует, будут отвергаться при попытке установить подлинность сертификата сервера. Если ни sslcrl, ни sslcrldir не заданы, используется файл `~/.postgresql/root.crl`.

`sslcrldir`

Этот параметр указывает имя каталога, содержащего список отозванных серверных сертификатов (CRL) для SSL. Сертификаты, перечисленные в файлах данного каталога, если он существует, будут считаться неприемлемыми при проверке подлинности сертификата сервера.

Каталог необходимо подготовить с помощью команды OpenSSL `openssl rehash` или `c_rehash`. За подробностями обратитесь к документации OpenSSL.

Параметры `sslcrl` и `sslcrldir` можно указывать вместе.

`sslsni`

Со значением, равным 1 (по умолчанию), libpq включает расширение TLS «Server Name Indication» (SNI) для подключений с поддержкой SSL. При значении, равном 0, данное расширение отключается.

Указание имени сервера (SNI) может использоваться прокси-серверами с поддержкой SSL для маршрутизации соединений без расшифровывания потока SSL. (Обратите внимание, что если прокси-сервер не распознаёт пакеты инициализации протокола Postgres Pro, для `sslnegotiation` необходимо задать режим `direct`.) Однако при включении SNI имя целевого сервера передаётся по сети в открытом виде, что в некоторых случаях может быть нежелательно.

`requirepeer`

Этот параметр указывает имя пользователя операционной системы, предназначенное для сервера, например, `requirepeer=postgres`. При создании подключения через Unix-сокет, если этот параметр установлен, клиент проверяет в самом начале процедуры подключения, что серверный процесс запущен от имени указанного пользователя; если это не так, соединение аварийно прерывается с ошибкой. Этот параметр можно использовать, чтобы обеспечить аутентификацию сервера, подобную той, которая доступна с помощью SSL-сертификатов при соединениях по протоколу TCP/IP. (Заметьте, что если Unix-сокет находится в каталоге `/tmp` или в другом каталоге, запись в который разрешена всем пользователям, тогда любой пользователь сможет запустить сервер, прослушивающий сокет в том каталоге. Используйте этот параметр, чтобы гарантировать, что вы подключены к серверу, запущенному доверенным пользователем.) Он поддерживается только на платформах, для которых реализован метод аутентификации `peer`; см. Раздел 19.9.

`ssl_min_protocol_version`

Этот параметр определяет, с какой минимальной версией протокола SSL/TLS может быть установлено подключение. Допустимые значения: `TLSv1`, `TLSv1.1`, `TLSv1.2` и `TLSv1.3`. Набор поддерживаемых протоколов зависит от используемой версии OpenSSL; старые версии могут не поддерживать последние протоколы. По умолчанию минимальной считается версия `TLSv1.2`, что соответствует рекомендациям, актуальным в индустрии на момент написания этой документации.

`ssl_max_protocol_version`

Этот параметр определяет, с какой максимальной версией протокола SSL/TLS может быть установлено подключение. Допустимые значения: `TLSv1`, `TLSv1.1`, `TLSv1.2` и `TLSv1.3`. Набор поддерживаемых протоколов зависит от используемой версии OpenSSL; старые версии могут не поддерживать последние протоколы. Если этот параметр не задан, при подключении будет действовать ограничение версии сверху, установленное на стороне сервера (при наличии такового). Возможность ограничения максимальной версии протокола полезна прежде всего при тестировании или когда какой-либо компонент работает с новым протоколом некорректно.

`krbsrvname`

Имя службы Kerberos, которое будет использоваться при аутентификации GSSAPI. Чтобы аутентификация Kerberos прошла успешна, оно должно соответствовать имени службы, заданному в конфигурации сервера. (См. также Раздел 19.6.) По умолчанию обычно выбирается имя `postgres`, но его можно сменить при сборке Postgres Pro, передав ключ `--with-krb-srvnam` скрипту configure. В большинстве случаев менять этот параметр нет необходимости. Для некоторых реализаций Kerberos может требоваться другое имя службы, например, Microsoft Active Directory требует, чтобы имя службы задавалось в верхнем регистре (`POSTGRES`).

`gsslib`

Библиотека GSS, предназначенная для использования при аутентификации на основе GSSAPI. В настоящее время это действует только в сборках для Windows, поддерживающих одновременно и GSSAPI, и SSPI. Значение `gssapi` в таких сборках позволяет указать, что libpq должна использовать для аутентификации библиотеку GSSAPI, а не подразумеваемую по умолчанию SSPI.

`gssdelegation`

Переслать (делегировать) учётные данные GSS серверу. По умолчанию используется значение `0`. Это означает, что учётные данные не будут пересылаться серверу. Установите для этого параметра значение `1`, чтобы учётные данные пересылались, когда это возможно.

`service`

Имя сервиса, используемое для задания дополнительных параметров. Оно указывает имя сервиса в файле `pg_service.conf`, который содержит дополнительные параметры подключения. Это позволяет приложениям указывать только имя сервиса, поскольку параметры подключения могут поддерживаться централизованно. См. Раздел 32.17.

`target_session_attrs`

Этот параметр может накладывать ограничения на свойства сеансов, которые будут считаться приемлемыми. Обычно он используется вместе с указанием имён нескольких серверов и позволяет выбрать из них первый подходящий вариант. Есть шесть режимов:

`any` (по умолчанию)

любое успешное соединение приемлемо

`read-write`

сеанс должен принимать транзакции чтения-записи по умолчанию (то есть сервер не должен находиться в режиме горячего резерва, а у параметра `default_transaction_read_only` должно быть значение `off`)

`read-only`

сеанс не должен принимать транзакции чтения-записи по умолчанию (противоположный случай)

`primary`

сервер не должен находиться в режиме горячего резерва

`standby`

сервер должен находиться в режиме горячего резерва

`prefer-standby`

сначала пытаться найти резервный сервер, но если ни один из перечисленных серверов не является резервным, попробовать снова в режиме `any`

`load_balance_hosts`

Управляет порядком, в котором клиент пытается подключиться к доступным узлам и адресам. Если попытка подключения окажется успешной, клиент не будет пытаться подключиться к другим узлам и адресам. Этот параметр обычно используется вместе с несколькими именами узлов или записью DNS, которая возвращает несколько IP-адресов. Его также можно использовать вместе с target_session_attrs, например для балансировки нагрузки только между резервными серверами. После успешного подключения все последующие запросы по возвращённому подключению будут отправлены на один и тот же сервер. В настоящее время существует два режима:

`disable` (по умолчанию)

Балансировка нагрузки между узлами не выполняется. Узлы перебираются по очереди, а адреса — в том порядке, в котором они получены от DNS или файлов узлов.

`random`

Узлы и адреса перебираются в случайном порядке. Это значение наиболее полезно при одновременном открытии нескольких соединений, возможно с разных компьютеров. Таким образом, нагрузка может распределяться между соединениями нескольких серверов Postgres Pro.

Хотя случайная балансировка нагрузки по своей природе почти никогда не приводит к полностью равномерному распределению, статистически она довольно близка к этому. Важно то, что этот алгоритм использует два уровня случайного выбора: сначала узлы разрешаются в случайном порядке, затем перед разрешением следующего узла все разрешённые адреса текущего узла перебираются в случайном порядке. Такое поведение может сильно влиять на количество подключений, которые получает каждый узел в определённых случаях, например, когда некоторые узлы разрешают больше адресов, чем другие. Но такое влияние можно использовать и намеренно. Например, чтобы увеличить количество подключений к более крупному серверу, можно несколько раз указать его имя в соответствующей строке.

При использовании этого значения рекомендуется также выбрать подходящее значение для connect_timeout, потому что в таком случае, если один из узлов, используемых для балансировки нагрузки, не отвечает, будет использоваться новый узел.

.

32.2. Функции, описывающие текущее состояние подключения

Эти функции могут использоваться для опроса состояния объекта, описывающего существующее подключение к базе данных.

Подсказка

Разработчики приложений на основе libpq должны тщательно поддерживать абстракцию `PGconn`. Следует использовать функции доступа, описанные ниже, для получения содержимого `PGconn`. Обращение напрямую к внутренним полям `PGconn`, используя сведения из `libpq-int.h`, не рекомендуется, поскольку они могут измениться в будущем.

Следующие функции возвращают значения параметров, которые были установлены в момент подключения. Эти значения не изменяются во время жизни соединения. Если используется строка соединения с несколькими узлами, значения `PQhost`, `PQport` и `PQpass` могут меняться, если с тем же объектом `PGconn` устанавливается новое соединение. Другие значения не меняются на протяжении жизни объекта `PGconn`.

`PQdb`

Возвращает имя базы данных, с которой установлено соединение.

```
char *PQdb(const PGconn *conn);
```

`PQuser`

Возвращает имя пользователя, который установил соединение.

```
char *PQuser(const PGconn *conn);
```

`PQpass`

Возвращает пароль, использованный для подключения.

```
char *PQpass(const PGconn *conn);
```

`PQpass` возвратит либо пароль, указанный в параметрах подключения, либо пароль, полученный из файла паролей (в случае отсутствия первого). Во втором случае, если в параметрах подключения было указано несколько узлов, полагаться на результат `PQpass` нельзя, пока соединение не будет установлено. Состояние подключения позволяет проверить функция `PQstatus`.

`PQhost`

Возвращает имя сервера для активного соединения. Это может быть имя узла, IP-адрес или путь к каталогу, если подключение установлено через сокет Unix. (Признаком подключения к сокету будет абсолютный путь, который начинается с `/`.)

```
char *PQhost(const PGconn *conn);
```

Если в параметрах подключения был задан и `host`, и `hostaddr`, функция `PQhost` выдаст содержимое `host`. Если был задан только `hostaddr`, возвращается это значение. Если в параметрах подключения было задано несколько узлов, `PQhost` возвращает адрес узла, с которым фактически установлено соединение.

`PQhost` возвращает `NULL`, если аргумент *conn* равен `NULL`. Иначе в случае ошибки при получении информации об узле (это возможно, если соединение не установлено до конца или произошла ошибка) она возвращает пустую строку.

Если в параметрах подключения указаны несколько узлов, на результат `PQhost` нельзя полагаться, пока соединение не будет установлено. Проверить состояние соединения позволяет функция `PQstatus`.

`PQhostaddr`

Возвращает IP-адрес сервера для активного соединения. Это может быть адрес, в который разрешилось имя узла, или IP-адрес, переданный в параметре `hostaddr`.

```
char *PQhostaddr(const PGconn *conn);
```

`PQhostaddr` возвращает `NULL`, если аргумент *conn* равен `NULL`. Иначе в случае ошибки при получении информации об узле (это возможно, если соединение не установлено до конца или произошла ошибка) она возвращает пустую строку.

`PQport`

Возвращает номер порта активного соединения.

```
char *PQport(const PGconn *conn);
```

Если в параметрах подключения было задано несколько портов, `PQport` возвращает порт, с которым фактически установлено соединение.

`PQport` возвращает `NULL`, если аргумент *conn* равен `NULL`. Иначе в случае ошибки при получении информации о порте (это возможно, если соединение не установлено до конца или произошла ошибка) она возвращает пустую строку.

Если в параметрах подключения указаны несколько портов, на результат `PQport` нельзя полагаться, пока соединение не будет установлено. Проверить состояние соединения позволяет функция `PQstatus`.

`PQtty`

Эта функция теперь ничего не делает, но необходима для обратной совместимости. Функция всегда возвращает пустую строку или `NULL`, если аргумент *conn* имеет значение `NULL`.

```
char *PQtty(const PGconn *conn);
```

`PQoptions`

Возвращает параметры командной строки, переданные в запросе на подключение.

```
char *PQoptions(const PGconn *conn);
```

Следующие функции возвращают данные статуса, который может измениться в процессе выполнения операций на объекте `PGconn`.

`PQstatus`

Возвращает состояние подключения.

```
ConnStatusType PQstatus(const PGconn *conn);
```

Статус может принимать одно из ряда значений. Однако только два из них видны извне процедуры асинхронного подключения: `CONNECTION_OK` и `CONNECTION_BAD`. Успешное подключение к базе данных имеет статус `CONNECTION_OK`. О неудачной попытке подключения сигнализирует статус `CONNECTION_BAD`. Обычно статус OK остаётся таковым до вызова `PQfinish`, но в случае ошибки соединения статус может смениться на `CONNECTION_BAD` преждевременно. В таком случае приложение может попытаться восстановить подключение, вызвав `PQreset`.

О других кодах состояния, которые могут выдать эти функции, можно узнать в описании `PQconnectStartParams`, `PQconnectStart` и `PQconnectPoll`.

`PQtransactionStatus`

Возвращает текущий статус сервера, отражающий процесс выполнения транзакций.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

Статус может быть одним из `PQTRANS_IDLE` (в настоящее время не занят обработкой транзакции), `PQTRANS_ACTIVE` (команда в процессе обработки), `PQTRANS_INTRANS` (не выполняет работу, но находится в рамках действительной транзакции) или `PQTRANS_INERROR` (не выполняет работу, но находится в рамках транзакции, завершившейся сбоем). Статус принимает значение `PQTRANS_UNKNOWN`, если соединение не работает. Статус принимает значение `PQTRANS_ACTIVE` только тогда, когда запрос был отправлен серверу, но ещё не завершён.

`PQparameterStatus`

Отыскивает текущее значение параметра сервера.

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

Значения определённых параметров сервер сообщает автоматически в начале процедуры подключения или тогда, когда их значения изменяются. `PQparameterStatus` можно использовать, чтобы запросить эти значения. Функция возвращает текущее значение параметра, если оно известно, или `NULL`, если параметр неизвестен.

Параметры, об изменении которых сообщается в текущей версии:

`application_name`
`client_encoding`
`DateStyle`
`default_transaction_read_only`
`in_hot_standby`
`integer_datetimes`
`IntervalStyle`
`is_superuser`
`scram_iterations`
`server_encoding`
`server_version`
`session_authorization`
`standard_conforming_strings`
`TimeZone`

(`default_transaction_read_only` и `in_hot_standby` не отслеживались до 14 версии; `scram_iterations` не отслеживался до 16 версии.) Обратите внимание, что `server_version`, `server_encoding` и `integer_datetimes` не могут меняться после запуска сервера.

Если для `standard_conforming_strings` не передано никакого значения, то приложения могут принять его равным `off`. Это означает, что символы обратной косой черты в строковых литералах интерпретируются в качестве спецсимволов. Также, наличие этого параметра может рассматриваться как указание на то, что синтаксис escape-строк (`E'...'`) является приемлемым.

Хотя возвращаемый указатель объявлен со спецификатором `const`, фактически он указывает на изменяемое хранилище, связанное со структурой `PGconn`. Не стоит рассчитывать на то, что указатель останется действительным при последующих запросах.

`PQprotocolVersion`

Запрашивает протокол, используемый между клиентом и сервером.

```
int PQprotocolVersion(const PGconn *conn);
```

Приложения могут использовать эту функцию, чтобы определить, поддерживаются ли определённые функциональные возможности. В настоящее время возможными значениями являются 3 (протокол версии 3.0) или ноль (проблемы в подключении). Версия протокола не будет изменяться после завершения процедуры подключения, но теоретически она могла бы измениться в процессе переподключения. Версия протокола 3.0 обычно используется при взаимодействии с серверами PostgreSQL версии 7.4 или более поздними.

`PQserverVersion`

Возвращает целочисленное представление версии сервера.

```
int PQserverVersion(const PGconn *conn);
```

Приложения могут использовать эту функцию, чтобы определить версию сервера баз данных, к которому они подключены. Возвращаемое ей число получается в результате умножения номера основной версии на 10000 и добавления номера дополнительной версии. Например, для версии 10.1 будет выдано число 100001, а для версии 11.0 — 110000. Если соединение не работает, то возвращается ноль.

До версии 10 в Postgres Pro номера версий образовывались из трёх чисел, первые два из которых представляли основную версию. Для таких версий `PQserverVersion` отводит на каждое число по две цифры; например, для версии 9.1.5 будет выдано 90105, а для версии 9.2.0 — 90200.

Таким образом, чтобы получить логический номер основной версии для определения доступности функциональности, приложения должны разделить результат `PQserverVersion` на 100, а не на 10000. Во всех сериях номера дополнительных (корректирующих) выпусков различаются только в двух последних цифрах.

`PQerrorMessage`

Возвращает сообщение об ошибке, наиболее недавно сгенерированное операцией, выполненной в рамках текущего подключения.

```
char *PQerrorMessage(const PGconn *conn);
```

Почти все функции библиотеки libpq в случае сбоя сформируют сообщение для `PQerrorMessage` . Обратите внимание, что по соглашениям, принятым в libpq, непустой результат функции `PQerrorMessage` может состоять из нескольких строк и будет включать завершающий символ новой строки. Вызывающая функция не должна освобождать память, на которую указывает возвращаемое значение, напрямую. Она будет освобождена, когда связанный с ней дескриптор `PGconn` будет передан функции `PQfinish`. Не стоит ожидать, что результирующая строка останется той же самой при выполнении нескольких операций со структурой `PGconn`.

`PQsocket`

Получает номер файлового дескриптора для сокета соединения с сервером. Действительный дескриптор будет больше или равен 0; значение -1 показывает, что в данный момент не открыто ни одного соединения с сервером. (Значение не изменится во время обычной работы, но может измениться во время установки или переустановки подключения.)

```
int PQsocket(const PGconn *conn);
```

`PQbackendPID`

Возвращает ID (PID) серверного процесса, обрабатывающего это подключение.

```
int PQbackendPID(const PGconn *conn);
```

PID серверного процесса полезен для отладочных целей и для сопоставления с сообщениями команды `NOTIFY` (которые включают PID уведомляющего серверного процесса). Примите к сведению, что PID принадлежит процессу, выполняющемуся на узле сервера баз данных, а не на локальном узле.

`PQconnectionNeedsPassword`

Возвращает true (1), если метод аутентификации соединения требовал пароля, однако он не был предоставлен. Возвращает false (0), если пароль не требовался.

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

Эту функцию можно применить после неудачной попытки подключения, чтобы решить, нужно ли предлагать пользователю ввести пароль.

`PQconnectionUsedPassword`

Возвращает true (1), если метод аутентификации соединения использовал пароль. Возвращает false (0) в противном случае.

```
int PQconnectionUsedPassword(const PGconn *conn);
```

Эту функцию можно применить как после неудачной, так и после успешной попытки подключения, чтобы определить, требовал ли сервер предоставления пароля.

`PQconnectionUsedGSSAPI`

Возвращает true (1), если метод аутентификации соединения использовал GSSAPI. Возвращает false (0) в противном случае.

```
int PQconnectionUsedGSSAPI(const PGconn *conn);
```

Эту функцию можно применить, чтобы определить, использовалась ли для подключения аутентификация GSSAPI.

Следующие функции возвращают информацию, относящуюся к SSL. Эта информация обычно не меняется после того, как подключение установлено.

`PQsslInUse`

Возвращает true (1), если для текущего подключения используется SSL, и false (0) в противном случае.

```
int PQsslInUse(const PGconn *conn);
```

`PQsslAttribute`

Возвращает связанную с SSL информацию о соединении.

```
const char *PQsslAttribute(const PGconn *conn, const char *attribute_name);
```

Список доступных атрибутов зависит от используемой библиотеки SSL и типа подключения. Если подключение не использует SSL или указанное имя атрибута неизвестно используемой библиотеке, возвращается NULL.

Обычно доступны следующие атрибуты:

`library`

Имя используемой реализации SSL. (В настоящее время поддерживается только "`OpenSSL`")

`protocol`

Применяемая версия SSL/TLS. Наиболее распространены варианты "`TLSv1`", "`TLSv1.1`" и "`TLSv1.2`", но реализация может возвращать и другие строки, если применяется какой-то другой протокол.

`key_bits`

Число бит в ключе, используемом алгоритмом шифрования.

`cipher`

Краткое имя применяемого комплекта шифров, например: "`DHE-RSA-DES-CBC3-SHA`". Эти имена могут быть разными в разных реализациях SSL.

`compression`

Возвращает `on`, если используется сжатие SSL, или `off` в противном случае.

`alpn`

Протокол прикладного уровня, выбранный расширением для согласования протокола прикладного уровня (Application-Layer Protocol Negotiation, ALPN) протокола TLS. libpq поддерживает только протокол `postgresql`, поэтому применяться это значение может только для проверки, поддерживает ли сервер ALPN. Возвращает пустую строку, если ALPN не использовался.

В качестве исключения можно запросить атрибут `library` без подключения, передав NULL в качестве аргумента `conn`. В результате будет выдано имя используемой по умолчанию библиотеки SSL, либо NULL, если библиотека libpq была скомпилирована без поддержки SSL. (В версиях Postgres Pro ниже 15, когда в `conn` передавался NULL, результатом всегда был NULL. Отличить новую реализацию от старой в клиентских программах можно, проверив макрос `LIBPQ_HAS_SSL_LIBRARY_DETECTION`.)

`PQsslAttributeNames`

Возвращает массив имён доступных атрибутов SSL, которые могут использоваться в `PQsslAttribute()`. Завершается массив указателем NULL.

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

Если `conn` равен NULL, возвращаются атрибуты, доступные для библиотеки SSL по умолчанию, или пустой список, если библиотека libpq была скомпилирована без поддержки SSL. При значении `conn`, отличном от NULL, возвращаются атрибуты, доступные для библиотеки SSL, используемой для соединения, или пустой список, если соединение не зашифровано.

`PQsslStruct`

Возвращает указатель на специфичный для реализации SSL объект, описывающий подключение. Если соединение не зашифровано или реализация SSL, используемая для подключения, не поддерживает запрошенный тип объекта, возвращается NULL.

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

Какие структуры можно получить, зависит от используемой реализации SSL. OpenSSL предоставляет одну структуру с именем `OpenSSL` и возвращает указатель на объект `SSL` struct, описанный в OpenSSL. Для написания вызова этой функции можно воспользоваться кодом в следующих строках:

```
#include <libpq-fe.h>
#include <openssl/ssl.h>

...

    SSL *ssl;

    dbconn = PQconnectdb(...);
    ...

    ssl = PQsslStruct(dbconn, "OpenSSL");
    if (ssl)
    {
        /* use OpenSSL functions to access ssl */
    }
```

Эта структура может использоваться, чтобы сличить уровни шифрования, проверить сертификаты сервера и т. д. За информацией об этой структуре обратитесь к документации по OpenSSL.

`PQgetssl`

Возвращает структуру SSL, использовавшуюся в соединении, или NULL, если SSL не используется.

```
void *PQgetssl(const PGconn *conn);
```

Эта функция равнозначна `PQsslStruct(conn, "OpenSSL")`. Её не следует применять в новых приложениях, так как возвращаемая структура специфична для OpenSSL и её нельзя будет получить с другой реализацией SSL. Чтобы проверить, использует ли подключение SSL, лучше вызвать `PQsslInUse`, а чтобы получить свойства подключения — `PQsslAttribute`.

32.3. Функции для исполнения команд

После того как соединение с сервером было успешно установлено, функции, описанные в этом разделе, используются для выполнения SQL-запросов и команд.

32.3.1. Главные функции

`PQexec`

Передаёт команду серверу и ожидает результата.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Возвращает указатель на `PGresult` или, возможно, пустой указатель (null). Как правило, возвращается непустой указатель, исключением являются ситуации нехватки памяти или серьёзные ошибки, такие, как невозможность отправки команды серверу. Для проверки возвращаемого значения на наличие ошибок следует вызвать функцию `PQresultStatus` (в случае нулевого указателя она возвратит `PGRES_FATAL_ERROR`). Для получения дополнительной информации о таких ошибках используйте функцию `PQerrorMessage` .

Строка команды может включать в себя более одной SQL-команды (которые разделяются точкой с запятой). Несколько запросов, отправленных с помощью одного вызова `PQexec`, обрабатываются в рамках одной транзакции, если только команды `BEGIN`/`COMMIT` не включены явно в строку запроса, чтобы разделить его на несколько транзакций. (Подробнее о том, как сервер обрабатывает строки, включающие несколько команд, рассказывается в Подразделе 53.2.2.1.) Однако обратите внимание, что возвращаемая структура `PGresult` описывает только результат последней из выполненных команд, содержащихся в строке запроса. Если одна из команд завершается сбоем, то обработка строки запроса на этом останавливается, и возвращённая структура `PGresult` описывает состояние ошибки.

`PQexecParams`

Отправляет команду серверу и ожидает результата. Имеет возможность передать параметры отдельно от текста SQL-команды.

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

`PQexecParams` подобна `PQexec`, но предлагает дополнительную функциональность: значения параметров могут быть указаны отдельно от самой строки-команды, а результаты запроса могут быть затребованы либо в текстовом, либо в двоичном формате.

Параметры функции следующие:

`conn`

Объект, описывающий подключение, через которое пересылается команда.

`command`

Строка SQL-команды, которая должна быть выполнена. Если используются параметры, то в строке команды на них ссылаются, как `$1`, `$2` и т. д.

`nParams`

Число предоставляемых параметров. Оно равно длине массивов *`paramTypes[]`*, *`paramValues[]`*, *`paramLengths[]`* и *`paramFormats[]`*. (Указатели на массивы могут быть равны `NULL`, когда *`nParams`* равно нулю.)

`paramTypes[]`

Предписывает, посредством OID, типы данных, которые должны быть назначены параметрам. Если значение *`paramTypes`* равно `NULL` или какой-либо отдельный элемент в массиве равен нулю, тогда сервер самостоятельно определит тип данных для параметра точно таким же образом, как он сделал бы для литеральной строки, тип которой не указан.

`paramValues[]`

Указывает фактические значения параметров. Нулевой указатель в этом массиве означает, что соответствующий параметр равен null; в противном случае указатель указывает на текстовую строку, завершающуюся нулевым символом (для текстового формата), или на двоичные данные в формате, которого ожидает сервер (для двоичного формата).

`paramLengths[]`

Указывает фактические длины данных для параметров, представленных в двоичном формате. Он игнорируется для параметров, имеющих значение null, и для параметров, представленных в текстовом формате. Указатель на массив может быть нулевым, когда нет двоичных параметров.

`paramFormats[]`

Указывает, являются ли параметры текстовыми (поместите нуль в элемент массива, соответствующий такому параметру) или двоичными (поместите единицу в элемент массива, соответствующий такому параметру). Если указатель на массив является нулевым, тогда все параметры считаются текстовыми строками.

Значения, переданные в двоичном формате, требуют знания внутреннего представления, которого ожидает сервер. Например, целые числа должны передаваться с использованием сетевого порядка байтов. Передача значений типа `numeric` требует знания формата, в котором их хранит сервер.

`resultFormat`

Требует указать ноль, чтобы получить результаты в текстовом формате, или единицу, чтобы получить результаты в двоичном формате. (В настоящее время нет возможности получить различные столбцы результата в разных форматах, хотя это и возможно на уровне протокола, лежащего в основе подключений.)

Главным преимуществом `PQexecParams` перед `PQexec` является возможность отделить значения параметров от строки запроса. Это позволяет обойтись без кавычек и экранирующих символов, манипулирование которыми бывает трудоёмким и часто приводит к ошибкам.

В отличие от `PQexec`, `PQexecParams` позволяет включать не более одной SQL-команды в строку запроса. (В ней могут содержаться точки с запятой, однако может присутствовать не более одной непустой команды.) Это ограничение накладывается базовым протоколом, но оно приносит и некоторую пользу в качестве дополнительной защиты от атак методом SQL-инъекций.

Подсказка

Указание типов параметров с помощью OID является трудоёмким, особенно если вы предпочитаете не указывать явно значений OID в вашей программе. Однако вы можете избежать этого даже в случаях, когда сервер самостоятельно не может определить тип параметра или выбирает не тот тип, который вы хотите. В строке SQL-команды добавьте явное приведение типа для этого параметра, чтобы показать, какой тип данных вы будете отправлять. Например:

```
SELECT * FROM mytable WHERE x = $1::bigint;
```

Это приведёт к тому, что параметр `$1` будет считаться имеющим тип `bigint`, в то время как по умолчанию ему был бы назначен тот же самый тип, что и `x`. Такое явное принятие решения о типе параметра либо с помощью описанного метода, либо путём задания числового OID строго рекомендуется, когда значения параметров отправляются в двоичном формате, поскольку двоичный формат имеет меньшую избыточность, чем текстовый, и поэтому гораздо менее вероятно, что сервер обнаружит ошибку несоответствия типов, допущенную вами.

`PQprepare`

Отправляет запрос, чтобы создать подготовленный оператор с конкретными параметрами, и ожидает завершения.

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

`PQprepare` создаёт подготовленный оператор для последующего исполнения с помощью `PQexecPrepared`. Благодаря этому, команды, которые будут выполняться многократно, не потребуется разбирать и планировать каждый раз; за подробностями обратитесь к PREPARE.

Функция создаёт подготовленный оператор с именем *`stmtName`* из строки *`query`*, которая должна содержать единственную SQL-команду. *`stmtName`* может быть пустой строкой `""`, тогда будет создан неименованный оператор (в таком случае любой уже существующий неименованный оператор будет автоматически заменён), в противном случае, если имя оператора уже определено в текущем сеансе работы, будет ошибка. Если используются параметры, то в запросе к ним обращаются таким образом: `$1`, `$2` и т. д. *`nParams`* представляет число параметров, типы данных для которых указаны в массиве *`paramTypes[]`*. (Указатель на массив может быть равен `NULL`, когда значение *`nParams`* равно нулю.) *`paramTypes[]`* указывает, посредством OID, типы данных, которые будут назначены параметрам. Если *`paramTypes`* равен `NULL` или какой-либо элемент в этом массиве равен нулю, то сервер назначает тип данных соответствующему параметру точно таким же способом, как он сделал бы для литеральной строки, не имеющей типа. Также в запросе можно использовать параметры с номерами, большими, чем *`nParams`*; типы данных для них сервер также сможет подобрать. (См. описание `PQdescribePrepared`, где сказано, как можно определить, какие типы данных были подобраны).

Как и при вызове `PQexec`, результатом является объект `PGresult`, содержимое которого показывает успех или сбой на стороне сервера. Нулевой указатель означает нехватку памяти или невозможность вообще отправить команду. Для получения дополнительной информации о таких ошибках используйте `PQerrorMessage` .

Подготовленные операторы для использования с `PQexecPrepared` можно также создать путём исполнения SQL-команд PREPARE.

`PQexecPrepared`

Отправляет запрос на исполнение подготовленного оператора с данными параметрами и ожидает результата.

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

`PQexecPrepared` подобна `PQexecParams`, но команда, подлежащая исполнению, указывается путём передачи имени предварительно подготовленного оператора вместо передачи строки запроса. Эта возможность позволяет командам, которые вызываются многократно, подвергаться разбору и планированию только один раз, а не при каждом их исполнении. Оператор должен быть подготовлен предварительно в рамках текущего сеанса работы.

Параметры идентичны `PQexecParams`, за исключением того, что вместо строки запроса передаётся имя подготовленного оператора и отсутствует параметр *`paramTypes[]`* (он не нужен, поскольку типы данных для параметров подготовленного оператора были определены при его создании).

`PQdescribePrepared`

Передаёт запрос на получение информации об указанном подготовленном операторе и ожидает завершения.

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

`PQdescribePrepared` позволяет приложению получить информацию о предварительно подготовленном операторе.

Для ссылки на неименованный оператор значение *`stmtName`* может быть пустой строкой `""` или `NULL`, в противном случае оно должно быть именем существующего подготовленного оператора. В случае успешного выполнения возвращается `PGresult` со статусом `PGRES_COMMAND_OK`. Функции `PQnparams` и `PQparamtype` позволяют извлечь из `PGresult` информацию о параметрах подготовленного оператора, а функции `PQnfields`, `PQfname`, `PQftype` и т. п. предоставляют информацию о результирующих столбцах данного оператора (если они есть).

`PQdescribePortal`

Передаёт запрос на получение информации об указанном портале и ожидает завершения.

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

`PQdescribePortal` позволяет приложению получить информацию о предварительно созданном портале. (libpq не предоставляет прямого доступа к порталам, но вы можете использовать эту функцию для ознакомления со свойствами курсора, созданного с помощью SQL-команды `DECLARE CURSOR`.)

Для ссылки на неименованный портал значение *`portalName`* может быть пустой строкой `""` или `NULL`, в противном случае оно должно быть именем существующего портала. В случае успешного завершения возвращается `PGresult` со статусом `PGRES_COMMAND_OK`. С помощью функций `PQnfields`, `PQfname`, `PQftype` и т. д. можно извлечь из `PGresult` информацию о результирующих столбцах данного портала (если они есть).

`PQclosePrepared`

Передаёт запрос на закрытие указанного подготовленного оператора и ожидает завершения.

```
PGresult *PQclosePrepared(PGconn *conn, const char *stmtName);
```

`PQclosePrepared` позволяет приложению закрыть ранее подготовленный оператор. Закрытие оператора высвобождает все задействованные им ресурсы на сервере, а также позволяет переиспользовать его имя.

Для ссылки на безымянный оператор значение *`stmtName`* может быть пустой строкой `""` или `NULL`. Если имя совпадает с именем существующего оператора, команда ничего не делает. В случае успешного завершения возвращается `PGresult` со статусом `PGRES_COMMAND_OK`.

`PQclosePortal`

Передаёт запрос на закрытие указанного портала и ожидает завершения.

```
PGresult *PQclosePortal(PGconn *conn, const char *portalName);
```

`PQclosePortal` позволяет приложению закрыть ранее созданный портал. Закрытие портала высвобождает все задействованные им ресурсы на сервере, а также позволяет переиспользовать его имя. (libpq не предоставляет прямого доступа к порталам, но вы можете использовать эту функцию, чтобы закрыть курсор, созданный с помощью SQL-команды `DECLARE CURSOR`.)

Для ссылки на безымянный портал значение *`portalName`* может быть пустой строкой `""` или `NULL`. Если имя совпадает с именем существующего портала, команда ничего не делает. В случае успешного завершения возвращается `PGresult` со статусом `PGRES_COMMAND_OK`.

Структура `PGresult`содержит результат, возвращённый сервером. Разработчики приложений libpq должны тщательно поддерживать абстракцию `PGresult`. Для получения доступа к содержимому `PGresult` используйте функции доступа, описанные ниже. Избегайте непосредственного обращения к полям структуры `PGresult`, поскольку они могут измениться в будущем.

`PQresultStatus`

Возвращает статус результата выполнения команды.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

`PQresultStatus` может возвращать одно из следующих значений:

`PGRES_EMPTY_QUERY`

Строка, отправленная серверу, была пустой.

`PGRES_COMMAND_OK`

Успешное завершение команды, не возвращающей никаких данных.

`PGRES_TUPLES_OK`

Успешное завершение команды, возвращающей данные (такой, как `SELECT` или `SHOW`).

`PGRES_COPY_OUT`

Начат перенос данных Copy Out (с сервера).

`PGRES_COPY_IN`

Начат перенос данных Copy In (на сервер).

`PGRES_BAD_RESPONSE`

Ответ сервера не был распознан.

`PGRES_NONFATAL_ERROR`

Произошла некритическая ошибка (уведомление или предупреждение).

`PGRES_FATAL_ERROR`

Произошла критическая ошибка.

`PGRES_COPY_BOTH`

Начат перенос данных Copy In/Out (на сервер и с сервера). Эта функция в настоящее время используется только для потоковой репликации, поэтому такой статус не должен иметь место в обычных приложениях.

`PGRES_SINGLE_TUPLE`

Структура `PGresult` содержит только одну результирующую строку, возвращённую текущей командой. Этот статус имеет место только тогда, когда для данного запроса был выбран режим построчного вывода (см. Раздел 32.6).

`PGRES_TUPLES_CHUNK`

Структура `PGresult` содержит несколько результирующих кортежей, возвращённых текущей командой. Этот статус имеет место только тогда, когда для данного запроса был выбран режим построчного вывода (см. Раздел 32.6). Количество кортежей не превысит ограничения `PQsetChunkedRowsMode`.

`PGRES_PIPELINE_SYNC`

`PGresult` представляет точку синхронизации в конвейерном режиме, запрошенную `PQpipelineSync` или `PQsendPipelineSync`. Этот статус возможен, только если выбран конвейерный режим.

`PGRES_PIPELINE_ABORTED`

Структура `PGresult` представляет конвейер, получивший ошибку от сервера. Функция `PQgetResult` должна вызываться неоднократно, и каждый раз она будет возвращать этот код состояния до конца текущего конвейера, после чего она вернёт `PGRES_PIPELINE_SYNC` и сможет возобновиться нормальная обработка.

Если статус результата `PGRES_TUPLES_OK`, `PGRES_SINGLE_TUPLE` или `PGRES_TUPLES_CHUNK`, тогда для извлечения строк, возвращённых запросом, можно использовать функции, описанные ниже. Обратите внимание, что команда `SELECT`, даже когда она не извлекает ни одной строки, всё же показывает `PGRES_TUPLES_OK`. `PGRES_COMMAND_OK` предназначен для команд, которые никогда не возвращают строки (`INSERT` или `UPDATE` без использования предложения `RETURNING` и др.). Ответ `PGRES_EMPTY_QUERY` может указывать на наличие ошибки в клиентском программном обеспечении.

Результат со статусом `PGRES_NONFATAL_ERROR` никогда не будет возвращён напрямую функцией `PQexec` или другими функциями исполнения запросов; вместо этого результаты такого вида передаются обработчику уведомлений (см. Раздел 32.13).

`PQresStatus`

Преобразует значение перечислимого типа, возвращённое функцией `PQresultStatus`, в строковую константу, описывающую код статуса. Вызывающая функция не должна освобождать память, на которую указывает возвращаемый указатель.

```
char *PQresStatus(ExecStatusType status);
```

`PQresultErrorMessage`

Возвращает сообщение об ошибке, связанное с командой, или пустую строку, если ошибки не произошло.

```
char *PQresultErrorMessage(const PGresult *res);
```

Если произошла ошибка, то возвращённая строка будет включать завершающий символ новой строки. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель `PGresult` будет передан функции `PQclear`.

Если непосредственно после вызова `PQexec` или `PQgetResult` вызвать функцию `PQerrorMessage` (для данного подключения), то она возвратит ту же самую строку, что и `PQresultErrorMessage` (для данного результата). Однако `PGresult` сохранит своё сообщение об ошибке до тех пор, пока не будет уничтожен, в то время как сообщение об ошибке, связанное с данным подключением, будет изменяться при выполнении последующих операций. Воспользуйтесь функцией `PQresultErrorMessage`, когда вы хотите узнать статус, связанный с конкретной структурой `PGresult`; используйте функцию `PQerrorMessage`, когда вы хотите узнать статус выполнения самой последней операции на данном соединении.

`PQresultVerboseErrorMessage`

Возвращает переформатированную версию сообщения об ошибке, связанного с объектом `PGresult`.

```
char *PQresultVerboseErrorMessage(const PGresult *res,
                                  PGVerbosity verbosity,
                                  PGContextVisibility show_context);
```

В некоторых ситуациях клиент может захотеть получить более подробную версию ранее выданного сообщения об ошибке. Эту потребность удовлетворяет функция `PQresultVerboseErrorMessage`, формируя сообщение, которое было бы выдано функцией `PQresultErrorMessage`, если бы заданный уровень детализации был текущим для соединения в момент заполнения `PGresult`. Если же в `PGresult` не содержится ошибка, вместо этого выдаётся сообщение «PGresult is not an error result» (PGresult — не результат с ошибкой). Возвращаемое этой функцией сообщение завершается переводом строки.

В отличие от многих других функций, извлекающих данные из `PGresult`, результат этой функции — новая размещённая в памяти строка. Когда эта строка будет не нужна, вызывающий код должен освободить её место, вызвав `PQfreemem()`.

При нехватке памяти может быть возвращёно NULL.

`PQresultErrorField`

Возвращает индивидуальное поле из отчёта об ошибке.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

`fieldcode` это идентификатор поля ошибки; см. символические константы, перечисленные ниже. Если `PGresult` не содержит ошибки или предупреждения или не включает указанное поле, то возвращается `NULL`. Значения полей обычно не включают завершающий символ новой строки. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель `PGresult` будет передан функции `PQclear`.

Доступны следующие коды полей:

`PG_DIAG_SEVERITY`

Серьёзность; поле может содержать `ERROR`, `FATAL` или `PANIC` (в сообщении об ошибке) либо `WARNING`, `NOTICE`, `DEBUG`, `INFO` или `LOG` (в сообщении-уведомлении) либо локализованный перевод одного из этих значений. Присутствует всегда.

`PG_DIAG_SEVERITY_NONLOCALIZED`

Серьёзность; поле может содержать `ERROR`, `FATAL` или `PANIC` (в сообщении об ошибке) либо `WARNING`, `NOTICE`, `DEBUG`, `INFO` или `LOG` (в сообщении-уведомлении). Это поле подобно `PG_DIAG_SEVERITY`, но его содержимое никогда не переводится. Присутствует только в отчётах, выдаваемых Postgres Pro версии 9.6 и новее.

`PG_DIAG_SQLSTATE`

Код ошибки в соответствии с соглашением о кодах SQLSTATE. Код SQLSTATE идентифицирует тип случившейся ошибки; он может использоваться клиентскими приложениями, чтобы выполнять конкретные операции (такие, как обработка ошибок) в ответ на конкретную ошибку базы данных. Список возможных кодов SQLSTATE приведён в Приложении A. Это поле не подлежит локализации. Оно всегда присутствует.

`PG_DIAG_MESSAGE_PRIMARY`

Главное сообщение об ошибке, предназначенное для прочтения пользователем. Как правило составляет всего одну строку. Это поле всегда присутствует.

`PG_DIAG_MESSAGE_DETAIL`

Необязательное дополнительное сообщение об ошибке, передающее более детальную информацию о проблеме. Может занимать несколько строк.

`PG_DIAG_MESSAGE_HINT`

Подсказка: необязательное предположение о том, что можно сделать в данной проблемной ситуации. Оно должно отличаться от детальной информации в том смысле, что оно предлагает совет (возможно, и неподходящий), а не просто факты. Может занимать несколько строк.

`PG_DIAG_STATEMENT_POSITION`

Строка, содержащая десятичное целое число, указывающее позицию расположения ошибки в качестве индекса в оригинальной строке оператора. Первый символ имеет позицию 1, при этом позиции измеряются в символах а не в байтах.

`PG_DIAG_INTERNAL_POSITION`

Это поле определяется точно так же, как и поле `PG_DIAG_STATEMENT_POSITION`, но оно используется, когда позиция местонахождения ошибки относится к команде, сгенерированной внутренними модулями, а не к команде, представленной клиентом. Когда появляется это поле, то всегда появляется и поле `PG_DIAG_INTERNAL_QUERY`.

`PG_DIAG_INTERNAL_QUERY`

Текст команды, сгенерированной внутренними модулями, завершившейся сбоем. Это мог бы быть, например, SQL-запрос, выданный функцией на языке PL/pgSQL.

`PG_DIAG_CONTEXT`

Характеристика контекста, в котором произошла ошибка. В настоящее время она включает вывод стека вызовов активных функций процедурного языка и запросов, сгенерированных внутренними модулями. Стек выводится по одному элементу в строке, при этом первым идет самый последний из элементов (самый недавний вызов).

`PG_DIAG_SCHEMA_NAME`

Если ошибка была связана с конкретным объектом базы данных, то в это поле будет записано имя схемы, содержащей данный объект.

`PG_DIAG_TABLE_NAME`

Если ошибка была связана с конкретной таблицей, то в это поле будет записано имя таблицы. (Для получения имени схемы для данной таблицы обратитесь к полю, содержащему имя схемы.)

`PG_DIAG_COLUMN_NAME`

Если ошибка была связана с конкретным столбцом таблицы, то в это поле будет записано имя столбца. (Чтобы идентифицировать таблицу, обратитесь к полям, содержащим имена схемы и таблицы.)

`PG_DIAG_DATATYPE_NAME`

Если ошибка была связана с конкретным типом данных, то в это поле будет записано имя типа данных. (Чтобы получить имя схемы, которой принадлежит этот тип данных, обратитесь к полю, содержащему имя схемы.)

`PG_DIAG_CONSTRAINT_NAME`

Если ошибка была связана с конкретным ограничением, то в это поле будет записано имя ограничения. Чтобы получить имя таблицы или домена, связанных с этим ограничением, обратитесь к полям, перечисленным выше. (С этой целью индексы рассматриваются как ограничения, даже если они и не были созданы с помощью синтаксиса для создания ограничений.)

`PG_DIAG_SOURCE_FILE`

Имя файла, содержащего позицию в исходном коде, для которой было выдано сообщение об ошибка.

`PG_DIAG_SOURCE_LINE`

Номер строки той позиции в исходном коде, для которой было выдано сообщение об ошибке.

`PG_DIAG_SOURCE_FUNCTION`

Имя функции в исходном коде, сообщающей об ошибке.

Примечание

Поля для имени схемы, имени таблицы, имени столбца, имени типа данных и имени ограничения предоставляются лишь для ограниченного числа типов ошибок; см. Приложение A. Не рассчитывайте на то, что присутствие любого из этих полей гарантирует и присутствие какого-то другого поля. Базовые источники ошибок придерживаются взаимосвязей, описанных выше, но функции, определённые пользователем, могут использовать эти поля другими способами. Аналогично, не рассчитывайте на то, что эти поля обозначают объекты, существующие в текущей базе данных в настоящий момент.

Клиент отвечает за форматирование отображаемой информации в соответствии с его нуждами; в частности, он должен разбивать длинные строки, как требуется. Символы новой строки, встречающиеся в полях сообщения об ошибке, должны обрабатываться, как разрывы абзацев, а не строк.

Ошибки, сгенерированные внутренними модулями libpq, будут иметь поля серьёзности ошибки и основного сообщения, но, как правило, никаких других полей.

Заметьте, что поля ошибки доступны только из объектов `PGresult`, а не из объектов `PGconn`. Не существует функции `PQerrorField`.

`PQclear`

Освобождает область памяти, связанную с `PGresult`. Результат выполнения каждой команды должен быть освобождён с помощью `PQclear`, когда он больше не нужен.

```
void PQclear(PGresult *res);
```

Если аргумент является указателем `NULL`, операция не выполняется.

Вы можете сохранять объект `PGresult`, пока он вам нужен; он не исчезает ни когда вы выдаёте новую команду, ни даже если вы закрываете соединение. Чтобы от него избавиться, вы должны вызвать `PQclear`. Если этого не делать, в вашем приложении будут иметь место утечки памяти.

32.3.2. Извлечение информации, связанной с результатом запроса

Эти функции служат для извлечения информации из объекта `PGresult`, который представляет результат успешного запроса (то есть такого, который имеет статус `PGRES_TUPLES_OK`, `PGRES_SINGLE_TUPLE` или `PGRES_TUPLES_CHUNK`). Их также можно использовать для извлечения информации об успешной операции DESCRIBE: результат этой операции содержит всю ту же самую информацию о столбцах, которая была бы получена при реальном исполнении запроса, но не содержит ни одной строки. Для объектов, имеющих другие значения статуса, эти функции будут действовать таким образом, как будто результат не содержит ни одной строки и ни одного столбца.

`PQntuples`

Возвращает число строк (кортежей) в полученной выборке. (Заметьте, что объекты `PGresult` не могут содержать более чем `INT_MAX` строк, так что для результата достаточно типа `int`.)

```
int PQntuples(const PGresult *res);
```

`PQnfields`

Возвращает число столбцов (полей) в каждой строке полученной выборки.

```
int PQnfields(const PGresult *res);
```

`PQfname`

Возвращает имя столбца, соответствующего данному номеру столбца. Номера столбцов начинаются с 0. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на `PGresult` будет передан функции `PQclear`.

```
char *PQfname(const PGresult *res,
              int column_number);
```

Если номер столбца выходит за пределы допустимого диапазона, то возвращается `NULL`.

`PQfnumber`

Возвращает номер столбца, соответствующий данному имени столбца.

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

Если данное имя не совпадает с именем ни одного из столбцов, то возвращается -1.

Данное имя интерпретируется, как идентификатор в SQL-команде. Это означает, что оно переводится в нижний регистр, если только оно не заключено в двойные кавычки. Например, для выборки, сгенерированной с помощью такой SQL-команды:

```
SELECT 1 AS FOO, 2 AS "BAR";
```

мы получили бы следующие результаты:

```
PQfname(res, 0)              foo
PQfname(res, 1)              BAR
PQfnumber(res, "FOO")        0
PQfnumber(res, "foo")        0
PQfnumber(res, "BAR")        -1
PQfnumber(res, "\"BAR\"")    1
```

`PQftable`

Возвращает OID таблицы, из которой был получен данный столбец. Номера столбцов начинаются с 0.

```
Oid PQftable(const PGresult *res,
             int column_number);
```

В следующих случаях возвращается `InvalidOid`: если номер столбца выходит за пределы допустимого диапазона; если указанный столбец не является простой ссылкой на столбец таблицы. Вы можете сделать запрос к системной таблице `pg_class`, чтобы точно определить, к какой таблице было произведено обращение.

Тип данных `Oid` и константа `InvalidOid` будут определены, когда вы включите заголовочный файл для libpq. Они будут принадлежать к одному из целочисленных типов.

`PQftablecol`

Возвращает номер столбца (в пределах его таблицы) для указанного столбца в полученной выборке. Номера столбцов в полученной выборке начинаются с 0, но столбцы в таблице имеют ненулевые номера.

```
int PQftablecol(const PGresult *res,
                int column_number);
```

В следующих случаях возвращается ноль: если номер столбца выходит за пределы допустимого диапазона; если указанный столбец не является простой ссылкой на столбец таблицы.

`PQfformat`

Возвращает код формата, показывающий формат данного столбца. Номера столбцов начинаются с 0.

```
int PQfformat(const PGresult *res,
              int column_number);
```

Значение кода формата, равное нулю, указывает на текстовое представление данных, в то время, как значение, равное единице, означает двоичное представление. (Другие значения кодов зарезервированы для определения в будущем.)

`PQftype`

Возвращает тип данных, соответствующий данному номеру столбца. Возвращаемое целое значение является внутренним номером OID для этого типа. Номера столбцов начинаются с 0.

```
Oid PQftype(const PGresult *res,
            int column_number);
```

Вы можете сделать запрос к системной таблице `pg_type`, чтобы получить имена и свойства различных типов данных.

`PQfmod`

Возвращает модификатор типа для столбца, соответствующего данному номеру. Номера столбцов начинаются с 0.

```
int PQfmod(const PGresult *res,
           int column_number);
```

Интерпретация значений модификатора зависит от типа; они обычно показывают точность или предельные размеры. Значение -1 используется, чтобы показать «нет доступной информации». Большинство типов данных не используют модификаторов, в таком случае значение всегда будет -1.

`PQfsize`

Возвращает размер в байтах для столбца, соответствующего данному номеру. Номера столбцов начинаются с 0.

```
int PQfsize(const PGresult *res,
            int column_number);
```

`PQfsize` возвращает размер пространства, выделенного для этого столбца в строке базы данных, другими словами, это размер внутреннего представления этого типа данных на сервере. (Следовательно, эта информация не является по-настоящему полезной для клиентов.) Отрицательное значение говорит о том, что тип данных имеет переменную длину.

`PQbinaryTuples`

Возвращает 1, если `PGresult` содержит двоичные данные, или 0, если данные текстовые.

```
int PQbinaryTuples(const PGresult *res);
```

Эта функция не рекомендуется к использованию (за исключением применения в связи с командой `COPY`), поскольку один и тот же `PGresult` может содержать в некоторых столбцах текстовые данные, а в остальных — двоичные. Предпочтительнее использовать `PQfformat`. `PQbinaryTuples` возвращает 1, только если все столбцы в выборке являются двоичными (код формата 1).

`PQgetvalue`

Возвращает значение одного поля из одной строки, содержащейся в `PGresult`. Номера строк и столбцов начинаются с 0. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на `PGresult` будет передан функции `PQclear`.

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

Для данных в текстовом формате значение, возвращаемое функцией `PQgetvalue`, является значением поля, представленным в виде символьной строки с завершающим нулевым символом. Для данных в двоичном формате используется двоичное представление значения. Оно определяется функциями `typsend` и `typreceive` для конкретного типа данных. (В этом случае к значению также добавляется нулевой байт, но обычно это не приносит пользы, поскольку вероятно, что значение уже содержит нулевые байты.)

Пустая строка возвращается в том случае, когда поле содержит null. Чтобы отличить значения null от пустых строковых значений, воспользуйтесь функцией `PQgetisnull`.

Указатель, возвращаемый функцией `PQgetvalue`, указывает на область памяти внутри структуры `PGresult`. Модифицировать данные, на которые указывает этот указатель, не следует. Вместо этого нужно явно скопировать данные в другую область памяти, если предполагается использовать их за рамками жизненного цикла структуры `PGresult`.

`PQgetisnull`

Проверяет поле на предмет отсутствия значения (null). Номера строк и столбцов начинаются с 0.

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

Эта функция возвращает 1, если значение в поле отсутствует (null), и 0, если поле содержит отличное от null значение. (Заметьте, что если поле содержит null, функция `PQgetvalue` возвращает пустую строку, а не нулевой указатель.)

`PQgetlength`

Возвращает фактическую длину значения поля в байтах. Номера строк и столбцов начинаются с 0.

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

Это фактическая длина данных для конкретного значения данных, то есть размер объекта, на который указывает `PQgetvalue`. Для текстового формата данных это то же самое, что `strlen()`. Для двоичного же формата это существенная информация. Учтите, что при расчёте фактического объёма данных *не* следует полагаться на `PQfsize`.

`PQnparams`

Возвращает число параметров подготовленного оператора.

```
int PQnparams(const PGresult *res);
```

Эта функция полезна только при исследовании результата работы функции `PQdescribePrepared`. Для других типов результатов она возвратит ноль.

`PQparamtype`

Возвращает тип данных для указанного параметра оператора. Номера параметров начинаются с 0.

```
Oid PQparamtype(const PGresult *res, int param_number);
```

Эта функция полезна только при исследовании результата работы функции `PQdescribePrepared`. Для других типов результатов она возвратит ноль.

`PQprint`

Выводит все строки и, по выбору, имена столбцов в указанный поток вывода.

```
void PQprint(FILE *fout,      /* поток вывода */
             const PGresult *res,
             const PQprintOpt *po);
typedef struct
{
    pqbool  header;      /* печатать заголовки полей и счётчик строк */
    pqbool  align;       /* выравнивать поля */
    pqbool  standard;    /* старый формат */
    pqbool  html3;       /* выводить HTML-таблицы */
    pqbool  expanded;    /* расширять таблицы */
    pqbool  pager;       /* использовать программу для постраничного просмотра, если
 нужно */
    char    *fieldSep;   /* разделитель полей */
    char    *tableOpt;   /* атрибуты для HTML-таблицы */
    char    *caption;    /* заголовок HTML-таблицы */
    char    **fieldName; /* массив заменителей для имён полей, завершающийся нулевым
 символом */
} PQprintOpt;
```

Эту функцию прежде использовала утилита psql для вывода результатов запроса, но больше она её не использует. Обратите внимание, предполагается, что все данные представлены в текстовом формате.

32.3.3. Получение другой информации о результате

Эти функции используются для получения остальной информации из объектов `PGresult`.

`PQcmdStatus`

Возвращает дескриптор статуса для SQL-команды, которая сгенерировала `PGresult`.

```
char *PQcmdStatus(PGresult *res);
```

Как правило, это просто имя команды, но могут быть включены и дополнительные сведения, такие, как число обработанных строк. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на `PGresult` будет передан функции `PQclear`.

`PQcmdTuples`

Возвращает число строк, которые затронула SQL-команда.

```
char *PQcmdTuples(PGresult *res);
```

Эта функция возвращает строковое значение, содержащее число строк, которые затронул SQL-оператор, сгенерировавший данный `PGresult`. Эту функцию можно использовать только сразу после выполнения команд `SELECT`, `CREATE TABLE AS`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `MOVE`, `FETCH` или `COPY`, а также после оператора `EXECUTE`, выполнившего подготовленный запрос, содержащий команды `INSERT`, `UPDATE`, `DELETE` или `MERGE`. Если команда, которая сгенерировала `PGresult`, была какой-то иной, то `PQcmdTuples` возвращает пустую строку. Вызывающая функция не должна напрямую освобождать память, на которую указывает возвращаемый указатель. Она будет освобождена, когда соответствующий указатель на `PGresult` будет передан функции `PQclear`.

`PQoidValue`

Возвращает OID вставленной строки, если SQL-команда была командой `INSERT`, которая вставила ровно одну строку в таблицу, имеющую идентификаторы OID, или командой `EXECUTE`, которая выполнила подготовленный запрос, содержащий соответствующий оператор `INSERT`. В противном случае эта функция возвращает `InvalidOid`. Эта функция также возвратит `InvalidOid`, если таблица, затронутая командой `INSERT`, не содержит идентификаторов OID.

```
Oid PQoidValue(const PGresult *res);
```

`PQoidStatus`

Эта функция считается не рекомендуемой к использованию (в качестве замены служит `PQoidValue`), а также она не является потокобезопасной. Она возвращает строковое значение, содержащее OID вставленной строки, в то время как `PQoidValue` возвращает значение OID.

```
char *PQoidStatus(const PGresult *res);
```

32.3.4. Экранирование строковых значений для включения в SQL-команды

`PQescapeLiteral`

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

`PQescapeLiteral` экранирует строковое значение для использования внутри SQL-команды. Это полезно при вставке в SQL-команды значений данных в виде строковых констант. Определённые символы (например, кавычки и обратная косая черта) должны экранироваться, чтобы предотвратить их специальную интерпретацию синтаксическим анализатором языка SQL. Эту операцию выполняет `PQescapeLiteral`.

`PQescapeLiteral` возвращает экранированную версию параметра *`str`*, размещённую в области памяти, распределённой с помощью функции `malloc()`. Эту память нужно освобождать с помощью функции `PQfreemem()`, когда возвращённое значение больше не требуется. Завершающий нулевой байт не нужен и не должен учитываться в параметре *`length`*. (Если завершающий нулевой байт был найден до того, как были обработаны *`length`* байт, то `PQescapeLiteral` останавливает работу на нулевом байте; таким образом, поведение функции напоминает `strncpy`.) В возвращённой строке все специальные символы заменены таким образом, что синтаксический анализатор строковых литералов Postgres Pro может обработать их должным образом. В строку также добавляется завершающий нулевой байт. Одинарные кавычки, которые должны окружать строковые литералы Postgres Pro, включаются в результирующую строку.

В случае ошибки `PQescapeLiteral` возвращает `NULL`, и в объект *`conn`* помещается соответствующее сообщение.

Подсказка

Особенно важно выполнять надлежащее экранирование при обработке строк, полученных из ненадёжных источников. В противном случае ваша безопасность подвергается риску из-за уязвимости в отношении атак с использованием «SQL-инъекций», с помощью которых нежелательные SQL-команды направляются в вашу базу данных.

Обратите внимание, что нет необходимости (и это будет даже некорректно) экранировать значения данных, передаваемых в виде отдельных параметров в функцию `PQexecParams` или родственные ей функции.

`PQescapeIdentifier`

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

`PQescapeIdentifier` экранирует строку, предназначенную для использования в качестве идентификатора SQL, например имени таблицы, столбца или функции. Это полезно, когда идентификатор, выбранный пользователем, может содержать специальные символы, которые в противном случае не интерпретировались бы синтаксическим анализатором SQL, как часть идентификатора, или когда идентификатор может содержать символы верхнего регистра и этот регистр требуется сохранить.

`PQescapeIdentifier` возвращает версию параметра *`str`*, экранированную как SQL-идентификатор, и размещённую в области памяти, распределённой с помощью функции `malloc()`. Эту память нужно освобождать с помощью функции `PQfreemem()`, когда возвращённое значение больше не требуется. Завершающий нулевой байт не нужен и не должен учитываться в параметре *`length`*. (Если завершающий нулевой байт был найден до того, как были обработаны *`length`* байт, то `PQescapeIdentifier` останавливается на нулевом байте; таким образом, поведение функции напоминает `strncpy`.) В возвращённой строке все специальные символы заменены таким образом, что она будет надлежащим образом обработана, как SQL-идентификатор. Завершающий нулевой байт также будет добавлен. Возвращённая строка также будет заключена в двойные кавычки.

В случае ошибки `PQescapeIdentifier` возвращает `NULL`, и в объект *`conn`* помещается соответствующее сообщение.

Подсказка

Как и в случае со строковыми литералами, для того чтобы предотвратить атаки с помощью SQL-инъекций, SQL-идентификаторы должны экранироваться, когда они получены из ненадёжного источника.

`PQescapeStringConn`

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

`PQescapeStringConn` экранирует строковые литералы наподобие `PQescapeLiteral`. Но, в отличие от `PQescapeLiteral`, за предоставление буфера надлежащего размера отвечает вызывающая функция. Более того, `PQescapeStringConn` не добавляет одинарные кавычки, которые должны окружать строковые литералы Postgres Pro; они должны быть включены в SQL-команду, в которую вставляется результирующая строка. Параметр *`from`* указывает на первый символ строки, которая должна экранироваться, а параметр *`length`* задаёт число байт в этой строке. Завершающий нулевой байт не требуется и не должен учитываться в параметре *`length`*. (Если завершающий нулевой байт был найден до того, как были обработаны *`length`* байт, то `PQescapeStringConn` останавливается на нулевом байте; таким образом, поведение функции напоминает `strncpy`.) Параметр *`to`* должен указывать на буфер, который сможет вместить как минимум на один байт больше, чем предписывает удвоенное значение параметра *`length`*, в противном случае поведение функции не определено. Поведение будет также не определено, если строки *`to`* и *`from`* перекрываются.

Если параметр *error* не равен `NULL`, тогда значение `*error` устанавливается равным нулю в случае успешной работы и не равным нулю в случае ошибки. В настоящее время единственным возможным условием возникновения ошибки является неверная мультибайтовая кодировка в исходной строке. Выходная строка формируется даже при наличии ошибки, но можно ожидать, что сервер отвергнет её как неверно сформированную. В случае ошибки в объект *conn* записывается соответствующее сообщение независимо от того, равно ли `NULL` значение параметра *error*.

`PQescapeStringConn` возвращает число байт, записанных по адресу *to*, не включая завершающий нулевой байт.

`PQescapeString`

`PQescapeString` является более старой, не рекомендованной к использованию версией функции `PQescapeStringConn`.

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

Единственное отличие от `PQescapeStringConn` состоит в том, что у функции `PQescapeString` нет параметров `PGconn` и *error*. Поэтому она не может скорректировать своё поведение в зависимости от свойств подключения (таких, как кодировка символов) и, следовательно, *она может выдавать неверные результаты*. Также она не имеет возможности сообщить об ошибках.

`PQescapeString` может безопасно использоваться в клиентских программах, которые работают лишь с одним подключением к Postgres Pro за один раз (в этом случае функция может найти то, что ей нужно знать, «за кулисами»). В других контекстах её использование несёт угрозу безопасности и его следует избегать в пользу применения функции `PQescapeStringConn`.

`PQescapeByteaConn`

Экранирует двоичные данные для их использования внутри SQL-команды с типом данных `bytea`. Как и в случае с `PQescapeStringConn`, эта функция применяется только тогда, когда данные вставляются непосредственно в строку SQL-команды.

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

Байты, имеющие определённые значения, должны экранироваться, когда они используются внутри литерала, имеющего тип `bytea`, в SQL-операторе. `PQescapeByteaConn` экранирует байты, используя либо hex-кодирование, либо экранирование с помощью обратной косой черты. См. Раздел 8.4 для получения дополнительной информации.

Параметр *from* указывает на первый байт строки, которая должна экранироваться, а параметр *from_length* задаёт число байт в этой двоичной строке. (Завершающий нулевой байт не нужен и не учитывается.) Параметр *to_length* указывает на переменную, которая будет содержать длину результирующей экранированной строки. Эта длина включает завершающий нулевой байт результирующей строки.

`PQescapeByteaConn` возвращает экранированную версию двоичной строки, на которую указывает параметр *from*, и размещает её в памяти, распределённой с помощью `malloc()`. Эта память должна быть освобождена с помощью функции `PQfreemem()`, когда результирующая строка больше не нужна. В возвращаемой строке все специальные символы заменены так, чтобы синтаксический анализатор литеральных строк Postgres Pro и функция ввода для типа `bytea` могли обработать их надлежащим образом. Завершающий нулевой байт также добавляется. Одинарные кавычки, которые должны окружать строковые литералы Postgres Pro, не являются частью результирующей строки.

В случае ошибки возвращается нулевой указатель, и соответствующее сообщение об ошибке записывается в объект *conn*. В настоящее время единственной возможной ошибкой может быть нехватка памяти для результирующей строки.

`PQescapeBytea`

`PQescapeBytea` является устаревшей и не рекомендуемой к использованию версией функции `PQescapeByteaConn`.

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

Единственное отличие от `PQescapeByteaConn` состоит в том, что у функции `PQescapeBytea` нет параметра `PGconn`. Поэтому `PQescapeBytea` может безопасно использоваться в клиентских программах, которые работают лишь с одним подключением к Postgres Pro в один момент времени (в этом случае функция может узнать то, что ей нужно, «за кулисами»). Она *может выдавать неверные результаты* при использовании в программах, которые устанавливает несколько подключений к базам данных (в таких случаях используйте `PQescapeByteaConn`).

`PQunescapeBytea`

Преобразует строковое представление двоичных данных в двоичные данные — является обратной функцией к функции `PQescapeBytea`. Она нужна, когда данные типа `bytea` извлекаются в текстовом формате, но не когда они извлекаются в двоичном формате.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

Параметр *`from`* указывает на строку, такую, какую могла бы возвратить функция `PQgetvalue`, применённая к столбцу типа `bytea`. `PQunescapeBytea` преобразует это строковое представление в его двоичное представление. Она возвращает указатель на буфер, выделенный функцией `malloc()`, или `NULL` в случае ошибки и помещает размер буфера по адресу *`to_length`*. Когда результат не будет нужен, необходимо освободить его память, вызвав `PQfreemem`.

Это преобразование не является точной инверсией для `PQescapeBytea`, поскольку ожидается, что строка, полученная от `PQgetvalue`, не будет «экранированной». В частности, это означает, что учитывать режим спецпоследовательностей не нужно, и поэтому в параметре `PGconn` нет необходимости.

32.4. Асинхронная обработка команд

Функция `PQexec` хорошо подходит для отправки команд серверу в нормальных, синхронных приложениях. Однако она имеет ряд недостатков, которые могут иметь значение для некоторых пользователей:

- `PQexec` ожидает завершения выполнения команды. Однако приложение может быть занято чем-то другим (например, обрабатывать активность в пользовательском интерфейсе), в таком случае блокировка приложения в ожидании ответа будет нежелательной.
- Поскольку выполнение клиентского приложения приостанавливается, пока оно ожидает результата, то приложению трудно решить, что оно хотело бы попытаться отменить выполняющуюся команду. (Это можно сделать из обработчика сигнала, но никак иначе.)
- `PQexec` может возвратить только одну структуру `PGresult`. Если отправленная серверу командная строка содержит несколько SQL-команд, функция `PQexec` отбрасывает все результаты `PGresult`, кроме последнего.
- `PQexec` всегда собирает все результаты выполнения команды, буферизуя их в единственной структуре `PGresult`. В то время как для приложения это упрощает логику обработки ошибок, это может быть непрактично, когда результат содержит много строк.

Приложения, которым эти ограничения не подходят, могут вместо `PQexec` использовать функции, на которых она базируется, `PQsendQuery` и `PQgetResult`. Есть также функции `PQsendQueryParams`, `PQsendPrepare`, `PQsendQueryPrepared`, `PQsendDescribePrepared`, `PQsendDescribePortal`, `PQsendClosePrepared` и `PQsendClosePortal`, которые в сочетании с

`PQgetResult` действуют аналогично функциям `PQexecParams`, `PQprepare`, `PQexecPrepared`, `PQdescribePrepared`, `PQdescribePortal`, `PQclosePrepared` и `PQclosePortal`, соответственно.

`PQsendQuery`

Отправляет команду серверу, не ожидая получения результата. Если команда была отправлена успешно, то функция возвратит значение 1, в противном случае она возвратит 0 (тогда нужно воспользоваться функцией `PQerrorMessage` для получения дополнительной информации о сбое).

```
int PQsendQuery(PGconn *conn, const char *command);
```

После успешного вызова `PQsendQuery` вызовите `PQgetResult` один или несколько раз, чтобы получить результаты. Функцию `PQsendQuery` нельзя вызвать повторно (на том же самом соединении) до тех пор, пока `PQgetResult` не вернёт нулевой указатель, означающий, что выполнение команды завершено.

В конвейерном режиме эту функцию вызывать нельзя.

`PQsendQueryParams`

Отправляет серверу команду и обособленные параметры, не ожидая получения результатов.

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

Эта функция эквивалентна функции `PQsendQuery`, за исключением того, что параметры запроса можно указать отдельно от самой строки запроса. Эта функция обрабатывает свои параметры точно так же, как и функция `PQexecParams`. Аналогично функции `PQexecParams` она позволяет включить только одну команду в строку запроса.

`PQsendPrepare`

Посылает запрос на создание подготовленного оператора с данными параметрами и не дожидается завершения его выполнения.

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

Это асинхронная версия функции `PQprepare`. Она возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию `PQgetResult`, чтобы определить, создал ли сервер подготовленный оператор. Эта функция обрабатывает свои параметры точно так же, как и функция `PQprepare`.

`PQsendQueryPrepared`

Посылает запрос на выполнение подготовленного оператора с данными параметрами, не ожидая получения результата.

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

Эта функция подобна функции `PQsendQueryParams`, но выполняемую команду определяет не текст запроса, а ссылка на предварительно подготовленный оператор. Эта функция обрабатывает свои параметры точно так же, как и функция `PQexecPrepared`.

`PQsendDescribePrepared`

Отправляет запрос на получение информации об указанном подготовленном операторе и не дожидается завершения выполнения запроса.

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

Это асинхронная версия функции `PQdescribePrepared`. Она возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию `PQgetResult` для получения результата. Эта функция обрабатывает свои параметры точно так же, как и функция `PQdescribePrepared`.

`PQsendDescribePortal`

Отправляет запрос на получение информации об указанном портале и не дожидается завершения выполнения запроса.

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

Это асинхронная версия функции `PQdescribePortal`. Она возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию `PQgetResult` для получения результата. Эта функция обрабатывает свои параметры точно так же, как и функция `PQdescribePortal`.

`PQsendClosePrepared`

Отправляет запрос на закрытие указанного подготовленного оператора и не дожидается завершения выполнения запроса.

```
nt PQsendClosePrepared(PGconn *conn, const char *stmtName);
```

Это асинхронная версия функции `PQclosePrepared`. Она возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию `PQgetResult` для получения результата. Эта функция обрабатывает свои параметры точно так же, как и функция `PQclosePrepared`.

`PQsendClosePortal`

Отправляет запрос на закрытие указанного портала и не дожидается завершения выполнения запроса.

```
int PQsendClosePortal(PGconn *conn, const char *portalName);
```

Это асинхронная версия функции `PQclosePortal`. Она возвращает 1, если ей удалось отправить запрос, и 0 в противном случае. После её успешного вызова следует вызвать функцию `PQgetResult` для получения результата. Эта функция обрабатывает свои параметры точно так же, как и функция `PQclosePortal`.

`PQgetResult`

Ожидает получения следующего результата после предшествующего вызова `PQsendQuery`, `PQsendQueryParams`, `PQsendPrepare`, `PQsendQueryPrepared`, `PQsendDescribePrepared`, `PQsendDescribePortal`, `PQsendClosePrepared`, `PQsendClosePortal`, `PQsendPipelineSync` или `PQpipelineSync` и возвращает его. Когда команда завершена и результатов больше не будет, возвращается нулевой указатель.

```
PGresult *PQgetResult(PGconn *conn);
```

Функция `PQgetResult` должна вызываться повторно до тех пор, пока она не вернёт нулевой указатель, означающий, что команда завершена. (Если она вызвана, когда нет ни одной активной команды, тогда `PQgetResult` просто возвратит нулевой указатель сразу же.) Каждый ненулевой результат, полученный от `PQgetResult`, должен обрабатываться с помощью тех же самых функций доступа к структуре `PGresult`, которые были описаны выше. Не забывайте освобождать память, занимаемую каждым результирующим объектом, с помощью функции `PQclear` когда работа с этим объектом закончена. Обратите внимание, что `PQgetResult` заблокируется, только если какая-либо команда активна, а необходимые ответные данные ещё не были прочитаны функцией `PQconsumeInput` .

В конвейерном режиме `PQgetResult` будет работать как обычно, если не произойдёт ошибка; для любого запроса, отправленного после запроса, вызвавшего ошибку, до следующей точки синхронизации (но не включая её), будет возвращаться специальный результат типа `PGRES_PIPELINE_ABORTED`, а после него — нулевой указатель. При достижении точки синхронизации конвейера будет возвращён результат типа `PGRES_PIPELINE_SYNC`. Результат следующего запроса после точки синхронизации поступит немедленно (то есть после точки синхронизации нулевой указатель не возвращается).

Примечание

Даже когда `PQresultStatus` показывает критическую ошибку, всё равно следует вызывать функцию `PQgetResult` до тех пор, пока она не возвратит нулевой указатель, чтобы позволить libpq полностью обработать информацию об ошибке.

Использование `PQsendQuery` и `PQgetResult` решает одну из проблем `PQexec`: если строка запроса содержит несколько SQL-команд, то результаты каждой из них можно получить индивидуально. (Между прочим, это позволяет организовать частичное совмещение обработки: клиент может обрабатывать результаты одной команды, в то время как сервер ещё работает с более поздними запросами, содержащимися в той же строке.)

Ещё одной часто требующейся функциональной возможностью, которую можно получить с помощью `PQsendQuery` и `PQgetResult`, является извлечение больших выборок по одной строке за раз. Это обсуждается в Разделе 32.6.

Сам по себе вызов `PQgetResult` всё же приведёт к блокировке клиента до тех пор, пока сервер не завершит выполнение следующей SQL-команды. Этого можно избежать с помощью надлежащего использования ещё двух функций:

`PQconsumeInput`

Если сервер готов передать данные, принять их.

```
int PQconsumeInput(PGconn *conn);
```

`PQconsumeInput` обычно возвращает 1, показывая, что «ошибки нет», но возвращает 0, если имела место какая-либо проблема (в таком случае можно обратиться к функции `PQerrorMessage` за уточнением). Обратите внимание, что результат не говорит, были ли какие-либо входные данные фактически собраны. После вызова функции `PQconsumeInput` приложение может проверить `PQisBusy` и/или `PQnotifies`, чтобы посмотреть, не изменилось ли их состояние.

`PQconsumeInput` можно вызвать, даже если приложение ещё не готово иметь дело с результатом или уведомлением. Функция прочитает доступные данные и сохранит их в буфере, при этом обрабатывая условие готовности к чтению функции `select()`. Таким образом, приложение может использовать `PQconsumeInput` >, чтобы немедленно обработать это состояние `select()`, а изучать результаты позже.

`PQisBusy`

Возвращает 1, если команда занята работой, то есть функция `PQgetResult` в случае вызова будет заблокирована в ожидании ввода. Возвращаемое значение 0 показывает, что функция `PQgetResult` при её вызове гарантированно не будет заблокирована.

```
int PQisBusy(PGconn *conn);
```

Функция `PQisBusy` сама не будет пытаться прочитать данные с сервера, поэтому, чтобы выйти из занятого состояния, необходимо вызвать `PQconsumeInput` , .

В типичном приложении, использующем эти функции, будет главный цикл, в котором с применением функций `select()` или `poll()` организуется ожидание всех условий, на которые нужно реагировать в этом цикле. Одним из условий будет поступление на вход данных от сервера, что в терминах функции `select()` означает наличие данных, которые можно прочитать через файловый дескриптор, выдаваемый функцией `PQsocket`. Когда в главном цикле обнаруживаются готовые входные данные, следует вызвать `PQconsumeInput` , чтобы прочитать их. После этого можно вызвать `PQisBusy`, и если в результате получен 0 (что свидетельствует о свободном состоянии), затем вызвать `PQgetResult`. В главном цикле также можно вызвать `PQnotifies`, чтобы проверить сообщения `NOTIFY` (см. Раздел 32.9).

Клиент, который использует `PQsendQuery`/`PQgetResult`, может также попытаться отменить команду, которая всё ещё обрабатывается сервером; см. Раздел 32.7. Но независимо от возвращаемого значения функции `PQcancelBlocking`, приложение должно продолжать обычную последовательность операций чтения результатов запроса, вызывая `PQgetResult`. В случае успешной отмены команда просто завершится раньше, чем она завершилась бы, выполняясь до конца.

Используя функции, описанные выше, можно избежать блокирования, ожидая ввода от сервера баз данных. Однако всё же возможно, что приложение будет заблокировано в ожидании отправки вывода на сервер. Это бывает относительно нечасто, но может иметь место, если отправлены очень длинные SQL-команды или значения данных. (Однако это значительно более вероятно, если приложение отправляет данные через команду `COPY IN`.) Чтобы предотвратить эту возможность и достичь совершенно неблокирующего режима работы с базой данных, можно использовать следующие дополнительные функции.

`PQsetnonblocking`

Устанавливает неблокирующий статус подключения.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Устанавливает состояние подключения как неблокирующее, если *arg* равен 1, или блокирующее, если *arg* равен 0. Возвращает 0 в случае успешного завершения и -1 в случае ошибки.

В неблокирующем состоянии успешные вызовы `PQsendQuery`, `PQputline`, `PQputnbytes`, `PQputCopyData` и `PQendcopy` не блокируются; изменения сохраняются в локальном буфере вывода до тех пор, пока не будут сброшены. Неудачные вызовы возвращают ошибку, и их придётся повторить.

Обратите внимание, что функция `PQexec` не соблюдает неблокирующий режим. Если она вызывается, она всё равно работает в блокирующем режиме.

`PQisnonblocking`

Возвращает режим блокирования для подключения базы данных.

```
int PQisnonblocking(const PGconn *conn);
```

Возвращает 1, если подключение установлено в неблокирующем режиме, и 0, если режим блокирующий.

`PQflush`

Пытается сбросить любые выходные данные, стоящие в очереди, на сервер. Возвращает 0 в случае успеха (или если очередь на отправку пуста), -1 в случае сбоя по какой-либо причине или 1, если она ещё не смогла отправить все данные, находящиеся в очереди (этот случай может иметь место, только если соединение неблокирующее).

```
int PQflush(PGconn *conn);
```

После отправки любой команды или данных через неблокирующее подключение следует вызвать функцию `PQflush`. Если она возвратит 1, подождите, пока сокет станет готовым к чтению или записи. Если он станет готовым к записи, снова вызовите `PQflush`. Если он станет готовым к чтению, вызовите `PQconsumeInput` , а затем вновь вызовите `PQflush`. Повторяйте до тех пор, пока `PQflush` не возвратит 0. (Необходимо выполнять проверку на состояние готовности к чтению и забирать входные данные с помощью `PQconsumeInput` , потому что сервер может заблокироваться, пытаясь отправить нам данные, например, сообщения NOTICE, и не будет читать наши данные до тех пор, пока мы не прочитаем его.) Как только `PQflush` возвратит 0, подождите, пока сокет не станет готовым к чтению, а затем прочитайте ответ, как описано выше.

32.5. Конвейерный режим

Конвейерный режим libpq даёт приложению отправить запрос, не получая результат предыдущего. Используя преимущества конвейерного режима, клиент будет меньше ждать сервер, поскольку несколько запросов/результатов могут быть отправлены/получены в одной сетевой транзакции.

Хотя конвейерный режим обеспечивает значительный прирост производительности, разрабатывать клиентские приложения, использующие этот режим, гораздо сложнее, так как требуется реализовывать очереди ожидающих запросов и сопоставлять результаты с запросами в очереди.

Для конвейерного режима также обычно требуется больше памяти как на стороне клиента, так и на стороне сервера, хотя тщательное и агрессивное управление очередью отправки/получения может это нивелировать. Это не зависит от того, в каком режиме — блокирующем или нет — устанавливается подключение.

Хотя конвейерный API в libpq появился с выходом Postgres Pro 14, это клиентская функциональность, которая не требует специальной поддержки на стороне сервера и работает с любым сервером, поддерживающем 3-ю версию расширенного протокола запросов. За дополнительными сведениями обратитесь к Подразделу 53.2.4.

32.5.1. Использование конвейерного режима

Для запуска конвейеров приложение должно переключить соединение в конвейерный режим посредством функции `PQenterPipelineMode`. Можно проверить, включён ли данный режим, используя функцию `PQpipelineStatus`. В конвейерном режиме разрешены только асинхронные операции, использующие расширенный протокол запросов, а строки команд, содержащие несколько SQL-команд, и команда `COPY` запрещены. Использовать функции синхронного выполнения команд, такие как `PQfn`, `PQexec`, `PQexecParams`, `PQprepare`, `PQexecPrepared`, `PQdescribePrepared`, `PQdescribePortal`, `PQclosePrepared`, `PQclosePortal` в этом режиме нельзя. Функцию `PQsendQuery` применять также нельзя, так как она использует простой протокол запросов. Обработав результаты всех отправленных команд и итоговый результат конвейера, приложение может вернуться в обычный режим, вызвав `PQexitPipelineMode`.

Примечание

Конвейерный режим рекомендуется использовать при работе libpq в неблокирующем режиме. В блокирующем режиме возможны взаимоблокировки на уровне клиент-сервер. [1]

32.5.1.1. Отправка запросов

Перейдя в конвейерный режим, приложение отправляет запросы, вызывая `PQsendQueryParams` или родственную ей функцию `PQsendQueryPrepared`, работающую с подготовленными запросами. Данные запросы ставятся в очередь на стороне клиента, а затем сбрасываются на сервер; это происходит, когда вызывается `PQpipelineSync`, устанавливающая точку синхронизации в конвейере,

---

[1] Клиент может заблокироваться, пытаясь передать запросы серверу, а сервер заблокируется, пытаясь выдать клиенту результаты уже выполненных запросов. Это возможно, только когда клиент передаёт так много запросов, что заполняет и свой выходной буфер, и входной буфер сервера, и только затем переключается на обработку передаваемых сервером результатов, но предсказать, когда точно это произойдёт, сложно.

или когда вызывается `PQflush`. В конвейерном режиме также работают функции `PQsendPrepare`, `PQsendDescribePrepared`, `PQsendDescribePortal`, `PQsendClosePrepared` и `PQsendClosePortal`. Обработка результатов описана ниже.

Сервер выполняет операторы и в порядке их поступления от клиента возвращает результаты. Сервер начнёт выполнять команды в конвейере немедленно, не дожидаясь конца конвейера. Обратите внимание, что результаты буферизуются на стороне сервера; сервер сбрасывает этот буфер, когда функцией `PQpipelineSync` или `PQsendPipelineSync` устанавливается точка синхронизации или когда вызывается функция `PQsendFlushRequest`. Если при выполнении какого-либо оператора возникает ошибка, сервер прерывает текущую транзакцию и не выполняет никакую следующую команду в очереди до следующей точки синхронизации; для каждой такой команды выдаётся результат `PGRES_PIPELINE_ABORTED`. (Это справедливо и тогда, когда в конвейере передаются команды, которые могли бы откатить транзакцию.) Обработка запроса возобновляется после точки синхронизации.

Одна операция вполне может зависеть от результатов предыдущей; например, в одном запросе может создаваться таблица, которую будет использовать следующий запрос в том же конвейере. Точно так же приложение может создать именованный подготовленный оператор и выполнить его с последующими операторами в том же конвейере.

32.5.1.2. Обработка результатов

Чтобы обработать результат одного запроса в конвейере, приложение многократно вызывает `PQgetResult` и обрабатывает каждый её результат, пока `PQgetResult` не выдаст NULL. Затем может быть получен результат следующего запроса в конвейере, также с помощью `PQgetResult`, и весь цикл повторяется. Результаты отдельных запросов приложение обрабатывает обычным образом. После того, как будут выданы результаты всех запросов в конвейере, `PQgetResult` выдаёт результат со значением статуса `PGRES_PIPELINE_SYNC`.

Клиент может отложить обработку результатов до тех пор, пока весь конвейер не будет отправлен, или чередовать её с отправкой дальнейших запросов в конвейере; см. Подраздел 32.5.1.4.

Функция `PQgetResult` работает так же, как и при обычной асинхронной обработке, но может дополнительно выдавать результаты новых типов `PGRES_PIPELINE_SYNC` и `PGRES_PIPELINE_ABORTED`. `PGRES_PIPELINE_SYNC` выдаётся ровно один раз для каждого вызова `PQpipelineSync` или `PQsendPipelineSync` в соответствующей точке конвейера. `PGRES_PIPELINE_ABORTED` выдаётся вместо обычного результата запроса для первой ошибки и всех последующих результатов до следующего `PGRES_PIPELINE_SYNC`; см. Подраздел 32.5.1.3.

Функции `PQisBusy`, `PQconsumeInput` и т. п. работают как обычно при обработке результатов конвейера. В частности, вызов `PQisBusy` в середине конвейера возвращает 0, если были обработаны результаты всех выполненных на данный момент запросов.

libpq не предоставляет приложению никакой информации о запросе, обрабатываемом в данный момент (за исключением того, что `PQgetResult` возвращает NULL, чтобы указать, что далее выдаются результаты следующего запроса). Приложение должно отслеживать порядок отправки запросов, чтобы связать их с соответствующими результатами. Для этого приложение обычно использует конечный автомат или очередь.

32.5.1.3. Обработка ошибок

С точки зрения клиента, после того, как `PQresultStatus` возвращает `PGRES_FATAL_ERROR`, конвейер помечается как нерабочий. `PQresultStatus` будет выдавать результат `PGRES_PIPELINE_ABORTED` для каждой оставшейся в очереди операции в нерабочем конвейере. Функция `PQpipelineSync` или `PQsendPipelineSync` выдаёт результат `PGRES_PIPELINE_SYNC`, сигнализируя о возвращении конвейера в рабочее состояние и возобновлении нормальной обработки результатов.

Клиент *должен* обрабатывать результаты, вызывая `PQgetResult` во время восстановления после ошибки.

Если в конвейере передавалась неявная транзакция, то операции, которые уже были выполнены, откатываются, а операции, которые были поставлены в очередь после неудачной операции, полностью пропускаются. То же самое происходит, если в конвейере запускается и фиксируется одна явная транзакция (т. е. первый оператор — `BEGIN`, а последний — `COMMIT`), за исключением того, что сеанс остаётся в состоянии прерванной транзакции в конце конвейера. Если конвейер содержит *несколько явных транзакций*, все транзакции, зафиксированные до ошибки, остаются зафиксированными, текущая транзакция прерывается, а все последующие операции полностью пропускаются, включая транзакции. Если выполняется точка синхронизации, когда явный блок транзакции находится в прерванном состоянии, следующий конвейер сразу же становится нерабочим, если следующая команда (`ROLLBACK`) не переключает его в обычный режим.

Примечание

Клиент не должен рассчитывать на то, что выполненная работа зафиксирована сразу после того, как был *отправлен* `COMMIT`; только получение соответствующего результата даёт такую гарантию. Поскольку ошибки поступают асинхронно, приложение должно уметь возвращаться к моменту последнего *подтверждённого* зафиксированного изменения и повторно отправлять работу, выполненную после этого момента, если что-то пойдёт не так.

32.5.1.4. Чередование обработки результатов и отправки запросов

Во избежание взаимоблокировок с большими конвейерами, клиент должен быть построен вокруг неблокирующего цикла событий, реализованного с использованием таких механизмов операционной системы, как `select`, `poll`, `WaitForMultipleObjectEx` и т. д.

Клиентское приложение, как правило, должно поддерживать очередь ещё не отправленной работы и очередь работы, которая уже отправлена, но её результаты ещё не обработаны. Когда сокет доступен для записи, в него следует отправлять очередной объём работы. Когда он доступен для чтения, из него следует прочитать и обработать результаты, сопоставив их со следующей записью в очереди ожидания результатов. Если объём памяти позволяет, результаты следует читать достаточно часто — дожидаться окончания конвейера не требуется. Конвейеры должны быть ограничены логическими единицами работы, обычно (но не обязательно) по одной транзакции на конвейер. Нет необходимости выходить из конвейерного режима и возвращаться в него между конвейерами, так же как не нужно дожидаться завершения одного конвейере, прежде чем передавать работу в другой.

32.5.2. Функции, связанные с конвейерным режимом

`PQpipelineStatus`

Возвращает текущее состояние конвейерного режима для подключения libpq.

```
PGpipelineStatus PQpipelineStatus(const PGconn *conn);
```

`PQpipelineStatus` может выдавать одно из следующих значений:

`PQ_PIPELINE_ON`

Подключение libpq находится в конвейерном режиме.

`PQ_PIPELINE_OFF`

Подключение libpq *не* находится в конвейерном режиме.

`PQ_PIPELINE_ABORTED`

Соединение libpq находится в конвейерном режиме, и при обработке текущего конвейера произошла ошибка. Флаг прерывания сбрасывается, когда `PQgetResult` возвращает результат типа `PGRES_PIPELINE_SYNC`.

`PQenterPipelineMode`

Переводит подключение в конвейерный режим, если оно в данный момент находится в режиме ожидания или уже находится в конвейерном режиме.

```
int PQenterPipelineMode(PGconn *conn);
```

Возвращает 1 в случае успеха. Возвращает 0 и ничего не делает, если соединение в настоящий момент не простаивает, т. е. если у него есть готовый результат, или оно ожидает поступления дополнительных данных от сервера и т. д. Эта функция на самом деле ничего не отправляет серверу, а просто изменяет состояние соединения libpq.

`PQexitPipelineMode`

Выводит подключение из конвейерного режима, если подключение находится в нём и его очереди пусты.

```
int PQexitPipelineMode (PGconn * conn);
```

Возвращает 1 в случае успеха. Не в конвейерном режиме возвращает 1 и не выполняет никаких действий. Если обработка текущего оператора не завершена или `PQgetResult` не была вызвана для сбора результатов всех ранее отправленных запросов, возвращает 0 (в этом случае используйте `PQerrorMessage` , чтобы получить дополнительную информацию о проблеме).

`PQpipelineSync`

Отмечает точку синхронизации в конвейере, отправляя сообщение синхронизации и очищая буфер отправки. Эта точка служит ограничителем неявной транзакции и точкой восстановления после ошибки; см. Подраздел 32.5.1.3.

```
int PQpipelineSync(PGconn *conn);
```

Возвращает 1 в случае успеха. Возвращает 0, если соединение не находится в конвейерном режиме или сообщение синхронизации отправить не удалось.

`PQsendPipelineSync`

Отмечает точку синхронизации в конвейере, отправляя сообщение синхронизации, но не очищая буфер отправки. Эта точка служит ограничителем неявной транзакции и точкой восстановления после ошибки; см. Подраздел 32.5.1.3.

```
int PQsendPipelineSync(PGconn *conn);
```

Возвращает 1 в случае успеха. Возвращает 0, если соединение не находится в конвейерном режиме или сообщение синхронизации отправить не удалось. Обратите внимание, что сообщение не сохраняется на сервере автоматически. При необходимости используйте функцию `PQflush`.

`PQsendFlushRequest`

Отправляет серверу команду сбросить его буфер вывода.

```
int PQsendFlushRequest(PGconn *conn);
```

Возвращает 1 в случае успеха. Возвращает 0 в случае любой ошибки.

Сервер сбрасывает свой буфер вывода автоматически, когда вызывается `PQpipelineSync` или передаётся любой запрос не в конвейерном режиме; эта функция полезна в конвейерном режиме: она позволяет сбросить серверный буфер, не устанавливая точку синхронизации. Обратите внимание, что сам этот запрос не передаётся серверу автоматически; чтобы передать его немедленно, вызовите `PQflush`.

32.5.3. Когда использовать конвейерный режим

Как и в случае с асинхронным режимом запросов, при использовании конвейерного режима нет значительных издержек производительности. Использование конвейерного режима увеличивает сложность клиентского приложения и требует дополнительной осторожности во избежание взаимоблокировок клиент-сервер, но может предложить значительное улучшение производительности в обмен на увеличение объёма используемой памяти из-за более длительного выхода из состояния.

Конвейерный режим наиболее полезен, когда сервер находится на большом расстоянии от клиента, т. е. когда сетевая задержка («ping time») велика, а также когда много небольших операций выполняются в быстрой последовательности. Как правило, использование конвейерных команд даёт меньше преимуществ, когда выполнение каждого запроса занимает в несколько раз больше времени, чем передача данных клиент-сервер и обратно. Операция из 100 операторов, выполняемая на сервере за 300 миллисекунд, без конвейеризации займёт 30 секунд из-за одной только сетевой задержки; с конвейеризацией данная операция потратит не более 0,3 секунды на ожидание результатов от сервера.

Используйте конвейерные команды, когда ваше приложение выполняет множество небольших операций `INSERT`, `UPDATE` и `DELETE`, которые нелегко преобразовать в наборы операций или в операцию `COPY`.

Конвейерный режим бесполезен, когда информация из одной операции требуется клиенту для выполнения следующей операции. В таких случаях клиенту придётся ввести точку синхронизации и дождаться полного цикла передачи данных клиент-сервер, чтобы получить требуемые результаты. Однако часто можно настроить клиент для обмена необходимой информацией на стороне сервера. Циклы чтения-изменения-записи особенно хорошо подходят для такой настройки; например:

```
BEGIN;
SELECT x FROM mytable WHERE id = 42 FOR UPDATE;
-- result: x=2
-- client adds 1 to x:
UPDATE mytable SET x = 3 WHERE id = 42;
COMMIT;
```

можно гораздо эффективнее сделать с помощью:

```
UPDATE mytable SET x = x + 1 WHERE id = 42;
```

Конвейеризация менее полезна и более сложна, когда один конвейер содержит несколько транзакций (см. Подраздел 32.5.1.3).

32.6. Извлечение порциями информации, связанной с результатом запроса

Обычно libpq собирает весь результат выполнения SQL-команды и возвращает его приложению в виде единственной структуры `PGresult`. Это может оказаться неприемлемым для команд, которые возвращают большое число строк. В таких случаях приложение может воспользоваться функциями `PQsendQuery` и `PQgetResult` в *однострочном режиме* или *порционном режиме*. В этих режимах результирующие строки передаются приложению по мере того, как они принимаются от сервера, — по одной за раз в однострочном и по несколько в порционном режиме.

Для того чтобы войти в один из этих режимов, вызовите `PQsetSingleRowMode` или `PQsetChunkedRowsMode` сразу же после успешного вызова функции `PQsendQuery` (или родственной функции). Выбор этого режима действителен только для запроса, исполняемого в данный момент. Затем повторно вызывайте функцию `PQgetResult` до тех пор, пока она не возвратит null, как описано в Раздел 32.4. Если запрос возвращает какое-то число строк, то они возвращаются в виде одного или нескольких объектов `PGresult`, которые выглядят, как обычные выборки, за исключением того, что их код статуса будет `PGRES_SINGLE_TUPLE` в однострочном режиме или `PGRES_TUPLES_CHUNK` в порционном режиме вместо `PGRES_TUPLES_OK`. В каждом объекте `PGRES_SINGLE_TUPLE` будет выведена только одна результирующая строка, в то время как объект `PGRES_TUPLES_CHUNK` содержит хотя бы одну строку, но не больше указанного числа строк на одну порцию. После последней строки (или сразу же, если запрос не возвращает ни одной строки) будет возвращён объект, не содержащий ни одной строки и имеющий статус `PGRES_TUPLES_OK`; это сигнал о том, что строк больше не будет. (Но обратите внимание, что всё же необходимо продолжать вызывать функцию `PQgetResult`, пока она не возвратит null.) Все эти объекты `PGresult` будут содержать те же самые описательные данные (имена столбцов, типы и т. д.), которые имел бы обычный объект `PGresult`. Память, занимаемую каждым объектом, нужно освобождать с помощью `PQclear`, как обычно.

При использовании конвейерного режима необходимо активировать однострочный или порционный режим для каждого запроса в конвейере, прежде чем получать результаты для этого запроса с помощью `PQgetResult`. См. Раздел 32.5 для получения дополнительной информации.

`PQsetSingleRowMode`

Выбирает однострочный режим для текущего выполняющегося запроса.

```
int PQsetSingleRowMode(PGconn *conn);
```

Эту функцию можно вызывать только непосредственно после функции `PQsendQuery` или одной из её родственных функций, до выполнения любой другой операции через это подключение, такой, как `PQconsumeInput` или `PQgetResult`. Если вызвать её в подходящий момент, функция активирует однострочный режим для текущего запроса и возвращает 1. В противном случае режим остаётся прежним, а функция возвращает 0. Режим в любом случае сбрасывается по завершении текущего запроса.

`PQsetChunkedRowsMode`

Выбирает порционный режим для текущего выполняющегося запроса.

```
int PQsetChunkedRowsMode(PGconn *conn, int chunkSize);
```

Эта функция похожа на `PQsetSingleRowMode`, но вместо того, чтобы получать на каждый `PGresult` только одну строку за раз, выводит порции строк, максимальное количество которых указано в *`chunkSize`*. Данную функцию можно вызывать только сразу после функции `PQsendQuery` или одной из её родственных функций, до выполнения любой другой операции через это подключение, такой, как `PQconsumeInput` или `PQgetResult`. Если вызвать её в подходящий момент, функция активирует порционный режим для текущего запроса и возвращает 1. В противном случае режим остаётся прежним, а функция возвращает 0. Режим в любом случае сбрасывается по завершении текущего запроса.

Внимание

В процессе обработки запроса сервер может возвратить некоторое количество строк, а затем столкнуться с ошибкой, вынуждающей его аварийно завершить запрос. Обычно libpq отбрасывает такие строки и сообщает только об ошибке. Но в однострочном или порционном режиме эти строки уже будут возвращены приложению. Следовательно, приложение увидит ряд объектов `PGresult`, имеющих статус `PGRES_SINGLE_TUPLE` или `PGRES_TUPLES_CHUNK`, за которыми последует объект со статусом `PGRES_FATAL_ERROR`. Для обеспечения надлежащего поведения транзакций приложение должно быть спроектировано таким образом, чтобы отбрасывать или отменять все операции, проведённые с уже обработанными строками, если запрос в конечном итоге завершается сбоем.

32.7. Отмена запросов в процессе выполнения

32.7.1. Функции для отправки запросов отмены

`PQcancelCreate`

Готовит соединение для отправки запросов отмены.

```
PGcancelConn *PQcancelCreate(PGconn *conn);
```

Функция `PQcancelCreate` создаёт объект `PGcancelConn`, однако отправка запросов отмены не начинается сразу. Запрос отмены можно отправить через это соединение блокирующим способом, используя `PQcancelBlocking`, и неблокирующим, используя `PQcancelStart`. Полученное значение можно передать через `PQcancelStatus` для проверки, что объект `PGcancelConn` создан успешно. Объект `PGcancelConn` является непрозрачной структурой, которая не предполагает доступ напрямую через приложение. Этот объект `PGcancelConn` можно использовать для потокобезопасной отмены запроса, который выполняется на исходном соединении.

При настройке соединения для отправки запроса отмены переиспользуются многие параметры соединения исходного клиента. Важно помнить, что если исходное соединение требует шифрования соединения и/или проверки целевого узла (через `sslmode` или `gssencmode`), то соединение для запроса отмены создаётся с такими же требованиями. Любые параметры соединения, используемые только во время или сразу же после аутентификации клиента, игнорируются, поскольку запросы отмены не требуют аутентификации, а соединение закрывается сразу после отправки запроса отмены.

Заметьте, что если функция `PQconnectStart` возвращает ненулевой указатель, то, закончив его использование, вы должны вызвать `PQcancelFinish`, чтобы освободить полученную структуру и все связанные с ней блоки памяти. Это нужно сделать, даже если попытка подключения не последует или окажется неуспешной.

`PQcancelBlocking`

Требует, чтобы сервер прекратил обработку текущей команды блокирующим способом.

```
int PQcancelBlocking(PGcancelConn *cancelConn);
```

Запрос отправляется по заданному соединению `PGcancelConn`, которое необходимо создать функцией `PQcancelCreate`. `PQcancelBlocking` возвращает 1, если запрос отмены успешно отправлен, и 0 в противном случае. В случае неудачи сообщение об ошибке можно получить с помощью функции `PQcancelErrorMessage` .

Успешная отправка отмены ещё не является гарантией того, что запрос будет иметь какой-то эффект. Если отмена сработала, текущая команда завершится досрочно и возвратит в качестве результата ошибку. Если же отмена не получится (например, потому, что сервер уже завершил обработку команды), тогда видимого результата не будет вовсе.

`PQcancelStart`
`PQcancelPoll`

Выдаёт запрос на то, чтобы сервер прерывал выполнение текущей команды неблокирующим способом.

```
int PQcancelStart(PGcancelConn *cancelConn);

PostgresPollingStatusType PQcancelPoll(PGcancelConn *cancelConn);
```

Запрос отправляется по заданному соединению `PGcancelConn`, которое необходимо создать функцией `PQcancelCreate`. `PQcancelStart` возвращает 1, если запрос отмены может быть отправлен, и 0 в противном случае. В случае неудачи сообщение об ошибке можно получить с помощью функции `PQcancelErrorMessage` .

Если вызов `PQcancelStart` оказался успешным, нужно опросить libpq для продолжения процедуры подключения. Вызовите `PQcancelSocket` для получения дескриптора нижележащего сокета, через который устанавливается соединение с базой данных. (Внимание: этот сокет может меняться от вызова к вызову `PQcancelPoll`.) Организуйте цикл таким образом: если `PQcancelPoll(cancelConn)` при последнем вызове возвращает `PGRES_POLLING_READING`, ожидайте, пока сокет не окажется готовым для чтения (это покажет функция `select()`, `poll()` или подобная системная функция). Затем снова вызовите `PQcancelPoll(cancelConn)`. Если же `PQcancelPoll(cancelConn)` при последнем вызове возвратила `PGRES_POLLING_WRITING`, дождитесь готовности сокета к записи, а затем снова вызовите `PQcancelPoll(cancelConn)`. На первой итерации, то есть когда вы ещё не вызывали `PQcancelPoll(cancelConn)`, реализуйте то же поведение, что и после получения `PGRES_POLLING_WRITING`. Продолжайте этот цикл, пока `PQcancelPoll(cancelConn)` не выдаст значение `PGRES_POLLING_FAILED`, сигнализирующее об ошибке при установлении соединения, или `PGRES_POLLING_OK`, показывающее, что соединение установлено успешно.

Успешная отправка отмены ещё не является гарантией того, что запрос будет иметь какой-то эффект. Если отмена сработала, текущая команда завершится досрочно и возвратит в качестве результата ошибку. Если же отмена не получится (например, потому, что сервер уже завершил обработку команды), тогда видимого результата не будет вовсе.

В любое время в процессе подключения его состояние можно проверить, вызвав `PQcancelStatus`. Если этот вызов возвратит `CONNECTION_BAD`, значит, процедура отмены завершилась сбоем; если вызов возвратит `CONNECTION_OK`, значит, запрос отмены успешно отправлен. Оба эти состояния можно определить на основе возвращаемого значения функции `PQcancelPoll`, описанной выше. Другие состояния могут также иметь место в течение (и только в течение) асинхронной процедуры подключения. Они показывают текущую стадию процедуры подключения и могут быть полезны, например, для предоставления обратной связи пользователю. Вот эти состояния:

`CONNECTION_ALLOCATED`

Ожидание, когда вызов функции `PQcancelStart` или `PQcancelBlocking` фактически откроет сокет. Это этап соединения сразу после вызова `PQcancelCreate` или `PQcancelReset`, на котором соединение с сервером ещё не инициализировано. Чтобы фактически начать отправку запроса отмены, используйте `PQcancelStart` или `PQcancelBlocking`.

`CONNECTION_STARTED`

Ожидание, пока соединение будет установлено.

`CONNECTION_MADE`

Соединение установлено; ожидание отправки.

`CONNECTION_AWAITING_RESPONSE`

Ожидание ответа от сервера.

`CONNECTION_SSL_STARTUP`

Согласование SSL-шифрования.

`CONNECTION_GSS_STARTUP`

Согласование GSS-шифрования.

Заметьте, что, хотя эти константы и сохранятся (для поддержания совместимости), приложение никогда не должно полагаться на то, что они появятся в каком-то конкретном порядке или вообще появятся, а также на то, что состояние всегда примет одно из этих документированных значений. Приложение может сделать что-то наподобие:

```
switch(PQcancelStatus(conn))
{
        case CONNECTION_STARTED:
            feedback = "Подключение...";
            break;

        case CONNECTION_MADE:
            feedback = "Подключён к серверу...";
            break;
.
.
```

```
.
        default:
            feedback = "Подключение...";
}
```

Параметр подключения `connect_timeout` игнорируется, когда используется `PQcancelPoll`; именно приложение отвечает за принятие решения о том, является ли истекшее время чрезмерным. В противном случае вызов `PQcancelStart` с последующим вызовом `PQcancelPoll` в цикле будут эквивалентны вызову `PQcancelBlocking`.

`PQcancelStatus`

Возвращает состояние подключения для отмены.

```
ConnStatusType PQcancelStatus(const PGcancelConn *cancelConn);
```

Статус может принимать одно из ряда значений. Однако только три из них видны извне процедуры асинхронной отмены: `CONNECTION_ALLOCATED`, `CONNECTION_OK` и `CONNECTION_BAD`. Изначальное состояние подключения `PGcancelConn`, успешно созданного через `PQcancelCreate`, — `CONNECTION_ALLOCATED`. Успешно отправленный запрос отмены имеет статус `CONNECTION_OK`. О неудачной попытке отмены сигнализирует статус `CONNECTION_BAD`. Обычно статус OK остаётся таковым до вызова `PQcancelFinish` или `PQcancelReset`.

О других кодах состояния, которые могут выдать эти функции, можно узнать в описании `PQcancelStart`.

Успешная отправка отмены ещё не является гарантией того, что запрос будет иметь какой-то эффект. Если отмена сработала, текущая команда завершится досрочно и возвратит в качестве результата ошибку. Если же отмена не получится (например, потому, что сервер уже завершил обработку команды), тогда видимого результата не будет вовсе.

`PQcancelSocket`

Получает номер файлового дескриптора для сокета отмены соединения с сервером.

```
int PQcancelSocket(const PGcancelConn *cancelConn);
```

Действительный дескриптор будет больше или равен 0; значение -1 показывает, что в данный момент не открыто ни одного соединения с сервером. Значение может измениться в результате вызова любой функции в этом разделе в `PGcancelConn` (кроме `PQcancelErrorMessage` и самой `PQcancelSocket`).

`PQcancelErrorMessage`

Возвращает сообщение об ошибке, наиболее недавно сгенерированное операцией отмены соединения.

```
char *PQcancelErrorMessage(const PGcancelConn *cancelconn);
```

Почти все функции библиотеки libpq, принимающие `PGcancelConn`, в случае сбоя сформируют сообщение для `PQcancelErrorMessage` . Обратите внимание, что по соглашениям, принятым в libpq, непустой результат функции `PQcancelErrorMessage` может состоять из нескольких строк и будет включать завершающий символ новой строки. Вызывающая функция не должна освобождать память, на которую указывает возвращаемое значение, напрямую. Она будет освобождена, когда связанный с ней дескриптор `PGcancelConn` будет передан функции `PQcancelFinish`. Не стоит ожидать, что результирующая строка останется той же самой при выполнении нескольких операций со структурой `PGcancelConn`.

`PQcancelFinish`

Закрывает соединение для отмены (если отправка запроса отмены ещё не завершена). Также освобождает память, используемую объектом `PGcancelConn`.

```
void PQcancelFinish(PGcancelConn *cancelConn);
```

Обратите внимание, что даже если попытка отмены потерпела неудачу (как показывает `PQcancelStatus`), приложение все равно должно вызвать `PQcancelFinish`, чтобы освободить память, используемую объектом `PGcancelConn`. Указатель `PGcancelConn` не должен использоваться повторно после того, как была вызвана функция `PQcancelFinish`.

`PQcancelReset`

Переустанавливает `PGcancelConn` в целях повторного использования для нового соединения для отмены.

```
void PQcancelReset(PGcancelConn *cancelConn);
```

Если `PGcancelConn` используется для отправки запроса отмены, соединение закрывается. Позже будет подготовлен объект `PGcancelConn`, чтобы через него можно было отправить новый запрос отмены.

Это можно использовать для создания `PGcancelConn` для `PGconn` и многократно его переиспользовать на протяжении работы исходного `PGconn`.

32.7.2. Устаревшие функции для отправки запросов отмены

Через данные функции раньше отправлялись запросы отмены. Хотя эти функции всё ещё работают, они признаны устаревшими, поскольку отправляют запросы отмены незашифрованными, даже если в исходном соединении указано требование шифрования `sslmode` или `gssencmode`. По этой причине крайне не рекомендуется использовать их в новом коде, а так же лучше изменить существующий код и вписать в него новые функции.

`PQgetCancel`

Создаёт структуру данных, содержащую информацию, необходимую для отмены команды, запущенной через `PQcancel`.

```
PGcancel *PQgetCancel(PGconn *conn);
```

Функция `PQgetCancel` создаёт объект `PGcancel`, получив объект `PGconn`, описывающий подключение. Она возвратит `NULL`, если параметр *conn* равен `NULL` или представляет недействительное подключение. Объект `PGcancel` является непрозрачной структурой, которая не предназначена для того, чтобы приложение обращалось к ней напрямую; её можно только передавать функции `PQcancel` или `PQfreeCancel`.

`PQfreeCancel`

Освобождает память, занимаемую структурой данных, созданной функцией `PQgetCancel`.

```
void PQfreeCancel(PGcancel *cancel);
```

`PQfreeCancel` освобождает память, занимаемую объектом, предварительно созданным функцией `PQgetCancel`.

`PQcancel`

`PQcancel` является устаревшим и небезопасным аналогом функции `PQcancelBlocking`, но её можно безопасно использовать в рамках обработчика сигналов.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

Функция `PQcancel` существует только в целях обратной совместимости. Вместо неё необходимо использовать `PQcancelBlocking`. Единственное преимущество `PQcancel` состоит в том, что её можно безопасно вызывать из обработчика сигнала, если *errbuf* является локальной переменной в таком обработчике. Однако это преимущество не исключает проблем с безопасностью, связанных с этой функцией, и не является весомой причиной для её использования.

Объект `PGcancel` доступен только в режиме чтения, пока речь идёт о функции `PQcancel`, поэтому её можно также вызывать из потока, отдельного от того, который управляет объектом `PGconn`.

Возвращаемое функцией `PQcancel` значение равно 1, если запрос на отмену был успешно отправлен, и 0 в противном случае. В случае неудачной отправки *`errbuf`* заполняется пояснительным сообщением об ошибке. *`errbuf`* должен быть массивом символов, имеющим размер *`errbufsize`* (рекомендуемый размер составляет 256 байт).

`PQrequestCancel`

`PQrequestCancel` является устаревшим и небезопасным аналогом функции `PQcancelBlocking`.

```
int PQrequestCancel(PGconn *conn);
```

Функция `PQrequestCancel` существует только в целях обратной совместимости. Вместо неё необходимо использовать `PQcancelBlocking`. Нет никаких преимуществ использования `PQrequestCancel` вместо `PQcancelBlocking`.

Выдаёт запрос на то, чтобы сервер прекратил обработку текущей команды. Функция работает напрямую с объектом `PGconn` и в случае сбоя сохраняет сообщение об ошибке в объекте `PGconn` (откуда его можно извлечь с помощью `PQerrorMessage`). Хотя функциональность та же самая, этот подход небезопасен для многопоточных программ или обработчиков сигналов, поскольку перезапись сообщения об ошибке, хранящегося в объекте `PGconn`, может помешать ходу операции, выполняемой через данное подключение.

32.8. Интерфейс быстрого пути

Postgres Pro предоставляет интерфейс для передачи серверу простых вызовов функций по быстрому пути.

Подсказка

Этот интерфейс несколько устарел, поскольку можно достичь подобной производительности и большей функциональности путём создания подготовленного оператора, определяющего вызов функции. Последующее выполнение этого оператора с передачей параметров и результатов в двоичном виде можно считать заменой вызову по быстрому пути.

Функция `PQfn`запрашивает выполнение серверной функции посредством интерфейса быстрого доступа:

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
```

```
} PQArgBlock;
```

Аргумент *`fnid`* представляет собой OID функции, которая подлежит выполнению. *`args`* и *`nargs`* определяют параметры, которые должны быть переданы этой функции; они должны соответствовать списку аргументов объявленной функции. Когда поле *`isint`* структуры, передаваемой в качестве параметра, имеет значение true, тогда значение *`u.integer`* передаётся серверу в виде целого числа указанной длины (это должно быть 2 или 4 байта); при этом устанавливается нужный порядок байтов. Когда *`isint`* имеет значение false, тогда указанное число байт по адресу *`*u.ptr`* отправляется без какой-либо обработки; данные должны быть представлены в формате, которого ожидает сервер для передачи в двоичном виде данных того типа, что и аргументы функции. (Объявление поля *`u.ptr`*, как имеющего тип `int *`, является историческим; было бы лучше рассматривать его как тип `void *`.) *`result_buf`* указывает на буфер, в который должно быть помещено возвращаемое значение функции. Вызывающий код должен выделить достаточное место для сохранения возвращаемого значения. (Это никак не проверяется!) Фактическая длина результирующего значения в байтах будет возвращена в переменной целого типа, на которую указывает *`result_len`*. Если ожидается получение двух- или четырёхбайтового целочисленного результата, то присвойте параметру *`result_is_int`* значение 1, в противном случае назначьте ему 0. Когда параметр *`result_is_int`* равен 1, libpq переставляет байты в передаваемом значении, если это необходимо, так, чтобы оно было доставлено на клиентскую машину в виде правильного значения типа `int`; обратите внимание, что по адресу *`*result_buf`* доставляется четырёхбайтовое целое для любого допустимого размера результата. Когда *`result_is_int`* равен 0, тогда строка байтов в двоичном формате, отправленная сервером, будет возвращена немодифицированной. (В этом случае лучше рассматривать *`result_buf`* как имеющий тип `void *`.)

`PQfn` всегда возвращает действительный указатель на объект `PGresult` со статусом `PGRES_COMMAND_OK` при успешном выполнении функции или `PGRES_FATAL_ERROR`, если произошла какая-то ошибка. Перед использованием результата нужно сначала проверить его статус. Вызывающая функция отвечает за освобождение памяти, занимаемой объектом `PGresult`, когда он больше не нужен, с помощью `PQclear`.

Чтобы передать функции аргумент NULL, запишите в поле *`len`* этой структуры значение `-1`; поля *`isint`* и *`u`* в этом случае не важны.

Если функция возвращает NULL, в *`*result_len`* записывается `-1`, а *`*result_buf`* не изменяется.

Обратите внимание, что при использовании этого интерфейса невозможно обработать множества значений в результате. Кроме того, функция должна быть обычной функцией, а не процедурой, агрегатной или оконной функцией.

32.9. Асинхронное уведомление

Postgres Pro предлагает асинхронное уведомление посредством команд `LISTEN` и `NOTIFY`. Клиентский сеанс работы регистрирует свою заинтересованность в конкретном канале уведомлений с помощью команды `LISTEN` (и может остановить прослушивание с помощью команды `UNLISTEN`). Все сеансы, прослушивающие конкретный канал, будут уведомляться в асинхронном режиме, когда в рамках любого сеанса команда `NOTIFY` выполняется с параметром, указывающим имя этого канала. Для передачи дополнительных данных прослушивающим сеансам может использоваться строка «payload».

Приложения, использующие libpq, отправляют серверу команды `LISTEN`, `UNLISTEN` и `NOTIFY` как обычные SQL-команды. Поступление сообщений от команды `NOTIFY` можно впоследствии отследить с помощью функции `PQnotifies`.

Функция `PQnotifies` возвращает следующее уведомление из списка необработанных уведомительных сообщений, полученных от сервера. Она возвращает нулевой указатель, если нет уведомлений, ожидающих обработки. Как только уведомление возвращено из функции `PQnotifies`, оно считается обработанным и будет удалено из списка уведомлений.

```
PGnotify *PQnotifies(PGconn *conn);
```

```
typedef struct pgNotify
{
    char *relname;              /* имя канала уведомлений */
    int  be_pid;                /* ID серверного процесса, посылающего уведомление */
    char *extra;                /* строка сообщения в уведомлении */
} PGnotify;
```

После обработки объекта `PGnotify`, возвращённого функцией `PQnotifies`, обязательно освободите память, занимаемую им, с помощью функции `PQfreemem`. Достаточно освободить указатель на `PGnotify`; поля `relname` и `extra` не представляют отдельных областей памяти. (Имена этих полей являются таковыми по историческим причинам; в частности, имена каналов не обязаны иметь ничего общего с именами реляционных отношений.)

Пример 32.2 представляет пример программы, иллюстрирующей использование асинхронного уведомления.

Функция `PQnotifies` в действительности не читает данные с сервера; она просто возвращает сообщения, предварительно собранные другой функцией библиотеки libpq. В очень старых выпусках libpq обеспечить своевременное получения сообщений от команды `NOTIFY` можно было только одним способом — постоянно отправлять команды, пусть даже пустые, а затем проверять `PQnotifies` после каждого вызова `PQexec`. Хотя этот метод всё ещё работает, он считается устаревшим ввиду неэффективного использования процессора.

Более удачным способом проверки наличия сообщений от команды `NOTIFY`, когда у вас нет полезных команд для выполнения, является вызов функции `PQconsumeInput` с последующей проверкой `PQnotifies`. Вы можете использовать `select()`, чтобы подождать прибытия данных с сервера, не занимая тем самым ресурсы CPU в отсутствие задач для выполнения. (Получить номер дескриптора для использования с `select()` можно с помощью функции `PQsocket`.) Заметьте, что это будет работать независимо от того, отправляете ли вы команды, используя `PQsendQuery`/`PQgetResult`, или просто вызываете `PQexec`. Однако важно не забывать проверять `PQnotifies` после каждого вызова `PQgetResult` или `PQexec`, чтобы увидеть, не поступили ли какие-либо уведомления в процессе обработки команды.

32.10. Функции, связанные с командой COPY

Команда `COPY` в Postgres Pro имеет возможность читать и записывать данные через сетевое подключение, установленное libpq. Описанные в этом разделе функции позволяют приложениям воспользоваться этой возможностью для передачи или приёма копируемых данных.

Общая процедура такова: сначала приложение выдаёт SQL-команду `COPY`, вызывая `PQexec` или одну из подобных функций. В ответ оно должно получить (если не возникла ошибка) объект `PGresult` с кодом состояния `PGRES_COPY_OUT` или `PGRES_COPY_IN` (в зависимости от направления копирования). Затем приложение должно использовать функции, описанные в этом разделе, и принимать или передавать строки данных. По завершении передачи возвращается ещё один объект `PGresult`, сообщающий о состоянии завершения передачи. В случае успеха он содержит код состояния `PGRES_COMMAND_OK`, а если возникает какая-то проблема — `PGRES_FATAL_ERROR`. После этого можно продолжать выполнять SQL-команды через `PQexec`. (Пока операция `COPY` не завершена, выполнять другие SQL-команды через то же подключение нельзя.)

Если команда `COPY` была выполнена через `PQexec` в строке, содержащей дополнительные команды, приложение должно продолжать получать результаты через `PQgetResult` после завершения последовательности `COPY`. Только когда `PQgetResult` возвращает `NULL`, можно с уверенностью считать, что переданные `PQexec` команды выполнены полностью, и безопасно передавать другие команды.

Функции, описанные в этом разделе, должны выполняться только после получения кода состояния `PGRES_COPY_OUT` или `PGRES_COPY_IN` от функции `PQexec` или `PQgetResult`.

Объект `PGresult` с таким кодом состояния содержит дополнительные данные о начавшейся операции `COPY`. Эти данные можно получить функциями, также применяющимися при обработке результатов запроса:

`PQnfields`

Возвращает число копируемых столбцов (полей).

`PQbinaryTuples`

Значение 0 указывает, что для всей операции копирования применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами-разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат. За дополнительными сведениями обратитесь к COPY.

`PQfformat`

Возвращает код формата (0 — текстовый, 1 — двоичный), связанный с каждым копируемым столбцом. Коды форматов столбцов всегда будут нулевыми, если общий формат копирования — текстовый, но с двоичным форматом поддерживаются и текстовые, и двоичные столбцы. (Однако в текущей реализации `COPY` при двоичном копировании столбцы могут быть только двоичными, так что форматы столбцов должны всегда соответствовать общему формату.)

32.10.1. Функции для передачи данных COPY

Эти функции применяются для передачи данных при операции `COPY FROM STDIN`. Они не будут работать, если подключение находится не в состоянии `COPY_IN`.

`PQputCopyData`

Отправляет данные на сервер, когда активно состояние `COPY_IN`.

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

Передаёт серверу данные `COPY` из указанного буфера (*buffer*), длиной *nbytes* байт. Она возвращает 1, если данные были переданы, 0, если они не попали в очередь, так как буферы были заполнены (это возможно только в неблокирующем режиме), или -1, если произошла ошибка. (Если возвращено -1, подробности ошибки можно узнать, вызвав `PQerrorMessage` . Если получен 0, дождитесь состояния готовности к записи и повторите попытку.)

Приложение может разделять поток данных `COPY` на буферизуемые блоки любого удобного размера. Границы буфера не имеют семантического значения при передаче. Содержимое потока данных должно соответствовать формату данных, ожидаемому командой `COPY`; за подробностями обратитесь к COPY.

`PQputCopyEnd`

Отправляет признак конца данных на сервер, когда активно состояние `COPY_IN`.

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

Завершает операцию `COPY_IN` с успешным результатом, если в *errormsg* передаётся `NULL`. Если *errormsg* не `NULL`, команда `COPY` будет завершена с ошибкой, а сообщением об ошибке будет строка, переданная в *errormsg*. (Однако не следует полагать, что именно это сообщение будет получено от сервера назад, так как сервер мог уже прервать операцию `COPY` по своим причинам.)

Эта функция возвращает 1, если сообщение завершения было передано; в неблокирующем режиме это означает только, что сообщение завершения успешно поставлено в очередь. (Чтобы удостовериться, что данные были успешно отправлены в неблокирующем режиме, следует дождаться готовности к записи и вызывать `PQflush` в цикле, пока она не вернёт ноль.) Нулевой результат означает, что функция не смогла поставить сообщение завершения в очередь по причине заполнения буферов; это возможно только в неблокирующем режиме. (В этом случае нужно дождаться готовности к записи и попытаться вызвать `PQputCopyEnd` снова.) Если действительно происходит ошибка, возвращается -1; получить её подробности можно, вызвав `PQerrorMessage` .

После успешного вызова `PQputCopyEnd` вызовите `PQgetResult`, чтобы узнать окончательный результат команды `COPY`. Ожидать появления этого результата можно обычным образом. Затем вернитесь к обычным операциям.

32.10.2. Функции для приёма данных COPY

Эти функции применяются для получения данных при операции `COPY TO STDOUT`. Они не будут работать, если подключение находится не в состоянии `COPY_OUT`.

`PQgetCopyData`

Принимает данные от сервера, когда активно состояние `COPY_OUT`.

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

Запрашивает следующую строку данных с сервера в процессе операции `COPY`. Данные всегда возвращаются строка за строкой; если поступила только часть строки, она не возвращается. Успешное получение строки данных подразумевает выделение блока памяти для этих данных. В параметре *`buffer`* ей передаётся указатель, отличный от `NULL`. По адресу *`*buffer`* записывается указатель на выделенную память, либо `NULL`, когда буфер не возвращается. Если буфер результата отличен от `NULL`, его следует освободить, когда он станет не нужен, вызвав `PQfreemem`.

Когда строка получена успешно, возвращается число байт данных в этой строке (это число всегда больше нуля). Возвращаемое строковое значение всегда завершается нулём, хотя это полезно, вероятно, только для текстовой `COPY`. Нулевой результат означает, что операция `COPY` продолжает выполняться, но строка ещё не готова (это возможно, только когда параметр *`async`* равен true). Возвращённое значение -1 означает, что команда `COPY` завершена, а -2 показывает, что произошла ошибка (её причину можно узнать с помощью `PQerrorMessage`).

Когда параметр *`async`* отличен от нуля (признак установлен), функция `PQgetCopyData` не будет блокироваться, ожидая данных; она возвратит ноль, если выполнение `COPY` продолжается, но полная строка ещё не получена. (В этом случае нужно дождаться готовности к чтению и затем вызвать `PQconsumeInput` , прежде чем вызывать `PQgetCopyData` ещё раз.) Когда *`async`* равен нулю (признак не установлен), `PQgetCopyData` будет заблокирована до поступления данных или окончания операции.

Когда `PQgetCopyData` возвращает -1, вызовите `PQgetResult`, чтобы узнать окончательный результат команды `COPY`. Ожидать появления этого результата можно обычным образом. Затем вернитесь к обычным операциям.

32.10.3. Устаревшие функции для COPY

Эти функции представляют старые методы выполнения операции `COPY`. Хотя они продолжают работать, они признаны устаревшими из-за плохой обработки ошибок, неудобных способов обнаружения конца данных и отсутствия поддержки двоичных или неблокирующих передач.

`PQgetline`

Читает передаваемую сервером строку символов, завершающуюся символом новой строки, в буфер (buffer) размера *`length`*.

```
int PQgetline(PGconn *conn,
```

```
              char *buffer,
              int length);
```

Эта функция копирует *`length`*-1 символов в буфер и преобразует символ конца строки в нулевой байт. `PQgetline` возвращает `EOF` в конце ввода, 0, если была прочитана вся строка, и 1, если буфер заполнен, но завершающий символ конца строки ещё не прочитан.

Заметьте, что приложение должно проверить, не состоит ли новая строка в точности из двух символов `\.`, что будет означать, что сервер завершил передачу результатов команды `COPY`. Если приложение может принимать строки длиннее *`length`*-1 символов, необходимо позаботиться о том, чтобы оно корректно распознавало строку `\.` (а не воспринимало, например, конец длинной строки данных как завершающую строку).

`PQgetlineAsync`

Читает передаваемую сервером строку данных `COPY` в буфер без блокировки.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

Эта функция похожа на `PQgetline`, но может применяться в приложениях, которые должны читать данные `COPY` асинхронно, то есть без блокировки. Запустив команду `COPY` и получив ответ `PGRES_COPY_OUT`, приложение должно вызывать `PQconsumeInput` и `PQgetlineAsync`, пока не будет получен сигнал «конец данных».

В отличие от `PQgetline`, эта функция сама отвечает за обнаружение конца данных.

При каждом вызове `PQgetlineAsync` будет возвращать данные, если во входном буфере libpq оказывается полная строка данных. В противном случае никакие данные не возвращаются до поступления остального содержимого строки. Эта функция возвращает -1, если обнаруживается признак завершения копирования, или 0, если данные не получены, или положительное количество возвращённых байт данных. Если возвращается -1, вызывающий код должен затем вызвать `PQendcopy` и после этого перейти в обычный режим работы.

Возвращаемые данные не будут пересекать границы строк данных. При этом может быть возвращена одна строка целиком. Но если буфер, выделенный вызывающим кодом, оказывается слишком мал для строки, передаваемой сервером, возвращена будет часть строки. Когда передаются текстовые данные, это можно выявить, проверив, содержит ли последний возвращаемый байт символ `\n`. (Для `COPY` в двоичном формате потребуется собственно разобрать формат данных `COPY`, чтобы выявить подобную ситуацию.) Возвращаемая строка не завершается нулём. (Если вы хотите получить строку с нулём в конце, передайте в *`bufsize`* число на единицу меньше фактического размера блока.)

`PQputline`

Передаёт серверу строку, завершённую нулём. Возвращает 0 в случае успеха, либо `EOF`, если передать строку не удаётся.

```
int PQputline(PGconn *conn,
              const char *string);
```

Поток данных `COPY`, передаваемых последовательностью вызовов `PQputline`, имеет тот же формат, что возвращает `PQgetlineAsync`, за исключением того, что приложения не обязательно должны передавать по одной строке данных за вызов `PQputline`; они могут посылать части строк или сразу несколько строк.

Примечание

До версии 3.0 протокола Postgres Pro приложение должно было явно отправлять два символа `\.` последней строкой, чтобы сообщить серверу, что оно закончило передачу данных

> `COPY`. Хотя это по-прежнему работает, такое поведение считается устаревшим и ожидается, что особое значение `\.` будет исключено в будущих версиях. Сейчас, передав собственно данные, достаточно вызвать `PQendcopy`.

`PQputnbytes`

Передаёт серверу строку, не завершённую нулём. Возвращает 0 в случае успеха, либо `EOF`, если передать строку не удаётся.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

Поведение этой функции не отличается от `PQputline`, но её буфер данных не должен содержать завершающий ноль, так как для неё число передаваемых байт задаётся непосредственным образом. Используйте эту функцию для передачи двоичных данных.

`PQendcopy`

Производит синхронизацию с сервером.

```
int PQendcopy(PGconn *conn);
```

Эта функция ожидает завершения копирования сервером. Её следует вызывать, либо когда серверу была передана последняя строка функцией `PQputline`, либо когда от сервера была получена последняя строка функцией `PQgetline`. Если её не вызвать, сервер «потеряет синхронизацию» с клиентом. После завершения этой функции сервер готов принимать следующую команду SQL. В случае успешного завершения возвращается 0, в противном случае — ненулевое значение. (Чтобы получить подробности ошибки при ненулевом значении, вызовите `PQerrorMessage` .)

Вызывая `PQgetResult`, приложение должно обрабатывать результат `PGRES_COPY_OUT`, в цикле выполняя `PQgetline`, а обнаружив завершающую строку, вызвать `PQendcopy`. Затем оно должно вернуться к циклу `PQgetResult` и выйти из него, когда `PQgetResult` возвратит нулевой указатель. Подобным образом, получив результат `PGRES_COPY_IN`, приложение должно выполнить серию вызовов `PQputline`, завершить её, вызвав `PQendcopy`, а затем вернуться к циклу `PQgetResult`. При такой организации обработки команда `COPY` будет корректно выполняться и в составе последовательности команд SQL.

Старые приложения обычно передают команду `COPY` через `PQexec` и рассчитывают, что транзакция будет завершена после `PQendcopy`. Это будет работать, только если команда `COPY` является единственной SQL-командой в строке запроса.

32.11. Функции управления

Эти функции управляют различными аспектами поведения libpq.

`PQclientEncoding`

Возвращает кодировку клиента.

```
int PQclientEncoding(const PGconn *conn);
```

Заметьте, что она возвращает идентификатор кодировки, а не символьную строку вида `EUC_JP`. В случае ошибки она возвращает -1. Преобразовать идентификатор кодировки в имя можно, воспользовавшись следующей функцией:

```
char *pg_encoding_to_char(int encoding_id);
```

`PQsetClientEncoding`

Устанавливает кодировку клиента.

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

В *`conn`* передаётся соединение с сервером, а в *`encoding`* — имя требуемой кодировки. Если функция устанавливает кодировку успешно, она возвращает 0, или -1 в противном случае. Определить текущую кодировку для соединения можно, воспользовавшись функцией `PQclientEncoding`.

`PQsetErrorVerbosity`

Определяет уровень детализации сообщений, возвращаемых функциями `PQerrorMessage` и `PQresultErrorMessage`.

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE,
    PQERRORS_SQLSTATE
} PGVerbosity;

PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

`PQsetErrorVerbosity` устанавливает режим детализации и возвращает предыдущее значение для соединения. В «лаконичном» режиме (*TERSE*) возвращаемые сообщения содержат только уровень важности, основной текст и позицию; всё это обычно умещается в одной строке. В режиме по умолчанию (*DEFAULT*) выдаваемые сообщения дополнительно содержат поля подробного описания, подсказки или контекста (они могут занимать несколько строк). В «многословном» режиме (*VERBOSE*) передаются все доступные поля сообщения. В режиме *SQLSTATE* выдаётся только уровень важности и код ошибки `SQLSTATE`, если он имеется (если же его нет, выводится та же информация, что и в режиме *TERSE*).

Изменение уровня детализации не влияет на сообщения, уже сформированные в существующих объектах `PGresult`, а затрагивает только последующие сообщения. (Но можно воспользоваться `PQresultVerboseErrorMessage`, чтобы получить предыдущую ошибку с другим уровнем детализации.)

`PQsetErrorContextVisibility`

Определяет вариант обработки полей `CONTEXT` в сообщениях, возвращаемых функциями `PQerrorMessage` и `PQresultErrorMessage`.

```
typedef enum
{
    PQSHOW_CONTEXT_NEVER,
    PQSHOW_CONTEXT_ERRORS,
    PQSHOW_CONTEXT_ALWAYS
} PGContextVisibility;

PGContextVisibility PQsetErrorContextVisibility(PGconn *conn, PGContextVisibility
 show_context);
```

`PQsetErrorContextVisibility` устанавливает режим вывода контекста и возвращает предыдущее значение для соединения. Этот режим определяет, будет ли поле `CONTEXT` включаться в сообщения. В режиме *NEVER* поле `CONTEXT` не включается никогда, а в режиме *ALWAYS* включается всегда, при наличии. В режиме *ERRORS* (по умолчанию) поле `CONTEXT` включается только в сообщения об ошибках, но не в замечания и предупреждения. (Однако при уровне детализации *TERSE* или *SQLSTATE* поле `CONTEXT` опускается вне зависимости от режима вывода контекста.)

Смена этого режима не влияет на сообщения, уже сформированные в существующих объектах `PGresult`, а затрагивает только последующие сообщения. (Но можно воспользоваться `PQresultVerboseErrorMessage`>, чтобы получить предыдущую ошибку в другом режиме вывода.)

`PQtrace`

Включает трассировку клиент-серверного взаимодействия с выводом в поток отладочных сообщений.

```
void PQtrace(PGconn *conn, FILE *stream);
```

Каждая строка содержит: необязательную отметку времени, индикатор направления (`F` для сообщений клиента серверу или `B` для сообщений сервера клиенту), длину сообщения, тип сообщения и содержимое сообщения. Поля содержимого, не связанные с сообщением (отметка времени, направление, длина и тип сообщения), разделяются табуляцией. Содержимое сообщения разделяется пробелами. Строки протокола заключаются в двойные кавычки, а строки, используемые в качестве значений данных, — в апострофы. Непечатаемые символы выводятся в виде шестнадцатеричных спецпоследовательностей. Дополнительную информацию о типах сообщений можно найти в Разделе 53.7.

Примечание

В Windows, если библиотека libpq и приложение скомпилированы с разными флагами, эта функция может вызвать крах приложения из-за различий внутреннего представления указателей `FILE`. В частности, флаги многопоточной/однопоточной, выпускаемой/отладочной или статической/динамической сборки должны быть одинаковыми для библиотеки и всех использующих её приложений.

`PQsetTraceFlags`

Управляет поведением трассировки клиент-серверного взаимодействия.

```
void PQsetTraceFlags(PGconn *conn, int flags);
```

`flags` содержит биты флагов, описывающие режим работы трассировки. Если `flags` содержит `PQTRACE_SUPPRESS_TIMESTAMPS`, метка времени не добавляется при печати каждого сообщения. Если параметр `flags` содержит `PQTRACE_REGRESS_MODE`, некоторые поля в выводимых сообщениях редактируются, в частности заменяются OID объектов, чтобы этот вывод можно было использовать при тестировании. Эта функция должна вызываться после вызова `PQtrace`.

`PQuntrace`

Выключает трассировку, запущенную функцией `PQtrace`.

```
void PQuntrace(PGconn *conn);
```

32.12. Функции разного назначения

Как всегда, находятся функции, которые не попадают ни в одну из категорий.

`PQfreemem`

Освобождает память, которую выделила libpq.

```
void PQfreemem(void *ptr);
```

Освобождает память, выделенную библиотекой libpq, а именно функциями `PQescapeByteaConn`, `PQescapeBytea`, `PQunescapeBytea` и `PQnotifies`. Особенно важно использовать именно эту функцию, а не `free()`, в Microsoft Windows. Это связано с тем, что выделение памяти в DLL и освобождение её в приложении будет работать, только если флаги многопоточной/однопоточной, выпускаемой/отладочной или статической/динамической сборки для DLL и приложения полностью совпадают. На других платформах эта функция действует так же, как стандартная библиотечная функция `free()`.

`PQconninfoFree`

Освобождает структуры данных, выделенные функциями `PQconndefaults` и `PQconninfoParse`.

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

Если аргумент является указателем `NULL`, операция не выполняется.

Простая функция `PQfreemem` не подойдёт для этого, так как эти структуры содержат ссылки на подчинённые строки.

`PQencryptPasswordConn`

Подготавливает зашифрованную форму пароля Postgres Pro.

```
char *PQencryptPasswordConn(PGconn *conn, const char *passwd, const char *user,
 const char *algorithm);
```

Эта функция предназначена для клиентских приложений, желающих передавать команды вида `ALTER USER joe PASSWORD 'pwd'`. В такой команде лучше не передавать исходный пароль открытым текстом, так как он может появиться в журналах команд, мониторе активности и т. д. Вместо этого воспользуйтесь данной функцией и переведите пароль в зашифрованную форму.

В аргументах *`passwd`* и *`user`* задаётся пароль в открытом виде и SQL-имя пользователя, для которого он задаётся. В аргументе *`algorithm`* задаётся алгоритм для шифрования пароля. В настоящее время поддерживаются алгоритмы `md5` и `scram-sha-256` (в качестве альтернативных обозначений `md5` для совместимости со старыми версиями сервера поддерживаются значения `on` и `off`). Заметьте, что поддержка `scram-sha-256` появилась в Postgres Pro версии 10 и со старыми версиями серверов этот вариант работать не будет. Если *`algorithm`* равен `NULL`, эта функция запросит у сервера текущее значение параметра password_encryption. При этом возможна блокировка и отказ при выполнении функции, если текущая транзакция прерывается или если в текущем соединении выполняется другой запрос. Если вы хотите использовать алгоритм по умолчанию для данного сервера, но при этом избежать блокировки, получите значение `password_encryption` самостоятельно до вызова `PQencryptPasswordConn` и передайте его в параметре *`algorithm`*.

Эта функция возвращает строку, выделенную функцией `malloc`. Вызывающий код может рассчитывать на то, что эта строка не содержит никаких спецсимволов, требующих экранирования. Закончив работу с ней, освободите память, вызвав `PQfreemem`. В случае ошибки эта функция возвращает `NULL`, а соответствующее сообщение помещается в объект соединения.

`PQchangePassword`

Меняет пароль Postgres Pro.

```
PGresult *PQchangePassword(PGconn *conn, const char *user, const char *passwd);
```

Данная функция использует `PQencryptPasswordConn` для создания и выполнения команды `ALTER USER ... PASSWORD '...'`, что приводит к изменению пароля. Она используется для того же, что и `PQencryptPasswordConn`, но является предпочтительней, поскольку и создаёт, и выполняет команду. Функции `PQencryptPasswordConn` передаётся `NULL` в качестве аргумента алгоритма, таким образом шифрование происходит в соответствии с конфигурацией сервера password_encryption.

В аргументах *`passwd`* и *`user`* задаётся пароль в открытом виде и SQL-имя пользователя, для которого он задаётся.

Возвращает указатель на `PGresult` как результат выполнения команды `ALTER USER` или, возможно, пустой указатель (null), если произошёл сбой до запуска команды. Для проверки возвращаемого значения на наличие ошибок следует вызвать функцию `PQresultStatus` (в случае нулевого указателя она возвратит `PGRES_FATAL_ERROR`). Для получения дополнительной информации о таких ошибках используйте функцию `PQerrorMessage` .

`PQencryptPassword`

Подготавливает зашифрованную md5 форму пароля Postgres Pro.

```
char *PQencryptPassword(const char *passwd, const char *user);
```

`PQencryptPassword` — старая, подлежащая ликвидации версия `PQencryptPasswordConn`. Отличие состоит в том, что для `PQencryptPassword` не требуется объект соединения, а в качестве алгоритма шифрования всегда используется `md5`.

`PQmakeEmptyPGresult`

Конструирует пустой объект `PGresult` с указанным состоянием.

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

Это внутренняя функция libpq, выделяющая память и инициализирующая пустой объект `PGresult`. Эта функция возвращает `NULL`, если не может выделить память. Она сделана экспортируемой, так как некоторые приложения находят полезным создавать объекты результатов (в частности, объекты с состоянием ошибки) самостоятельно. Если в *`conn`* передаётся не null и *`status`* указывает на ошибку, в `PGresult` копируется текущее сообщение об ошибке для заданного соединения. Также, если в *`conn`* передаётся не null, в `PGresult` копируются все процедуры событий, зарегистрированные для этого соединения. (При этом вызовы `PGEVT_RESULTCREATE` не выполняются; см. описание `PQfireResultCreateEvents`.) Заметьте, что в конце для этого объекта следует вызвать `PQclear`, как и для объекта `PGresult`, возвращённого самой библиотекой libpq.

`PQfireResultCreateEvents`

Вызывает событие `PGEVT_RESULTCREATE` (см. Раздел 32.14) для каждой процедуры событий, зарегистрированной в объекте `PGresult`. Возвращает ненулевое значение в случае успеха или ноль в случае ошибки в одной из процедур.

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

Аргумент `conn` передаётся процедурам событий, но непосредственно не используется. Он может быть равен `NULL`, если он не нужен процедурам событий.

Процедуры событий, уже получившие событие `PGEVT_RESULTCREATE` или `PGEVT_RESULTCOPY` для этого объекта, больше не вызываются.

Основная причина отделения этой функции от `PQmakeEmptyPGresult` в том, что часто требуется создать объект `PGresult` и наполнить его данными, прежде чем вызывать процедуры событий.

`PQcopyResult`

Создаёт копию объекта `PGresult`. Эта копия никак не связана с исходным результатом и поэтому, когда она становится не нужна, необходимо вызвать `PQclear`. Если функция завершается ошибкой, она возвращает `NULL`.

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

Создаваемая копия не будет точной. В возвращаемый результат всегда помещается состояние `PGRES_TUPLES_OK` и в него не копируются никакие сообщения об ошибках из исходного объекта. (Однако в него копируется строка состояния команды.) Что ещё в него будет копироваться, определяет аргумент *`flags`*, в котором складываются несколько флагов. Флаг `PG_COPYRES_ATTRS` включает копирование атрибутов исходного объекта (определений столбцов), а флаг `PG_COPYRES_TUPLES` включает копирование кортежей из исходного объекта (при этом также копируются и атрибуты.) Флаг `PG_COPYRES_NOTICEHOOKS` включает копирование обработчиков замечаний, а флаг `PG_COPYRES_EVENTS` — событий из исходного объекта результата. (Но любые данные, связанные с экземпляром исходного объекта, не копируются.) Процедуры событий получают события `PGEVT_RESULTCOPY`.

`PQsetResultAttrs`

Устанавливает атрибуты объекта `PGresult`.

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc *attDescs);
```

Предоставленная структура *attDescs* копируется в результат. Если указатель *attDescs* равен `NULL` или *numAttributes* меньше одного, запрос игнорируется и функция выполняется без ошибки. Если *res* уже содержит атрибуты, функция завершается ошибкой. В случае ошибки функция возвращает ноль, а в обратном случае — ненулевое значение.

`PQsetvalue`

Устанавливает значение поля кортежа в объекте `PGresult`.

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int len);
```

Эта функция автоматически увеличивает внутренний массив кортежей при необходимости. Однако значение *tup_num* должно быть меньше или равно `PQntuples`, что означает, что эта функция может увеличивать массив кортежей только на один кортеж. Но в существующем кортеже любые поля могут изменяться в любом порядке. Если значение в поле с номером *field_num* уже существует, оно будет перезаписано. Если *len* равно -1 или *value* равно `NULL`, в поле будет записано значение SQL NULL. Устанавливаемое значение (*value*) копируется в закрытую область объекта результата, так что от него можно избавиться после завершения функции. Если функция завершается ошибкой, она возвращает ноль, а в обратном случае — ненулевое значение.

`PQresultAlloc`

Выделяет подчинённую область памяти для объекта `PGresult`.

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

Любая память, выделенная этой функцией, будет освобождена при очистке объекта *res*. В случае ошибки эта функция возвращает `NULL`. Результат гарантированно выравнивается должным образом для любого типа данных, как и при `malloc`.

`PQresultMemorySize`

Выдаёт объём памяти (в байтах), выделенной для объекта `PGresult`.

```
size_t PQresultMemorySize(const PGresult *res);
```

Этот объём равен сумме размеров всех запросов `malloc`, связанных с данным объектом `PGresult`, то есть это общий объём памяти, который будет освобождён функцией `PQclear`. Данная информация может быть полезна для управления потреблением памяти.

`PQlibVersion`

Возвращает версию используемой библиотеки libpq.

```
int PQlibVersion(void);
```

По результату этой функции можно во время выполнения определить, предоставляется ли определённая функциональность загруженной в данный момент версией libpq. Эта функция может использоваться, например, чтобы понять, какие параметры соединения может принять `PQconnectdb`.

Это число формируется в результате умножения номера основной версии библиотеки на 10000 и добавления номера дополнительной версии. Например, для версии 10.1 будет выдано 100001, а для версии 11.0 — 110000.

До версии 10, в Postgres Pro номера версий образовывались из трёх чисел, первые два из которых представляли основную версию. Для таких версий `PQlibVersion` отводит на каждое число по две цифры; например, для версии 9.1.5 будет выдано 90105, а для версии 9.2.0 — 90200.

Таким образом, чтобы получить логический номер версии для определения доступности функциональности, приложения должны разделить результат `PQlibVersion` на 100, а не на 10000. Во всех сериях номера дополнительных (корректирующих) выпусков различаются только в двух последних цифрах.

Примечание

Эта функция появилась в PostgreSQL версии 9.1, поэтому с её помощью нельзя проверить функциональность предыдущих версий, так как при вызове её будет создана зависимость от версии 9.1 или новее.

`PQgetCurrentTimeUSec`

Выдаёт текущее время в микросекундах в формате Unix (то есть `time_t`, умноженное на 1 миллион).

```
pg_usec_time_t PQgetCurrentTimeUSec(void);
```

Главным образом это удобно для расчёта тайм-аута для использования с функцией `PQsocketPoll`.

32.13. Обработка замечаний

Сообщения с замечаниями и предупреждениями, выдаваемые сервером, не возвращаются функциями, выполняющими запросы, так как они не свидетельствуют об ошибке в запросе. Вместо этого они передаются функции обработки замечаний и после завершения этой функции выполнение продолжается как обычно. Стандартная функция обработки замечаний выводит сообщение в `stderr`, но приложение может переопределить это поведение, предоставив собственный обработчик.

По историческим причинам обработка замечаний выполняется на двух уровнях, приёмником замечаний и обработчиком замечаний. По умолчанию приёмник замечаний форматирует замечание и передаёт сформированную строку обработчику замечаний для вывода. Однако приложения, которые реализуют свой приёмник замечаний, обычно просто игнорируют слой обработчика и выполняют все действия в коде приёмника.

Функция `PQsetNoticeReceiver` устанавливает или возвращает текущий приёмник замечаний для объекта соединения. Подобным образом, `PQsetNoticeProcessor` устанавливает или возвращает текущий обработчик замечаний.

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

Каждая из этих функций возвращает указатель на ранее установленный приёмник или обработчик замечаний и устанавливает новый указатель. Если ей передаётся нулевой указатель, она ничего не делает, только возвращает текущий указатель.

Когда сообщение с замечанием или предупреждением поступает от сервера, либо выдаётся самой библиотекой libpq, вызывается функция приёмника замечания. Сообщение передаётся ей в виде состояния `PGRES_NONFATAL_ERROR` объекта `PGresult`. (Это позволяет приёмнику извлечь из него отдельные поля, используя `PQresultErrorField`, либо получить полное готовое сообщение, вызвав `PQresultErrorMessage` или `PQresultVerboseErrorMessage`.) Ей также передаётся тот же неопределённый указатель, что был передан функции `PQsetNoticeReceiver`. (Этот указатель может пригодиться для обращения к внутреннему состоянию приложения при необходимости.)

Стандартный приёмник замечаний просто извлекает сообщение (вызывая `PQresultErrorMessage`) и передаёт его обработчику замечаний.

Обработчик замечаний отвечает за обработку сообщения с замечанием или предупреждением в текстовом виде. Ему передаётся строка с текстом сообщения (включающая завершающий символ новой строки) и неопределённый указатель, который был передан функции `PQsetNoticeProcessor`. (Этот указатель может пригодиться для обращения к внутреннему состоянию приложения при необходимости.)

Стандартный обработчик замечаний прост:

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

Установив приёмник или обработчик замечаний, вы можете ожидать, что эта функция будет вызываться, пока будут существовать объект `PGconn` или объекты `PGresult`, созданные с ней. Когда создаётся `PGresult`, указатели текущих обработчиков замечаний, установленные в `PGconn`, копируются в `PGresult` для возможного использования функциями вроде `PQgetvalue`.

32.14. Система событий

Система событий libpq разработана для уведомления функций-обработчиков об интересных событиях libpq, например, о создании и уничтожении объектов `PGconn` и `PGresult`. Основное их предназначение в том, чтобы позволить приложениям связать собственные данные с объектами `PGconn` и `PGresult` и обеспечить их освобождение в нужное время.

Каждый зарегистрированный обработчик событий связывается с двумя элементами данных, которые известны libpq только как скрытые указатели `void *`. Первый *сквозной* указатель передаётся приложением, когда обработчик событий регистрируется в `PGconn`. Этот указатель никогда не меняется на протяжении жизни `PGconn` и все объекты `PGresult` создаются с ним; поэтому, если он используется, он должен указывать на долгоживущие данные. В дополнение к нему имеется указатель *данных экземпляра*, который изначально равен `NULL` во всех объектах `PGconn` и `PGresult`. Этим указателем можно управлять с помощью функций `PQinstanceData`, `PQsetInstanceData`, `PQresultInstanceData` и `PQresultSetInstanceData`. Заметьте, что в отличие от сквозного указателя, данные экземпляра `PGconn` автоматически не наследуются объектами `PGresult`, создаваемыми из него. Библиотека libpq не знает, на что указывают сквозной указатель и указатель данных экземпляра (если они ненулевые), и никогда не будет пытаться освобождать их — за это отвечает обработчик событий.

32.14.1. Типы событий

Перечисление `PGEventId` описывает типы событий, обрабатываемых системой событий. Имена всех их значений начинаются с `PGEVT`. Для каждого типа событий имеется соответствующая структура информации о событии, содержащая параметры, передаваемые обработчикам событий. Определены следующие типы событий:

`PGEVT_REGISTER`

Событие регистрации происходит, когда вызывается `PQregisterEventProc`. Это подходящий момент для инициализации данных экземпляра (`instanceData`), которые могут понадобиться процедуре событий. Для каждого обработчика событий в рамках соединения будет выдаваться только одно событие регистрации. Если обработка события завершается ошибкой (возвращается ноль), регистрация отменяется.

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

При поступлении события `PGEVT_REGISTER` указатель *evtInfo* следует привести к `PGEventRegister *`. Эта структура содержит объект `PGconn`, который должен быть в состоянии `CONNECTION_OK`; это гарантируется, если `PQregisterEventProc` вызывается сразу после получения рабочего объекта `PGconn`. В случае выдачи кода ошибки всю очистку необходимо провести самостоятельно, так как событие `PGEVT_CONNDESTROY` не поступит.

`PGEVT_CONNRESET`

Событие сброса соединения происходит при завершении `PQreset` или `PQresetPoll`. В обоих случаях это событие вызывается, только если сброс был успешным. Значение, возвращаемое процедурой события, в Postgres Pro v15 и более поздних версиях игнорируется. Однако с более ранними версиями важно возвращать успешный результат (отличный от нуля), иначе соединение будет прервано.

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

При поступлении события `PGEVT_CONNRESET` указатель *evtInfo* следует привести к `PGEventConnReset *`. Хотя переданный объект `PGconn` был только что сброшен, все данные события остаются неизменными. При поступлении этого события должны быть сброшены/перезагружены/вновь запрошены все сопутствующие данные `instanceData`. Заметьте, что даже если обработчик события выдаст ошибку при обработке `PGEVT_CONNRESET`, событие `PGEVT_CONNDESTROY` всё равно поступит при закрытии соединения.

`PGEVT_CONNDESTROY`

Событие уничтожения соединения вызывается в ответ на вызов `PQfinish`. Обработчик этого события отвечает за корректную очистку своих данных событий, так как libpq не может управлять его памятью. Невыполнение очистки должным образом приведёт к утечкам памяти.

```
typedef struct
{
    PGconn *conn;
} PGEventConnDestroy;
```

При поступлении события `PGEVT_CONNDESTROY` указатель *evtInfo* следует привести к `PGEventConnDestroy *`. Это событие происходит перед тем, как `PQfinish` производит всю остальную очистку. Значение, возвращаемое обработчиком событий, игнорируется, так как из `PQfinish` никак нельзя сообщить об ошибке. Кроме того, ошибка в обработчике событий не должна прерывать процесс очистки ставшей ненужной памяти.

`PGEVT_RESULTCREATE`

Событие создания объекта результата происходит при завершении любой функции, выполняющей запрос и получающей результат, включая `PQgetResult`. Это событие происходит только после того, как результат был успешно создан.

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

При поступлении события `PGEVT_RESULTCREATE` указатель *evtInfo* следует привести к `PGEventResultCreate *`. В *conn* передаётся соединение, для которого сформирован результат. Это подходящее место для инициализации любых данных `instanceData`, которые нужно связать с результатом. Если процедура события завершится ошибкой (возвратит ноль), она будет игнорироваться для данного результата, пока он будет существовать; то есть эта процедура не будет получать события `PGEVT_RESULTCOPY` и `PGEVT_RESULTDESTROY` для него и для результатов, скопированных из него.

`PGEVT_RESULTCOPY`

Событие копирования объекта результата происходит при выполнении функции `PQcopyResult`. Это событие происходит только после завершения копирования. Только те процедуры событий, которые успешно обработали событие `PGEVT_RESULTCREATE` или `PGEVT_RESULTCOPY` для исходного результата, получат событие `PGEVT_RESULTCOPY`.

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

При поступлении события `PGEVT_RESULTCOPY` указатель *`evtInfo`* следует привести к `PGEventResultCopy *`. Поле *`src`* указывает на объект результата, который копируется, а *`dest`* — на целевой объект. Это событие может применяться для реализации внутреннего копирования `instanceData`, так как сама функция `PQcopyResult` не может это сделать. Если процедура события завершится ошибкой (возвратит ноль), она будет игнорироваться для данного результата, пока он будет существовать; то есть эта процедура не будет получать события `PGEVT_RESULTCOPY` или `PGEVT_RESULTDESTROY` для него или для результатов, скопированных из него.

`PGEVT_RESULTDESTROY`

Событие уничтожения объекта результата происходит при выполнении `PQclear`. Обработчик этого события отвечает за корректную очистку своих данных событий, так как libpq не может управлять его памятью. Невыполнение очистки должным образом приведёт к утечкам памяти.

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

При поступлении события `PGEVT_RESULTDESTROY` указатель *`evtInfo`* следует привести к `PGEventResultDestroy *`. Это событие происходит перед тем, как `PQclear` производит всю остальную очистку. Значение, возвращаемое обработчиком событий, игнорируется, так как из `PQclear` никак нельзя сообщить об ошибке. Кроме того, ошибка в обработчике событий не должна прерывать процесс очистки ставшей ненужной памяти.

32.14.2. Процедура обработки событий

`PGEventProc`

`PGEventProc` — это определение типа для указателя на обработчик событий, то есть функцию обратного вызова, получающую события от libpq. Обработчик событий должен иметь такую сигнатуру:

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

Параметр *`evtId`* говорит, какое событие `PGEVT` произошло. Указатель *`evtInfo`* должен приводиться к типу определённой структуры для получения дополнительной информации о событии. В параметре *`passThrough`* передаётся сквозной указатель, поступивший в `PQregisterEventProc` при регистрации обработчика события. Эта функция должна вернуть ненулевое значение в случае успеха или ноль в противном случае.

Обработчик определённого события может быть зарегистрирован в любом `PGconn` только раз. Это связано с тем, что адрес обработчика используется как ключ для выбора связанных данных экземпляра.

Внимание

В Windows функции могут иметь два разных адреса: один, видимый снаружи DLL, и второй, видимый внутри DLL. Учитывая это, надо позаботиться о том, чтобы только один из адресов использовался с функциями обработки событий libpq, иначе возникнет путаница. Самый простой способ написать код, который будет работать — всегда помечать обработчик событий как `static`. Если адрес обработчика нужно получить вне его исходного файла, экспортируйте отдельную функцию, которая будет возвращать этот адрес.

32.14.3. Функции поддержки событий

`PQregisterEventProc`

Регистрирует обработчик событий в libpq.

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void *passThrough);
```

Обработчик событий должен быть зарегистрирован один раз для каждого соединения `PGconn`, события которого представляют интерес. Число обработчиков событий, которые можно зарегистрировать для соединения, не ограничивается ничем, кроме объёма памяти. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае.

Процедура, переданная в аргументе *proc*, будет вызываться, когда произойдёт событие libpq. Её адрес в памяти также применяется для поиска данных `instanceData`. Аргумент *name* используется при упоминании обработчика событий в сообщениях об ошибках. Это значение не может быть равно `NULL` или указывать на строку нулевой длины. Эта строка имени копируется в `PGconn`, так что переданная строка может быть временной. Сквозной указатель (*passThrough*) будет передаваться обработчику *proc* при каждом вызове события. Этот аргумент может равняться `NULL`.

`PQsetInstanceData`

Устанавливает для подключения *conn* указатель `instanceData` для обработчика *proc* равным *data*. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае. (Ошибка возможна, только если обработчик *proc* не был корректно зарегистрирован для соединения *conn*.)

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

`PQinstanceData`

Возвращает для соединения *conn* указатель на `instanceData`, связанный с обработчиком *proc*, либо `NULL`, если такого обработчика нет.

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

`PQresultSetInstanceData`

Устанавливает для объекта результата (*res*) указатель `instanceData` для обработчика *proc* равным *data*. Эта функция возвращает ненулевое значение в случае успеха или ноль в противном случае. (Ошибка возможна, только если обработчик *proc* не был корректно зарегистрирован для объекта результата.)

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

Имейте в виду, что память, представленная параметром *data*, не будет учитываться в `PQresultMemorySize`, если только она не была выделена функцией `PQresultAlloc`. (Этой функцией рекомендуется пользоваться, так как это избавляет от необходимости явно освобождать память после уничтожения результата.)

`PQresultInstanceData`

Возвращает для объекта результата (*res*) указатель на `instanceData`, связанный с обработчиком *proc*, либо `NULL`, если такого обработчика нет.

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

32.14.4. Пример обработки событий

Ниже показан схематичный пример управления внутренними данными, связанными с подключениями и результатами libpq.

```
/* required header for libpq events (note: includes libpq-fe.h) */
#include <libpq-events.h>

/* The instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn =
        PQconnectdb("dbname=postgres options=-csearch_path=");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        /* PQerrorMessage's result includes a trailing newline */
        fprintf(stderr, "%s", PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    /* called once on any connection that should receive events.
     * Sends a PGEVT_REGISTER to myEventProc.
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }

    /* conn instanceData is available */
    data = PQinstanceData(conn, myEventProc);

    /* Sends a PGEVT_RESULTCREATE to myEventProc */
    res = PQexec(conn, "SELECT 1 + 1");

    /* result instanceData is available */
    data = PQresultInstanceData(res, myEventProc);

    /* If PG_COPYRES_EVENTS is used, sends a PGEVT_RESULTCOPY to myEventProc */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);

    /* result instanceData is available if PG_COPYRES_EVENTS was
```

```
     * used during the PQcopyResult call.
     */
    data = PQresultInstanceData(res_copy, myEventProc);

    /* Both clears send a PGEVT_RESULTDESTROY to myEventProc */
    PQclear(res);
    PQclear(res_copy);

    /* Sends a PGEVT_CONNDESTROY to myEventProc */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* associate app specific data with connection */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
              memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* free instance data because the conn is being destroyed */
            if (data)
              free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            /* associate app specific data with result (copy it from conn) */
            PQresultSetInstanceData(e->result, myEventProc, res_data);
```

```
                break;
            }

            case PGEVT_RESULTCOPY:
            {
                PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
                mydata *src_data = PQresultInstanceData(e->src, myEventProc);
                mydata *dest_data = dup_mydata(src_data);

                /* associate app specific data with result (copy it from a result) */
                PQresultSetInstanceData(e->dest, myEventProc, dest_data);
                break;
            }

            case PGEVT_RESULTDESTROY:
            {
                PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
                mydata *data = PQresultInstanceData(e->result, myEventProc);

                /* free instance data because the result is being destroyed */
                if (data)
                  free_mydata(data);
                break;
            }

            /* unknown event ID, just return true. */
            default:
                break;
        }

        return true; /* event processing succeeded */
    }
```

32.15. Переменные окружения

Воспользовавшись следующими переменными окружения, можно задать значения параметров соединения по умолчанию, которые будут использоваться функциями `PQconnectdb`, `PQsetdbLogin` и `PQsetdb`, если никакое значение не будет задано вызывающим кодом. В частности, используя их, можно обойтись без жёсткого задания параметров соединения в простых клиентских приложениях.

- `PGHOST` действует так же, как параметр соединения host.
- `PGSSLNEGOTIATION` действует так же, как параметр соединения sslnegotiation.
- `PGHOSTADDR` действует так же, как параметр соединения hostaddr. Эту переменную можно задать вместо или вместе с `PGHOST` для предотвращения поиска адреса в DNS.
- `PGPORT` действует так же, как параметр соединения port.
- `PGDATABASE` действует так же, как параметр соединения dbname.
- `PGUSER` действует так же, как параметр соединения user.
- `PGPASSWORD` действует так же, как параметр соединения password. Использовать эту переменную окружения не рекомендуется по соображениям безопасности, так как в некоторых операционных системах непривилегированные пользователи могут видеть переменные окружения процессов в выводе ps; вместо этого лучше использовать файл паролей (см. Раздел 32.16).
- `PGPASSFILE` действует так же, как параметр соединения passfile.
- `PGREQUIREAUTH` действует так же, как параметр соединения require_auth.

- `PGCHANNELBINDING` действует так же, как параметр соединения channel_binding.
- `PGSERVICE` действует так же, как параметр соединения service.
- `PGSERVICEFILE` задаёт имя личного файла пользователя с параметрами подключения к службам (см. Раздел 32.17). По умолчанию применяется имя файла `~/.pg_service.conf` (или `%APPDATA%\postgresql\.pg_service.conf` в Microsoft Windows).
- `PGOPTIONS` действует так же, как параметр соединения options.
- `PGAPPNAME` действует так же, как параметр соединения application_name.
- `PGSSLMODE` действует так же, как параметр соединения sslmode.
- `PGREQUIRESSL` действует так же, как параметр соединения requiressl. Эта переменная окружения утратила актуальность с появлением переменной `PGSSLMODE`; если установить обе переменные, значение данной не возымеет эффекта.
- `PGSSLCOMPRESSION` действует так же, как параметр соединения sslcompression.
- `PGSSLCERT` действует так же, как параметр соединения sslcert.
- `PGSSLKEY` действует так же, как параметр соединения sslkey.
- `PGSSLCERTMODE` действует так же, как параметр соединения sslcertmode.
- `PGSSLROOTCERT` действует так же, как параметр соединения sslrootcert.
- `PGSSLCRL` действует так же, как параметр соединения sslcrl.
- `PGSSLCRLDIR` действует так же, как параметр соединения sslcrldir.
- `PGSSLSNI` действует так же, как параметр соединения sslsni.
- `PGREQUIREPEER` действует так же, как параметр соединения requirepeer.
- `PGSSLMINPROTOCOLVERSION` действует так же, как параметр соединения ssl_min_protocol_version.
- `PGSSLMAXPROTOCOLVERSION` действует так же, как параметр соединения ssl_max_protocol_version.
- `PGGSSENCMODE` действует так же, как параметр соединения gssencmode.
- `PGKRBSRVNAME` действует так же, как параметр соединения krbsrvname.
- `PGGSSLIB` действует так же, как параметр соединения gsslib.
- `PGGSSDELEGATION` действует так же, как параметр соединения gssdelegation.
- `PGCONNECT_TIMEOUT` действует так же, как параметр соединения connect_timeout.
- `PGCLIENTENCODING` действует так же, как параметр соединения client_encoding.
- `PGTARGETSESSIONATTRS` действует так же, как параметр соединения target_session_attrs.
- `PGLOADBALANCEHOSTS` действует так же, как параметр соединения load_balance_hosts.

Следующие переменные окружения позволяют задать поведение по умолчанию для каждого отдельного сеанса Postgres Pro. (См. также описание команд ALTER ROLE и ALTER DATABASE, позволяющих установить поведение по умолчанию для отдельного пользователя или отдельной базы.)

- `PGDATESTYLE` устанавливает стиль представления даты/времени по умолчанию. (Равносильно `SET datestyle TO ....`)
- `PGTZ` устанавливает часовой пояс по умолчанию. (Равносильно `SET timezone TO ....`)
- `PGGEQO` устанавливает режим по умолчанию для генетического оптимизатора запросов. (Равносильно `SET geqo TO ....`)

Информацию о корректных значениях этих переменных окружения можно найти в описании SQL-команды SET.

Следующие переменные среды определяют внутреннее поведение libpq; они переопределяют встроенные значения.

- `PGSYSCONFDIR` задаёт каталог, в котором содержится файл `pg_service.conf`, а в будущем он может содержать и другие общесистемные файлы конфигурации.
- `PGLOCALEDIR` задаёт каталог, содержащий файлы `locale`, предназначенные для перевода сообщений.

32.16. Файл паролей

Файл `.pgpass` в домашнем каталоге пользователя может содержать пароли, которые будут использоваться, если для подключения требуется пароль (и пароль не задаётся другим способом). В Microsoft Windows этот файл называется `%APPDATA%\postgresql\pgpass.conf` (где `%APPDATA%` обозначает каталог с данными приложений в профиле пользователя). Имя используемого файла паролей также можно задать в параметре подключения passfile или в переменной окружения `PGPASSFILE`.

Этот файл должен содержать строки следующего формата:

```
имя_узла:порт:база_данных:имя_пользователя:пароль
```

(Вы можете вставить в этот файл комментарий-памятку, скопировав показанную строку в него и добавив в начало `#`.) Первые четыре поля могут содержать строковые значения, либо знак `*`, соответствующий всему. Применяться будет пароль, указанный в первой из строк, значения полей в которой соответствуют текущему соединению. (Поэтому, если вы используете звёздочки, поместите более конкретные записи первыми.) Если запись должна содержать символ `:` или `\`, добавьте перед ним `\`. Поле с именем узла сопоставляется с параметром подключения `host` (если он указан) или с параметром `hostaddr` (если указан он); в случае отсутствия обоих параметров подразумевается имя `localhost`. Имя узла `localhost` также подразумевается, когда соединение устанавливается через Unix-сокет и параметр `host` соответствует установленному в libpq каталогу сокетов по умолчанию. На ведомом сервере имя базы данных `replication` соответствует подключениям к ведущему серверу, которые применяются для потоковой репликации. Поле *`база_данных`* имеет ограниченную ценность, так как пользователи используют один пароль для всех баз данных в кластере.

В системах Unix разрешения для файла паролей должны запрещать любой доступ к нему всем и группе; этого можно добиться командой `chmod 0600 ~/.pgpass`. Если разрешения будут менее строгими, этот файл будет игнорироваться. В Microsoft Windows предполагается, что файл хранится в безопасном месте, и никакие дополнительные проверки не производятся.

32.17. Файл соединений служб

Файл соединений служб позволяет связать параметры соединений libpq с одним именем службы. Затем это имя службы можно задать в строке подключения libpq и будут применены все связанные с ним параметры. Это позволяет модифицировать параметры соединений, обходясь без перекомпиляции приложения, использующего libpq. Имя службы можно также задать в переменной окружения `PGSERVICE`.

Имена служб могут быть определены либо в служебном файле каждого пользователя, либо в общесистемном файле. Если одно и то же имя службы существует как в пользовательском, так и в системном файле, пользовательский файл имеет приоритет. По умолчанию служебный файл пользователя называется `~/.pg_service.conf`. В Microsoft Windows этот файл называется `%APPDATA%\postgresql\pg_service.conf` (где `%APPDATA%` обозначает каталог данных приложений (Application Data) в профиле пользователя). Имя файла также можно задать в переменной окружения `PGSERVICEFILE`. Общесистемный файл называется `pg_service.conf`. По умолчанию он ищется в каталоге `etc` установки Postgres Pro (используйте `pg_config --sysconfdir`, чтобы точно установить этот каталог). Можно указать другой каталог, но не другое имя файла, задав переменную окружения `PGSYSCONFDIR`.

В любом служебном файле используется формат «INI-файлов», в котором имя раздела задаёт имя службы, а параметры внутри — параметры соединения; их список приведён в Подразделе 32.1.2. Например:

```
# комментарий
```

```
[mydb]
host=somehost
port=5433
user=admin
```

Пример такого файла можно найти в каталоге установки Postgres Pro в `share/pg_service.conf.sample`.

Параметры подключения, полученные из служебного файла, объединяются с параметрами, полученными из других источников. Значение в этом файле переопределяет соответствующую переменную окружения и, в свою очередь, может быть изменён значением, указанным непосредственно в строке подключения. Например, используя указанный выше служебный файл, строка подключения `service=mydb port=5434` будет использовать сервер `somehost`, порт `5434`, пользователя `admin` и другие параметры, установленные переменными окружения или встроенными значениями по умолчанию.

32.18. Получение параметров соединения через LDAP

Если библиотека libpq была собрана с поддержка LDAP (`configure` передавался ключ `--with-ldap`), такие параметры соединения, как `host` и `dbname`, можно получить через LDAP с центрального сервера. Преимущество такого подхода в том, что при изменении параметров подключения к базе данных свойства соединения не придётся изменять на всех клиентских компьютерах.

Для получения параметров соединений через LDAP используется файл соединений служб `pg_service.conf` (см. Раздел 32.17). Строка в `pg_service.conf`, начинающаяся с указания протокола `ldap://`, будет воспринята как URL в LDAP и выполнится как запрос к LDAP. Результатом запроса должен быть список пар `keyword = value`, которые и будут задавать параметры соединений. Заданный URL должен соответствовать *RFC 1959* и иметь следующий вид:

```
ldap://[имя_сервера[:порт]]/база_поиска?атрибут?область_поиска?фильтр
```

; по умолчанию *`имя_сервера`* — `localhost`, а *`порт`* — 389.

Обработка `pg_service.conf` прекращается после удачного поиска в LDAP, но если с сервером LDAP связаться не удаётся, обрабатываются следующие строки этого файла. Так сделано для того, чтобы можно было реализовать запасные варианты, добавив дополнительные строки с URL LDAP, указывающими на другие серверы LDAP, или классические пары `keyword = value`, либо используя параметры соединений по умолчанию. Если же вы хотите получить ошибку в этой ситуации, добавьте после строки с URL-адресом LDAP синтаксически некорректную строку.

Простую запись LDAP, созданную из такого файла LDIF

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:device
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

можно запросить из каталога LDAP, указав следующий URL:

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?(cn=mydatabase)
```

Также возможно сочетать обычные записи в файле служб с поиском в LDAP. Полный пример описания службы в `pg_service.conf` может быть таким:

```
# в LDAP хранится только имя и порт сервера; имя базы и пользователя нужно задать явно
[customerdb]
```

```
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

32.19. Поддержка SSL

Postgres Pro реализует собственную поддержку SSL-подключений, используя протоколы TLS, для шифрования клиент-серверного взаимодействия в качестве меры безопасности. Подробнее функциональность SSL на стороне сервера описывается в Разделе 17.9.

Библиотека libpq читает системный файл конфигурации OpenSSL. По умолчанию этот файл называется `openssl.cnf` и находится в каталоге, который сообщает команда `openssl version -d`. Если требуется указать другое расположение файла конфигурации, его можно задать в переменной окружения `OPENSSL_CONF`.

32.19.1. Проверка сертификатов сервера на стороне клиента

По умолчанию Postgres Pro не выполняет никакие проверки сертификата сервера. Это означает, что клиента можно ввести в заблуждение, подменив сервер (например, изменив запись в DNS или заняв его IP-адрес). Чтобы защититься от такой подмены, клиент должен иметь возможность проверять подлинность сервера по цепочке доверия. Для создания цепочки доверия нужно установить корневой (самоподписанный) сертификат центра сертификации (ЦС) на одном компьютере, а конечный сертификат, *подписанный* корневым, на другом. В цепочке может использоваться и «промежуточный» сертификат, который подписывается корневым сертификатом и подписывает подчинённые сертификаты.

Чтобы клиент мог проверить подлинность сервера, установите на клиенте корневой сертификат, а на сервере установите конечный сертификат, подписанный этим корневым. Чтобы сервер мог проверить подлинность клиента, установите на сервере корневой сертификат, а на клиенте конечный сертификат, подписанный данным корневым. Конечный сертификат также может связываться с корневым через один или несколько промежуточных сертификатов (они обычно хранятся вместе с конечным сертификатом).

Когда цепочка доверия присутствует, клиент может проверить конечный сертификат, переданный сервером, двумя способами. Если параметр `sslmode` имеет значение `verify-ca`, libpq будет проверять подлинность сервера, проверяя цепочку доверия до корневого сертификата, установленного на клиенте. Если в `sslmode` выбран режим `verify-full`, libpq будет *также* проверять соответствие имени узла сервера имени, записанному в сертификате. SSL-подключение не будет установлено, если проверить сертификат сервера не удастся. Режим `verify-full` рекомендуется для большинства окружений, где важна безопасность.

В режиме `verify-full` имя узла сверяется с атрибутом Subject Alternative Name (Альтернативное имя субъекта, SAN) в сертификате или с атрибутом Common Name (Общее имя), если в сертификате отсутствует атрибут SAN типа `dNSName`. Если атрибут имени сертификата начинается со звёздочки (`*`), звёздочка воспринимается как подстановочный знак и ей будут соответствовать все символы, *кроме* точки (`.`). Это означает, что такой сертификат не будет соответствовать поддоменам. Если подключение устанавливается по IP-адресу, а не по имени узла, IP-адрес будет сопоставляться с записями SAN типа `iPAddress` или `dNSName` (без поиска в DNS). Если подходящих записей `iPAddress` и `dNSName` не найдётся, IP-адрес будет сопоставлен с атрибутом Common Name.

Примечание

Для обратной совместимости с более ранними версиями Postgres Pro IP-адрес узла проверяется не совсем так, как требует *RFC 6125*. IP-адрес узла всегда сопоставляется с записями `dNSName`, а также с `iPAddress` в SAN и может сопоставляться с атрибутом Common Name, если соответствующие записи SAN не существуют.

Чтобы настроить проверку сертификата сервера, необходимо поместить один или несколько корневых сертификатов в файл `~/.postgresql/root.crt` в домашнем каталоге пользователя. (В Microsoft

Windows этот файл называется `%APPDATA%\postgresql\root.crt`.) Также следует добавлять в этот файл промежуточные сертификаты, если они нужны для связывания цепочки сертификатов, переданных сервером, с корневыми сертификатами, установленными на клиенте.

Если существует файл `~/.postgresql/root.crl` (или `%APPDATA%\postgresql\root.crl` в Microsoft Windows), при проверке также учитывается содержащийся в нём список отозванных сертификатов (CRL, Certificate Revocation List).

Размещение файла корневых сертификатов и CRL можно поменять, задав параметры соединения `sslrootcert` и `sslcrl` или переменные окружения `PGSSLROOTCERT` и `PGSSLCRL`, соответственно. Также вы можете указать расположение каталога с файлами CRL в параметре `sslcrldir` или в переменной окружения `PGSSLCRLDIR`.

> **Примечание**
>
> Для обратной совместимости с предыдущими версиями Postgres Pro, при наличии файла с сертификатами корневых ЦС поведение режима `sslmode=require` не отличается от режима `verify-ca`, то есть сертификат сервера будет проверяться по сертификату ЦС. Полагаться на это поведение не рекомендуется — приложения, которым нужно проверять сертификат, должны всегда выбирать режим `verify-ca` или `verify-full`.

32.19.2. Клиентские сертификаты

Если сервер попытается проверить подлинность клиента, запрашивая конечный сертификат клиента, libpq будет передавать сертификаты, сохранённые в файле `~/.postgresql/postgresql.crt` в домашнем каталоге пользователя. Эти сертификаты должны связываться по цепочке с корневым сертификатом, которому доверяет сервер. Также должен присутствовать соответствующий закрытый ключ `~/.postgresql/postgresql.key`. В Microsoft Windows эти файлы называются `%APPDATA%\postgresql\postgresql.crt` и `%APPDATA%\postgresql\postgresql.key`. Размещение файлов сертификатов и закрытого ключа можно переопределить с помощью параметров подключения `sslcert` и `sslkey` либо переменных окружения `PGSSLCERT` и `PGSSLKEY`.

В системах Unix разрешения для файла закрытого ключа должны запрещать любой доступ к нему всем и группе; этого можно добиться командой `chmod 0600 ~/.postgresql/postgresql.key`. Возможен и другой вариант, когда этим файлом владеет root, а группа имеет доступ на чтение (то есть, маска разрешений `0640`). Данный вариант полезен, когда файлами сертификатов и ключей управляет сама операционная система. В этом случае пользователь, запускающий libpq, должен быть членом группы, имеющей доступ к указанным файлам сертификата и ключа. (В Microsoft Windows каталог `%APPDATA%\postgresql` считается защищённым, поэтому разрешения для файлов не проверяются.)

Первым сертификатом в `postgresql.crt` должен быть сертификат клиента, так как он должен соответствовать закрытому ключу клиента. Дополнительно в этот файл могут быть добавлены «промежуточные» сертификаты — это позволит избежать хранения всех промежуточных сертификатов на сервере (см. ssl_ca_file).

Сертификат и ключ могут задаваться в формате PEM или ASN.1 DER.

Ключ может быть сохранён в открытом виде или зашифрован любым поддерживаемым OpenSSL алгоритмом, например AES-128. Если ключ хранится зашифрованным, пароль для его расшифровывания может быть задан в параметре подключения sslpassword. Если же ключ зашифрован, а параметр `sslpassword` не задан или имеет пустое значение, пароль будет запрошен интерактивно средствами OpenSSL в приглашении `Enter PEM pass phrase:` (Введите пароль для PEM:) при условии наличия TTY. Приложения могут переопределить приглашение для запроса пароля сертификата и обработку параметра `sslpassword`, установив собственную функцию-обработчик пароля ключа; см. `PQsetSSLKeyPassHook_OpenSSL`.

За инструкциями по созданию сертификатов обратитесь к Подразделу 17.9.5.

32.19.3. Защита, обеспечиваемая в различных режимах

Разные значения параметра `sslmode` обеспечивают разные уровни защиты. SSL позволяет защититься от следующих типов атак:

Прослушивание

Если третья сторона может прослушивать сетевой трафик между клиентом и сервером, она может получить как информацию соединения (включая имя пользователя и пароль), так и передаваемые данные. Чтобы защититься от этого, SSL шифрует трафик.

Посредник (MITM)

Если третья сторона может модифицировать данные, передаваемые между клиентом и сервером, она может представиться сервером и, таким образом, сможет видеть и модифицировать данные, *даже если они зашифрованы*. Третья сторона затем может воспроизводить характеристики соединения и данные для подлинного сервера, что сделает невозможным обнаружение этой атаки. Векторами такой атаки может быть «отравление» DNS и подмена адресов, в результате чего клиент будет обращаться не к тому серверу, к которому нужно. Также есть несколько других вариантов реализации этой атаки. Для защиты в SSL применяется проверка сертификатов, в результате которой сервер доказывает свою подлинность клиенту.

Олицетворение

Если третья сторона может представляться авторизованным клиентом, она может просто обращаться к данным, к которым не должна иметь доступа. Обычно это происходит вследствие небезопасного управления паролями. В SSL для предотвращения этой угрозы используются клиентские сертификаты, гарантирующие, что к серверу могут обращаться только владельцы действительных сертификатов.

Чтобы соединение было гарантированно защищено SSL, механизм SSL должен быть настроен *на клиенте и на сервере*, прежде чем будет установлено соединение. Если он настроен только на сервере, клиент может начать передавать важную информацию (например, пароли), до того как поймёт, что сервер требует высокого уровня безопасности. В libpq для установления безопасных соединений нужно задать для параметра `sslmode` значение `verify-full` или `verify-ca` и предоставить системе корневой сертификат для проверки. В качестве аналогии можно привести использование адреса с `https` для безопасного просмотра веб-содержимого.

Когда подлинность сервера подтверждена, клиент может передавать конфиденциальные данные. Это значит, что до этого момента клиенту не нужно знать, применяются ли сертификаты для аутентификации, так что настройка использования сертификатов только на стороне сервера не угрожает безопасности.

Все варианты использования SSL подразумевают издержки шифрования и обмена ключами, что порождает необходимость выбора между производительностью и безопасностью. В Таблице 32.1 описываются риски, от которых защищают различные варианты `sslmode`, и приводятся утверждения относительно защиты и издержек.

Таблица 32.1. Описания режимов SSL

`sslmode`	**Защита от прослушивания**	**Защита от MITM**	**Утверждение**
`disable`	Нет	Нет	Мне не важна безопасность и я не приемлю издержки, связанные с шифрованием.
`allow`	Возможно	Нет	Мне не важна безопасность, но я приемлю издержки, связанные с шифрованием, если на этом настаивает сервер.
`prefer`	Возможно	Нет	Мне не важна безопасность, но я предпочитаю шифрование (и прием-

`sslmode`	**Защита от прослушивания**	**Защита от MITM**	**Утверждение**
			лю связанные издержки), если это поддерживает сервер.
`require`	Да	Нет	Я хочу, чтобы мои данные шифровались, и я приемлю сопутствующие издержки. Я доверяю сети в том, что она обеспечивает подключение к нужному серверу.
`verify-ca`	Да	Зависит от политики ЦС	Я хочу, чтобы мои данные шифровались, и я приемлю сопутствующие издержки. Мне нужна уверенность в том, что я подключаюсь к доверенному серверу.
`verify-full`	Да	Да	Я хочу, чтобы мои данные шифровались, и я приемлю сопутствующие издержки. Мне нужна уверенность в том, что я подключаюсь к доверенному серверу и это именно указанный мной сервер.

Различие вариантов `verify-ca` и `verify-full` зависит от характера корневого ЦС. Если используется публичный ЦС, режим `verify-ca` допускает подключение к серверу с сертификатом, который получил *кто угодно* в этом ЦС. В такой ситуации нужно всегда использовать режим `verify-full`. Если же используется локальный ЦС или даже самоподписанный сертификат, режим `verify-ca` обычно обеспечивает достаточную защиту.

По умолчанию параметр `sslmode` имеет значение `prefer`. Как показано в таблице, оно неэффективно с точки зрения безопасности и может только привносить дополнительные издержки. Оно выбрано по умолчанию исключительно для обратной совместимости и не рекомендуется для защищённых окружений.

32.19.4. Файлы, используемые клиентом SSL

В Таблице 32.2 перечислены файлы, имеющие отношение к настройке SSL на стороне клиента.

Таблица 32.2. Файлы, используемые клиентом SSL/libpq

Файл	**Содержимое**	**Назначение**
`~/.postgresql/postgresql.crt`	сертификат клиента	передаётся серверу
`~/.postgresql/postgresql.key`	закрытый ключ клиента	подтверждает клиентский сертификат, передаваемый владельцем; не гарантирует, что владелец сертификата заслуживает доверия
`~/.postgresql/root.crt`	сертификаты доверенных ЦС	позволяет проверить, что сертификат сервера подписан доверенным центром сертификации
`~/.postgresql/root.crl`	сертификаты, отозванные центрами сертификации	сертификат сервера должен отсутствовать в этом списке

32.19.5. Инициализация библиотеки SSL

Если ваше приложение инициализирует библиотеку `libssl` и/или `libcrypto`, и libpq собрана с поддержкой SSL, вы должны вызвать `PQinitOpenSSL`, чтобы сообщить libpq, что библиотека `libssl` и/или `libcrypto` уже инициализированы вашим приложением, чтобы libpq не пыталась ещё раз инициализировать их. Однако в этом нет необходимости при использовании OpenSSL версии 1.1.0 или выше, в которой решена проблема повторных инициализаций.

`PQinitOpenSSL`

Позволяет приложениям выбрать, какие библиотеки безопасности нужно инициализировать.

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

Когда параметр *`do_ssl`* отличен от нуля, libpq будет инициализировать библиотеку OpenSSL перед первым подключением к базе данных. Когда параметр *`do_crypto`* не равен нулю, будет инициализироваться библиотека `libcrypto`. По умолчанию (если функция `PQinitOpenSSL` не вызывается) инициализируются обе библиотеки. Если поддержка SSL не была скомпилирована, эта функция присутствует, но ничего не делает.

Если ваше приложение использует и инициализирует библиотеку OpenSSL или её нижележащую библиотеку `libcrypto`, вы *должны* вызвать эту функцию, передав нули в соответствующих параметрах, перед первым подключением к базе данных. Собственно инициализацию также важно произвести перед установлением подключения.

`PQinitSSL`

Позволяет приложениям выбрать, какие библиотеки безопасности нужно инициализировать.

```
void PQinitSSL(int do_ssl);
```

Эта функция равнозначна вызову `PQinitOpenSSL(do_ssl, do_ssl)`. Приложениям достаточно инициализировать или не инициализировать обе библиотеки OpenSSL и `libcrypto` одновременно.

Функция `PQinitSSL` существует со времён PostgreSQL 8.0, тогда как `PQinitOpenSSL` появилась в PostgreSQL 8.4, так что `PQinitSSL` может быть предпочтительней для приложений, которым нужно работать с более старыми версиями libpq.

32.20. Поведение в многопоточных программах

Начиная с 17-й версии, библиотека libpq всегда поддерживает повторные вызовы и многопоточность. Однако эта реализация не лишена ограничений: два потока не должны пытаться одновременно работать с одним объектом `PGconn`. В частности, не допускается параллельное выполнение команд из разных потоков через один объект соединения. (Если вам нужно выполнять команды одновременно, используйте несколько соединений.)

Объекты `PGresult` после создания обычно доступны только для чтения, и поэтому их можно свободно передавать между потоками. Однако если вы используете какую-либо из функций, изменяющих `PGresult`, описанных в Разделе 32.12 или Разделе 32.14, вы должны также избегать одновременных обращений к одному объекту `PGresult`.

В ранних версиях библиотеку libpq можно было собрать с поддержкой многопоточности или без неё, в зависимости от параметров компилятора. Эта функция позволяет узнать, поддерживает ли libpq многопоточность:

`PQisthreadsafe`

Возвращает состояние потокобезопасности в библиотеке libpq.

```
int PQisthreadsafe();
```

Возвращает 1, если библиотека libpq потокобезопасная, или 0 в противном случае. Начиная с версии 17, всегда возвращает 1.

Устаревшие функции `PQrequestCancel` и `PQoidStatus` не являются потокобезопасными и не должны применяться в многопоточных программах. Вместо `PQrequestCancel` можно использовать `PQcancelBlocking`, а вместо `PQoidStatus` — `PQoidValue`.

Если протокол Kerberos применяется в пользовательском приложении (помимо возможного использования внутри libpq), необходимо обеспечить блокировку вокруг вызовов Kerberos, так как его функции не являются потокобезопасными.

32.21. Сборка программ с libpq

Чтобы собрать (то есть, скомпилировать и скомпоновать) программу, использующую libpq, вы должны проделать следующие действия:

- Включите заголовочный файл `libpq-fe.h`:

  ```
  #include <libpq-fe.h>
  ```

  Если вы не сделаете этого, обычно вас ждут примерно такие сообщения об ошибках от компилятора:

  ```
  foo.c: In function `main':
  foo.c:34: `PGconn' undeclared (first use in this function)
  foo.c:35: `PGresult' undeclared (first use in this function)
  foo.c:54: `CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- Сообщите вашему компилятору каталог, в котором установлены заголовочные файлы Postgres Pro, передав ему параметр `-I`*`каталог`*. (В некоторых случаях компилятор сам может обращаться к нужному каталогу, так что этот параметр можно опустить.) Например, ваша команда компиляции может быть такой:

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  Если вы используете скрипты сборки Makefile, добавьте этот параметр в переменную `CPPFLAGS`:

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  Если существует возможность, что вашу программу будут компилировать другие пользователи, то путь к каталогу не следует жёстко задавать таким образом. Вместо этого вы можете воспользоваться утилитой `pg_config`и узнать, где в локальной системе находятся заголовочные файлы, следующим образом:

  ```
  $ pg_config --includedir
  /usr/local/include
  ```

  Если у вас установлена программа `pkg-config`, вместо этого вы можете выполнить:

  ```
  $ pkg-config --cflags libpq
  -I/usr/local/include
  ```

  Заметьте, что при этом перед путём сразу будет добавлен ключ `-I`.

  Если требуемый параметр не будет передан компилятору, вы получите примерно такое сообщение об ошибке:

  ```
  testlibpq.c:8:22: libpq-fe.h: No such file or directory
  ```

- При компоновке окончательной программы добавьте параметр `-lpq`, чтобы была подключена библиотека libpq, а также параметр `-L`*`каталог`*, указывающий на каталог, в котором находится libpq. (Опять же, компилятор будет просматривать определённые каталоги по умолчанию.) Для максимальной переносимости указывайте ключ `-L` перед параметром `-lpq`. Например:

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  Каталог с библиотекой можно узнать, так же используя `pg_config`:

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

Или с помощью той же программы `pkg-config`:

```
$ pkg-config --libs libpq
-L/usr/local/pgsql/lib -lpq
```

Заметьте, что и в этом случае выводится полностью сформированный параметр, а не только путь.

В случае проблем в этой области возможны примерно такие сообщения об ошибках:

```
testlibpq.o: In function `main':
testlibpq.o(.text+0x60): undefined reference to `PQsetdbLogin'
testlibpq.o(.text+0x71): undefined reference to `PQstatus'
testlibpq.o(.text+0xa4): undefined reference to `PQerrorMessage'
```

Они означают, что вы забыли добавить параметр `-lpq`.

```
/usr/bin/ld: cannot find -lpq
```

Такая ошибка означает, что вы забыли добавить ключ `-L` или не указали правильный каталог.

32.22. Примеры программ

Эти и другие примеры можно найти в каталоге `src/test/examples` в дистрибутиве исходного кода PostgreSQL.

Пример 32.1. Первая программа, демонстрирующая использование libpq

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *
 *      Test the C version of libpq, the PostgreSQL frontend library.
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
```

```
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* Set always-secure search path, so malicious users can't take control. */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * Should PQclear PGresult whenever it is no longer needed to avoid memory
     * leaks
     */
    PQclear(res);

    /*
     * Our test case here involves using a cursor, for which we must be inside
     * a transaction block.  We could do the whole thing with a single
     * PQexec() of "select * from pg_database", but that's too trivial to make
     * a good example.
     */

    /* Start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    /*
     * Fetch rows from pg_database, the system catalog of databases
     */
    res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
        PQclear(res);
```

```
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in myportal");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the rows */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* close the portal ... we don't bother to check for errors ... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Пример 32.2. Вторая программа, демонстрирующая использование libpq

```
/*
 * src/test/examples/testlibpq2.c
 *
 *
 * testlibpq2.c
 *      Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 * Repeat four times to get this program to exit.
 *
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
```

```
 * (provided in src/test/examples/testlibpq2.sql):
 *
 *   CREATE SCHEMA TESTLIBPQ2;
 *   SET search_path = TESTLIBPQ2;
 *   CREATE TABLE TBL1 (i int4);
 *   CREATE TABLE TBL2 (i int4);
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * Start this program, then from psql do this four times:
 *
 *   INSERT INTO TESTLIBPQ2.TBL1 VALUES (10);
 */

#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>

#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
```

```
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* Set always-secure search path, so malicious users can't take control. */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * Should PQclear PGresult whenever it is no longer needed to avoid memory
     * leaks
     */
    PQclear(res);

    /*
     * Issue LISTEN command to enable notifications from the rule's NOTIFY.
     */
    res = PQexec(conn, "LISTEN TBL2");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    /* Quit after four notifies are received. */
    nnotifies = 0;
    while (nnotifies < 4)
    {
        /*
         * Sleep until something happens on the connection.  We use select(2)
         * to wait for input, but you could also use poll() or similar
         * facilities.
         */
        int         sock;
        fd_set      input_mask;

        sock = PQsocket(conn);

        if (sock < 0)
            break;              /* shouldn't happen */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }
```

```
        /* Now check for input */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                    notify->relname, notify->be_pid);
            PQfreemem(notify);
            nnotifies++;
            PQconsumeInput(conn);
        }
    }

    fprintf(stderr, "Done.\n");

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Пример 32.3. Третья программа, демонстрирующая использование libpq

```
/*
 * src/test/examples/testlibpq3.c
 *
 *
 * testlibpq3.c
 *      Test out-of-line parameters and binary I/O.
 *
 * Before running this, populate a database with the following commands
 * (provided in src/test/examples/testlibpq3.sql):
 *
 * CREATE SCHEMA testlibpq3;
 * SET search_path = testlibpq3;
 * SET standard_conforming_strings = ON;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 * INSERT INTO test1 values (1, 'joe''s place', '\000\001\002\003\004');
 * INSERT INTO test1 values (2, 'ho there', '\004\003\002\001\000');
 *
 * The expected output is:
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */

#ifdef WIN32
#include <windows.h>
#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * This function prints a query result that is a binary-format fetch from
 * a table defined as in the comment above.  We split it out because the
 * main() function uses it twice.
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /* Use PQfnumber to avoid assumptions about field order in result */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;

        /* Get the field values (we ignore possibility they are null!) */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * The binary representation of INT4 is in network byte order, which
         * we'd better coerce to the local byte order.
         */
        ival = ntohl(*((uint32_t *) iptr));
```

```
        /*
         * The binary representation of TEXT is, well, text, and since libpq
         * was nice enough to append a zero byte to it, it'll work just fine
         * as a C string.
         *
         * The binary representation of BYTEA is a bunch of bytes, which could
         * include embedded nulls so we have to pay attention to field length.
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
               PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* Set always-secure search path, so malicious users can't take control. */
    res = PQexec(conn, "SET search_path = testlibpq3");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
```

```
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    /*
     * The point of this program is to illustrate use of PQexecParams() with
     * out-of-line parameters, as well as binary transmission of data.
     *
     * This first example transmits the parameters as text, but receives the
     * results in binary format.  By using out-of-line parameters we can avoid
     * a lot of tedious mucking about with quoting and escaping, even though
     * the data is text.  Notice how we don't have to do anything special with
     * the quote mark in the parameter value.
     */

    /* Here is our out-of-line parameter value */
    paramValues[0] = "joe's place";

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE t = $1",
                       1,       /* one param */
                       NULL,    /* let the backend deduce param type */
                       paramValues,
                       NULL,    /* don't need param lengths since text */
                       NULL,    /* default to all text params */
                       1);      /* ask for binary results */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /*
     * In this second example we transmit an integer parameter in binary form,
     * and again retrieve the results in binary form.
     *
     * Although we tell PQexecParams we are letting the backend deduce
     * parameter type, we really force the decision by casting the parameter
     * symbol in the query text.  This is a good safety measure when sending
     * binary parameters.
     */

    /* Convert integer value "2" to network byte order */
    binaryIntVal = htonl((uint32_t) 2);

    /* Set up parameter arrays for PQexecParams */
    paramValues[0] = (char *) &binaryIntVal;
    paramLengths[0] = sizeof(binaryIntVal);
    paramFormats[0] = 1;        /* binary */

    res = PQexecParams(conn,
```

```
                       "SELECT * FROM test1 WHERE i = $1::int4",
                       1,       /* one param */
                       NULL,    /* let the backend deduce param type */
                       paramValues,
                       paramLengths,
                       paramFormats,
                       1);      /* ask for binary results */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Глава 33. Большие объекты

В Postgres Pro имеется механизм для работы с *большими объектами*, предоставляющий доступ в потоковом режиме к пользовательским данным, сохранённым в специальной структуре больших объектов. Потоковый доступ удобен, когда нужно обрабатывать данные, объём которых слишком велик, чтобы оперировать ими как единым целым.

В этой главе описывается реализация, а также программный интерфейс и функции языка запросов для работы с данными больших объектов Postgres Pro. В примерах в этой главе будет использоваться библиотека libpq для языка C, но та же функциональность поддерживается и другими программными интерфейсами Postgres Pro. Другие интерфейсы могут использовать внутри себя интерфейс больших объектов для реализации общего подхода к работе с большими значениями. Здесь это не описывается.

33.1. Введение

Все большие объекты хранятся в одной системной таблице с именем `pg_largeobject`. Для каждого большого объекта также имеется запись в системной таблице `pg_largeobject_metadata`. Большие объекты можно создавать, изменять и удалять, используя API чтения/записи, подобный стандартному API для работы с файлами.

Postgres Pro также поддерживает систему хранения, названную «TOAST», которая автоматически переносит значения, не умещающиеся в одну страницу таблицы, в дополнительную область хранилища. Вследствие этого подсистема больших объектов отчасти оказывается устаревшей. Однако её преимуществом остаётся то, что она позволяет сохранять значения размером до 4 Тбайт, тогда как поля в TOAST ограничиваются 1 Гбайтом. Кроме того, чтение и изменение больших объектов можно выполнять эффективнее по сравнению с полями TOAST, которые при большинстве операций считываются и записываются как единое целое.

33.2. Особенности реализации

Механизм больших объектов разбивает большие объекты на «фрагменты» и сохраняет эти фрагменты в строках таблицы. При произвольном доступе на запись и чтение быстрый поиск нужного фрагмента обеспечивается индексом-B-деревом в этой таблице.

Фрагменты больших объектов не должны быть последовательными. Например, если приложение откроет новый большой объект, переместится к смещению 1000000 байт и запишет несколько байт, это не приведёт к выделению лишнего 1000000 байт в хранилище; записаны будут только фрагменты, покрывающие диапазон собственно записанных байт. Операция чтения, однако, прочитает нули для всех неразмещённых в хранилище байт, предшествующих последнему записанному фрагменту. Это соответствует принятому поведению «разреженных» файлов в файловых системах Unix.

Начиная с PostgreSQL 9.0, для больших объектов назначается владелец и набор прав доступа, которыми можно управлять командами GRANT и REVOKE. Для чтения большого объекта требуются права `SELECT`, а для записи или усечения его — права `UPDATE`. Удалять большой объект, задавать комментарий для него, либо сменять его владельца разрешается только его владельцу (или суперпользователю базы данных). Для совместимости с предыдущими версиями можно скорректировать это поведение, изменив параметр времени выполнения lo_compat_privileges.

33.3. Клиентские интерфейсы

В этом разделе описываются средства, которые предоставляет клиентская библиотека Postgres Pro libpq для обращения к большим объектам. Интерфейс работы с большими объектами Postgres Pro создан по подобию интерфейса файловых систем Unix, так что он включает аналоги функций `open`, `read`, `write`, `lseek` и т. д.

Все операции с большими объектами с применением этих функций *должны* иметь место в блоке транзакции SQL, так как дескрипторы больших объектов актуальны только во время транзакции.

Операции записи, в том числе `lo_open` в режиме `INV_WRITE`, не допускаются в читающих транзакциях.

Если при выполнении одной из этих функций происходит ошибка, эта функция возвращает значение, иначе невозможное, обычно 0 или -1. Сообщение, описывающее ошибку, сохраняется в объекте соединения; получить его можно с помощью `PQerrorMessage` .

Клиентские приложения, которые используют эти функции, должны включать заголовочный файл `libpq/libpq-fs.h` и компоноваться с библиотекой libpq.

Клиентские приложения не могут использовать эти функции, когда соединение libpq находится в конвейерном режиме.

33.3.1. Создание большого объекта

Функция

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

создаёт новый большой объект. В *`lobjId`* можно задать назначаемый ему OID; при этом произойдёт ошибка, если этот OID уже присвоен какому-либо большому объекту. Если в *`lobjId`* передаётся `InvalidOid` (ноль), `lo_create` присваивает большому объекту свободный OID. Возвращаемым значением будет OID, назначенный новому большому объекту, либо `InvalidOid` (ноль) в случае ошибки.

Пример:

```
inv_oid = lo_create(conn, desired_oid);
```

Старая функция

```
Oid lo_creat(PGconn *conn, int mode);
```

также создаёт новый большой объект, всегда присваивая ему неиспользуемый OID. Возвращаемое значение — это OID, присвоенный новому большому объекту, или `InvalidOid` (ноль) в случае ошибки.

В PostgreSQL версии 8.1 и выше *`режим`* игнорируется, так что вызов `lo_creat` полностью равнозначен вызову `lo_create` с нулевым вторым аргументом. Однако едва ли стоит использовать `lo_creat`, если вам не нужно работать с серверами версии ниже 8.1. Чтобы работать с таким старым сервером, вы должны использовать `lo_creat`, а не `lo_create`, и передать в аргументе *`режим`* значение `INV_READ`, `INV_WRITE` или `INV_READ | INV_WRITE`. (Эти символические константы определены в заголовочном файле `libpq/libpq-fs.h`.)

Пример:

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

33.3.2. Импорт большого объекта

Чтобы импортировать в качестве большого объекта файл операционной системы, вызовите

```
Oid lo_import(PGconn *conn, const char *filename);
```

В *`filename`* задаётся имя файла в операционной системе, который будет импортирован как большой объект. Возвращаемым значением будет OID, назначенный новому большому объекту, либо `InvalidOid` (ноль) в случае ошибки. Заметьте, что этот файл читает библиотека клиентского интерфейса, а не сервер; таким образом, он должен существовать в файловой системе на стороне клиента и быть доступным для чтения клиентскому приложению.

Функция

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

также импортирует новый большой объект. В *`lobjId`* можно задать назначаемый ему OID; при этом произойдёт ошибка, если этот OID уже присвоен какому-либо большому объекту. Если в *`lobjId`* передаётся `InvalidOid` (ноль), `lo_import_with_oid` присваивает большому объекту свободный OID (так же, как и `lo_import`). Возвращаемым значением будет OID, назначенный новому большому объекту, либо `InvalidOid` (ноль) в случае ошибки.

Функция `lo_import_with_oid` появилась в PostgreSQL 8.4 и вызывает внутри `lo_create`, появившуюся в 8.1; если попытаться выполнить её с сервером версии 8.0 или ранней, она завершится ошибкой и возвратит `InvalidOid`.

33.3.3. Экспорт большого объекта

Чтобы экспортировать большой объект в файл операционной системы, вызовите

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

В аргументе *lobjId* задаётся OID экспортируемого большого объекта, а в аргументе *filename* задаётся имя файла в операционной системе. Заметьте, что файл записывается библиотекой клиентского интерфейса, а не сервером. Возвращает 1 при успешном выполнении, -1 при ошибке.

33.3.4. Открытие существующего большого объекта

Чтобы открыть существующий большой объект для чтения или записи, вызовите

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

В аргументе *lobjId* задаётся OID открываемого большого объекта. Биты в аргументе *mode* определяют, открывается ли файл для чтения (`INV_READ`), для записи (`INV_WRITE`), либо для чтения/записи. (Эти константы определяются в заголовочном файле `libpq/libpq-fs.h`.) Функция `lo_open` возвращает дескриптор большого объекта (неотрицательный) для последующего использования в функциях `lo_read`, `lo_write`, `lo_lseek`, `lo_lseek64`, `lo_tell`, `lo_tell64`, `lo_truncate`, `lo_truncate64` и `lo_close`. Этот дескриптор актуален только до завершения текущей транзакции. В случае ошибки возвращается -1.

В настоящее время сервер не различает режимы `INV_WRITE` и `INV_READ | INV_WRITE`: с таким дескриптором можно читать данные в любом случае. Однако есть значительное отличие этих режимов от одиночного `INV_READ`: с дескриптором `INV_READ` записывать данные нельзя, а данные, считываемые через него, будут отражать содержимое большого объекта в снимке транзакции, который был активен при выполнении `lo_open`, то есть не будут включать изменения, произведённые позже этой или другими транзакциями. При чтении с дескриптором `INV_WRITE` возвращаются данные, отражающие все изменения, произведённые другими зафиксированными транзакциями, а также текущей транзакцией. Это подобно различиям режимов `REPEATABLE READ` и `READ COMMITTED` для обычных команд SQL `SELECT`.

Функция `lo_open` завершится ошибкой, если пользователь не имеет права `SELECT` для данного большого объекта или если указан флаг `INV_WRITE` и отсутствует право `UPDATE`. (До PostgreSQL 11 права проверялись при первом фактическом вызове функции чтения или записи с этим дескриптором.) Отключить новые проверки можно с помощью параметра времени выполнения lo_compat_privileges.

Пример:

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

33.3.5. Запись данных в большой объект

Функция

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

записывает *len* байт из буфера *buf* (который должен иметь размер *len*) в дескриптор большого объекта *fd*. В *fd* должно передаваться значение, возвращённое предыдущим вызовом `lo_open`. Возвращает эта функция число фактически записанных байт (в текущей реализации это всегда *len*, если только не произошла ошибка). В случае ошибки возвращается значение -1.

Хотя параметр *len* объявлен как `size_t`, эта функция не принимает значение длины, превышающее `INT_MAX`. На практике всё равно лучше передавать данные фрагментами не больше нескольких мегабайт.

33.3.6. Чтение данных из большого объекта

Функция

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

читает до *len* байт из дескриптора большого объекта *fd* в буфер *buf* (который должен иметь размер *len*). В *fd* должно передаваться значение, возвращённое предыдущим вызовом `lo_open`. Возвращает эта функция число фактически прочитанных байт; это число должно быть меньше *len*, если при чтении был достигнут конец объекта. В случае ошибки возвращается -1.

Хотя параметр *len* объявлен как `size_t`, эта функция не принимает значение длины, превышающее `INT_MAX`. На практике всё равно лучше передавать данные фрагментами не больше нескольких мегабайт.

33.3.7. Перемещение в большом объекте

Чтобы изменить текущее положение чтения или записи, связанное с дескриптором большого объекта, вызовите

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

Эта функция перемещает указатель текущего положения для дескриптора большого объекта *fd* в новое положение, заданное аргументом *offset*. Для аргумента *whence* задаются значения `SEEK_SET` (перемещение от начала объекта), `SEEK_CUR` (перемещение от текущего положения) и `SEEK_END` (перемещение от конца объекта). Возвращает эта функция новое положение указателя, либо -1 в случае ошибки.

Оперируя с большими объектами, размер которых превышает 2 ГБ, используйте

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

Эта функция действует так же, как и `lo_lseek`, но может принять значение *offset*, превышающее 2 ГБ, и/или вернуть результат, превышающий 2 ГБ. Заметьте, что если новое положение указателя оказывается за границей в 2ГБ, функция `lo_lseek` выдаёт ошибку.

Функция `lo_lseek64` появилась в PostgreSQL 9.3. Если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено -1.

33.3.8. Получение текущего положения в большом объекте

Чтобы получить текущее положение чтения или записи для дескриптора большого объекта, вызовите

```
int lo_tell(PGconn *conn, int fd);
```

Если возникает ошибка, возвращается -1.

Оперируя с большими объектами, размер которых может превышать 2 ГБ, используйте

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

Эта функция действует так же, как `lo_tell`, но может выдавать результат, превышающий 2 ГБ. Заметьте, что `lo_tell` выдаёт ошибку, если текущее положение чтения/записи оказывается за границей в 2 ГБ.

Функция `lo_tell64` появилась в PostgreSQL 9.3. Если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено -1.

33.3.9. Усечение большого объекта

Чтобы усечь большой объект до требуемой длины, вызовите

```
int lo_truncate(PGconn *conn, int fd, size_t len);
```

Эта функция усекает большой объект с дескриптором *fd* до длины *len*. В *fd* должно передаваться значение, возвращённое предыдущим вызовом `lo_open`. Если *len* превышает текущую длину большого объекта, большой объект расширяется до заданной длины нулевыми байтами ('\0'). В случае успеха `lo_truncate` возвращает ноль, а при ошибке возвращается -1.

Положение чтения/записи, связанное с дескриптором *`fd`*, при этом не меняется.

Хотя параметр *`len`* объявлен как `size_t`, `lo_truncate` не принимает значение длины, превышающее `INT_MAX`.

Оперируя с большими объектами, размер которых может превышать 2 ГБ, используйте

```
int lo_truncate64(PGconn *conn, int fd, pg_int64 len);
```

Эта функция действует так же, как `lo_truncate`, но может принимать значения *`len`*, превышающие 2 ГБ.

Функция `lo_truncate` появилась в PostgreSQL 8.3; если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено -1.

Функция `lo_truncate64` появилась в PostgreSQL 9.3; если попытаться выполнить её с сервером более старой версии, произойдёт ошибка и будет возвращено -1.

33.3.10. Закрытие дескриптора большого объекта

Дескриптор большого объекта можно закрыть, вызвав

```
int lo_close(PGconn *conn, int fd);
```

Здесь *`fd`* — дескриптор большого объекта, возвращённый функцией `lo_open`. В случае успеха `lo_close` возвращает ноль. При ошибке возвращается -1.

Все дескрипторы больших объектов, остающиеся открытыми в конце транзакции, закрываются автоматически.

33.3.11. Удаление большого объекта

Чтобы удалить большой объект из базы данных, вызовите

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

В аргументе *`lobjId`* задаётся OID большого объекта, который нужно удалить. В случае успеха возвращается 1, а в случае ошибки -1.

33.4. Серверные функции

Функции, предназначенные для работы с большими объектами на стороне сервера из SQL, перечислены в Таблице 33.1.

Таблица 33.1. SQL-ориентированные функции для работы с большими объектами

Функция **Описание** **Примеры**

`lo_from_bytea` (*`loid`* `oid`, *`data`* `bytea`) → `oid`
Создаёт большой объект и сохраняет в нём переданные данные (*`data`*). Если *`loid`* равен нулю, система выбирает свободный OID, а иначе использует заданный идентификатор OID (и выдаёт ошибку, если этот OID уже назначен какому-то большому объекту). В случае успеха возвращается OID созданного большого объекта.
`lo_from_bytea(0, '\xffffff00')` → `24528`

`lo_put` (*`loid`* `oid`, *`offset`* `bigint`, *`data`* `bytea`) → `void` Записывает данные (*`data`*) в большой объект по заданному смещению; в случае необходимости большой объект расширяется. `lo_put(24528, 1, '\xaa')` →

Функция **Описание** **Примеры**
`lo_get` (*loid* oid [, *offset* bigint, *length* integer]) → bytea Извлекает содержимое большого объекта или его часть. `lo_get(24528, 0, 3)` → `\xffaaff`

Каждой из клиентских функций, описанных ранее, соответствуют дополнительные функции на стороне сервера; на самом деле, по большей части клиентские функции представляют собой просто интерфейсы к равнозначным серверным функциям. К функциям, которые так же удобно вызывать командами SQL, относятся: `lo_creat`, `lo_create`, `lo_unlink`, `lo_import`и `lo_export`. Ниже приведены примеры их использования:

```
CREATE TABLE image (
    name            text,
    raster          oid
);

SELECT lo_creat(-1);       -- возвращает OID нового пустого большого объекта

SELECT lo_create(43213);   -- пытается создать большой объект с OID 43213

SELECT lo_unlink(173454);  -- удаляет большой объект с OID 173454

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster)  -- то же, что выше, но с предопределённым OID
    VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

Серверные функции `lo_import` и `lo_export` значительно отличаются от их аналогов, выполняемых на стороне клиента. Эти две функции читают и пишут файлы в файловой системе сервера с правами пользователя-владельца базы данных. Поэтому по умолчанию использовать их разрешено только суперпользователям. Клиентские функции импорта и экспорта, напротив, используют права клиентской программы для чтения и записи файлов в файловой системе клиента. Для выполнения клиентских функций никакие права в базе данных не требуется, кроме права на чтение или запись требуемого большого объекта.

Внимание

Использование серверных функций `lo_import` и `lo_export` можно разрешить (воспользовавшись GRANT) не только суперпользователям, но при этом следует серьёзно оценить возможные угрозы безопасности. Злонамеренный пользователь, имея такие права, легко может стать суперпользователем (например, перезаписав файлы конфигурации сервера) или атаковать другие области файловой системы сервера, не стремясь получить полномочия суперпользователя в базе данных. *Поэтому доступ к ролям с такими правами должен ограничиваться так же строго, как и доступ к ролям суперпользователей.* Тем не менее, если серверные функции `lo_import` и `lo_export` необходимо применять для каких-либо регулярных операций, безопаснее будет использовать роль с такими правами, чем с правами суперпользователя, так как это помогает предохраниться от случайных ошибок.

Функциональность `lo_read` и `lo_write` также представляется через вызовы на стороне сервера, но имена серверных функций, в отличие от клиентских, не содержат символы подчёркивания. Эти функции нужно вызывать по именам `loread` и `lowrite`.

33.5. Пример программы

В Примере 33.1 представлена пробная программа, демонстрирующая использование интерфейса больших объектов в libpq. Части этой программы закомментированы, но оставлены в тексте для читателя.

Пример 33.1. Пример использования больших объектов с применением libpq

```
/*-------------------------------------------------------------------------
 *
 * testlo.c
 *    test using large objects with libpq
 *
 * Portions Copyright (c) 1996-2024, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    src/test/examples/testlo.c
 *
 *-------------------------------------------------------------------------
 */
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE         1024

/*
 * importFile -
 *    import file "in_filename" into database as large object "lobjOid"
 *
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file\"%s\"\n", filename);
    }
```

```
    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading \"%s\"", filename);
    }

    close(fd);
    lo_close(conn, lobj_fd);

    return lobjId;
}

static void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\0';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;              /* no more data? */
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
```

```
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nWRITE FAILED!\n");
            break;
        }
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}


/*
 * exportFile -
 *    export large object "lobjOid" to file "out_filename"
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the large object
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    /*
     * open the file to be written to
     */
```

```
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file\"%s\"",
                filename);
    }

    /*
     * read in from the inversion file and write to the Unix file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing \"%s\"",
                    filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);
}

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
```

```
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* Set always-secure search path, so malicious users can't take control. */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file \"%s\" ...\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    if (lobjOid == 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
    else
    {
        printf("\tas large object %u.\n", lobjOid);

        printf("picking out bytes 1000-2000 of the large object\n");
        pickout(conn, lobjOid, 1000, 1000);

        printf("overwriting bytes 1000-2000 of the large object with X's\n");
        overwrite(conn, lobjOid, 1000, 1000);

        printf("exporting large object to file \"%s\" ...\n", out_filename);
/*      exportFile(conn, lobjOid, out_filename); */
        if (lo_export(conn, lobjOid, out_filename) < 0)
            fprintf(stderr, "%s\n", PQerrorMessage(conn));
    }

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

Глава 34. ECPG — Встраиваемый SQL в C

В этой главе описывается встраиваемый SQL для Postgres Pro. Авторы этой разработки: Линус Толке (`<linus@epact.se>`) и Михаэль Мескес (`<meskes@postgresql.org>`). Изначально этот проект предназначался для C. Сейчас он также работает с C++, но пока не распознаёт все конструкции C++.

Эта документация не абсолютно полная, но так как этот интерфейс стандартизирован, дополнительные сведения можно почерпнуть во многих ресурсах, посвящённых SQL.

34.1. Концепция

Программа со встраиваемым SQL состоит из кода, написанного на обычном языке программирования, в данном случае C, дополненного командами SQL в специально обозначенных секциях. Чтобы собрать программу, её исходный код (`*.pgc`) сначала нужно пропустить через препроцессор встраиваемого SQL, который превратит её в обычную программу на C (`*.c`), воспринимаемую компилятором C. (Подробнее компиляция и компоновка описывается в Разделе 34.10). Преобразованные приложения ECPG вызывают функции в библиотеке libpq через библиотеку встраиваемого SQL (ecpglib) и взаимодействуют с сервером Postgres Pro по обычному клиент-серверному протоколу.

Встраиваемый SQL имеет ряд преимуществ по сравнению с другими методами вызова команд SQL из кода C. Во-первых, этот механизм берёт на себя заботу о передаче информации через переменные в программе на C. Во-вторых, код SQL в программе проверяется на синтаксическую правильность во время сборки. В-третьих, встраиваемый SQL в C описан стандартом SQL и поддерживается многими другими СУБД SQL. Реализация в Postgres Pro разработана так, чтобы максимально соответствовать этому стандарту, поэтому обычно достаточно легко портировать в Postgres Pro программы с встраиваемым SQL, написанные для других СУБД.

Как уже сказано, программы, написанные для интерфейса встраиваемого SQL, представляют собой обычные программы на C с добавленным специальным кодом, который выполняет действия, связанные с базой данных. Этот специальный код всегда имеет следующую форму:

```
EXEC SQL ...;
```

Такие операторы синтаксически занимают место операторов C. В зависимости от конкретного оператора, они могут размещаться на глобальном уровне или внутри функции.

Встраиваемые операторы SQL следуют правилам учёта регистра, принятым в обычном коде SQL, а не в C. Они также допускают вложенные комментарии в стиле C, разрешённые стандартом SQL. Однако остальная часть программы, написанная на C, в соответствии со стандартом C содержать вложенные комментарии не может. Также правила SQL, а не C, действуют при разборе строк в кавычках и идентификаторов во встраиваемых операторах SQL. (См. Подраздел 4.1.2.1 и Подраздел 4.1.1 соответственно. Обратите внимание: ECPG предполагает, что `standard_conforming_strings` имеет значение `on`). Конечно, часть программы на C следует правилам цитирования C.

Все встраиваемые SQL-операторы рассматриваются в следующих разделах.

34.2. Управление подключениями к базе данных

В этом разделе описывается, как открывать, закрывать и переключать подключения к базам данных.

34.2.1. Подключение к серверу баз данных

Подключение к базе данных выполняется следующим оператором:

```
EXEC SQL CONNECT TO цель-подключения [AS имя-подключения] [USER имя-пользователя];
```

`Цель` может задаваться следующими способами:

- `dbname[@имя_сервера][:порт]`

- `tcp:postgresql://`*`имя_сервера`*`[:`*`порт`*`][/`*`dbname`*`][?`*`параметры`*`]`
- `unix:postgresql://localhost[:`*`порт`*`][/`*`dbname`*`][?`*`параметры`*`]`
- строковая константа SQL, содержащая одну из вышеприведённых записей
- ссылка на символьную переменную, содержащую одну из вышеприведённых записей (см. примеры)
- `DEFAULT`

С целью подключения `DEFAULT` устанавливается подключение к базе данных по умолчанию с именем пользователя по умолчанию. Другое имя пользователя или имя подключения в этом случае указать нельзя.

Если цель подключения задаётся буквально (не в виде текстовой строки и не через переменную), её компоненты разбираются как обычные элементы SQL; это означает, например, что *`имя_сервера`* должно выглядеть как один или несколько идентификаторов SQL, разделённых точками, и если они не заключены в кавычки, они будут приведены к нижнему регистру. Значения любых *`параметров`* должны быть идентификаторами SQL, целыми числами или ссылками на переменные. Разумеется, практически любую строку можно сделать идентификатором SQL, заключив её в кавычки. На практике же, во избежание ошибок, предпочтительнее использовать текстовые строки (в апострофах) или ссылки на переменные.

Также разными способами можно указать имя пользователя:

- *`имя_пользователя`*
- *`имя_пользователя`*`/`*`пароль`*
- *`имя_пользователя`* `IDENTIFIED BY` *`пароль`*
- *`имя_пользователя`* `USING` *`пароль`*

В показанных выше строках *`имя_пользователя`* и *`пароль`* могут задаваться идентификатором или строковой константой SQL, либо ссылкой на символьную переменную.

Если указание цели подключения включает какие-либо *`параметры`*, они должны записываться в виде *`имя`*`=`*`значение`* и разделяться амперсандами (`&`). В качестве имён параметров принимаются те же, что поддерживает libpq (см. Подраздел 32.1.2). Перед элементами *`имя`* или *`значение`* пробелы игнорируются, но сохраняются внутри или после этих элементов. Заметьте, что записать `&` внутри *`значения`* нет никакой возможности.

Обратите внимание, что при указании соединения через сокет (с префиксом `unix:`) имя сервера обязательно должно быть `localhost`. Чтобы выбрать нестандартный каталог сокета, укажите путь к каталогу как значение параметра `host` в списке *`параметров`* в строке цели.

Указание *`имя-подключения`* применяется, когда в одной программе нужно использовать несколько подключений. Его можно опустить, если программа работает только с одним подключением. Соединение, открытое последним, становится текущим и будет использоваться по умолчанию при выполнении SQL-операторов (это описывается далее в этой главе).

Вот некоторые примеры оператора `CONNECT`:

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO tcp:postgresql://sql.mydomain.com/mydb AS myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
 ...
```

```
EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* or EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

В последней форме используется вариант, названный выше ссылкой на символьную переменную. В последующих разделах вы узнаете, как в SQL-операторах можно использовать переменные C, приставляя перед именем двоеточие.

Учтите, что формат цели подключения не описывается в стандарте SQL. Поэтому, если вы хотите разрабатывать переносимые приложения, имеет смысл применить подход, показанный в последнем примере, и сформировать строку подключения отдельно.

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления схем, доступных всем для записи, из пути поиска (`search_path`). Например, добавьте `options=-c search_path=` в *`options`* или выполните `EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);` после подключения. Это касается не только ECPG, но и любых других интерфейсов для выполнения произвольных SQL-команд.

34.2.2. Выбор подключения

SQL-операторы в программах со встраиваемым SQL по умолчанию выполняются с текущим подключением, то есть с подключением, которое было открыто последним. Если приложению нужно управлять несколькими подключениями, это можно сделать тремя способами.

Первый вариант — явно выбирать подключение для каждого SQL-оператора, например, так:

```
EXEC SQL AT имя-подключения SELECT ...;
```

Этот вариант хорошо подходит для случаев, когда приложению нужно использовать несколько подключений в смешанном порядке.

Если ваше приложение выполняется в нескольких потоках, они не могут использовать подключение одновременно. Поэтому вы должны либо явно управлять доступом (используя мьютексы), либо использовать отдельные подключения для каждого потока.

Второй вариант — выполнять оператор, переключающий текущее подключение. Этот оператор записывается так:

```
EXEC SQL SET CONNECTION имя-подключения;
```

Этот вариант особенно удобен, когда с одним подключением нужно выполнить несколько операторов.

Следующий пример программы демонстрирует управление несколькими подключениями к базам данных:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    /* This query would be executed in the last opened database "testdb3". */
```

```
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);

    /* Using "AT" to run a query in "testdb2" */
    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);

    /* Switch the current connection to "testdb1". */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);

    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

Этот пример должен вывести следующее:

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

Третий вариант заключается в объявлении идентификатора SQL, связанного с подключением. Например:

```
EXEC SQL AT имя-подключения DECLARE имя-оператора STATEMENT;
EXEC SQL PREPARE имя-оператора FROM :динамическая-строка;
```

Связанный с подключением SQL-идентификатор затем нужно выполнять без предложения AT. Обратите внимание, что этот вариант действует как директивы препроцессора, поэтому это связывание будет работать только в рамках файла.

Следующий пример программы демонстрирует использование данного варианта:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
char dbname[128];
char *dyn_sql = "SELECT current_database()";
EXEC SQL END DECLARE SECTION;

int main(){
  EXEC SQL CONNECT TO postgres AS con1;
  EXEC SQL CONNECT TO testdb AS con2;
  EXEC SQL AT con1 DECLARE stmt STATEMENT;
  EXEC SQL PREPARE stmt FROM :dyn_sql;
  EXEC SQL EXECUTE stmt INTO :dbname;
  printf("%s\n", dbname);

  EXEC SQL DISCONNECT ALL;
  return 0;
}
```

Этот пример должен вывести следующее, даже если подключение по умолчанию установлено к базе testdb:

```
postgres
```

34.2.3. Закрытие подключения

Чтобы закрыть подключение, примените следующий оператор:

```
EXEC SQL DISCONNECT [подключение];
```

`Подключение` можно задать следующими способами:

- *`имя-подключения`*
- `CURRENT`
- `ALL`

Если имя подключения не задано, закрывается текущее подключение.

Хорошим стилем считается, когда приложение явно закрывает каждое подключение, которое оно открыло.

34.3. Запуск команд SQL

В приложении со встраиваемым SQL можно запустить любую команду SQL. Ниже приведены несколько примеров, показывающих как это делать.

34.3.1. Выполнение SQL-операторов

Создание таблицы:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

Добавление строк:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Удаление строк:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Изменение:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

Операторы `SELECT`, возвращающие одну строку результата, также могут выполняться непосредственно командой `EXEC SQL`. Чтобы обработать наборы результатов с несколькими строками, приложение должно использовать курсоры; см. Подраздел 34.3.2 ниже. (В отдельных случаях приложение может выбрать сразу несколько строк в переменную массива; см. Подраздел 34.4.4.3.1.)

Выборка одной строки:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Так же можно получить параметр конфигурации командой `SHOW`:

```
EXEC SQL SHOW search_path INTO :var;
```

Идентификаторы вида `:`*`имя`* воспринимаются как *переменные среды*, то есть они ссылаются на переменные программы C. Они рассматриваются в Разделе 34.4.

34.3.2. Использование курсоров

Чтобы получить набор результатов, содержащий несколько строк, приложение должно объявить курсор и выбирать каждую строку через него. Использование курсора подразумевает следующие шаги: объявление курсора, открытие его, выборку строки через курсор, повторение предыдущего шага, и наконец, закрытие курсора.

Выборка с использованием курсоров:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Более подробно объявление курсора описывается в DECLARE, а выборка строк через курсор — в FETCH.

> **Примечание**
>
> Команда `DECLARE` в ECPG на самом деле не передаёт этот оператор серверу Postgres Pro. Курсор открывается на сервере (командой сервера `DECLARE`) в момент, когда выполняется команда `OPEN`.

34.3.3. Управление транзакциями

В режиме по умолчанию операторы фиксируются только когда выполняется `EXEC SQL COMMIT`. Интерфейс встраиваемого SQL также поддерживает автофиксацию транзакций (так работает libpq по умолчанию); она включается аргументом командной строки `-t` программы `ecpg` (см. ecpg) либо оператором `EXEC SQL SET AUTOCOMMIT TO ON`. В режиме автофиксации каждая команда фиксируется автоматически, если только она не помещена в явный блок транзакции. Этот режим можно выключить явным образом, выполнив `EXEC SQL SET AUTOCOMMIT TO OFF`.

Поддерживаются следующие команды управления транзакциями:

`EXEC SQL COMMIT`

Зафиксировать текущую транзакцию.

`EXEC SQL ROLLBACK`

Откатить текущую транзакцию.

`EXEC SQL PREPARE TRANSACTION`*ид_транзакции*

Подготовить текущую транзакцию для двухфазной фиксации.

`EXEC SQL COMMIT PREPARED`*ид_транзакции*

Зафиксировать транзакцию в подготовленном состоянии.

`EXEC SQL ROLLBACK PREPARED`*ид_транзакции*

Откатить транзакцию в подготовленном состоянии.

`EXEC SQL SET AUTOCOMMIT TO ON`

Включить режим автофиксации.

`EXEC SQL SET AUTOCOMMIT TO OFF`

Отключить режим автофиксации. По умолчанию он отключён.

34.3.4. Подготовленные операторы

Когда значения, передаваемые SQL-оператору, неизвестны во время компиляции, или один и тот же оператор будет использоваться многократно, могут быть полезны подготовленные операторы.

Оператор подготавливается командой `PREPARE`. Вместо значений, которые ещё неизвестны, вставляются местозаполнители «?»:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
```

Если оператор возвращает одну строку, приложение может вызвать `EXECUTE` после `PREPARE` для выполнения этого оператора, указав фактические значения для местозаполнителей в предложении `USING`:

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

Если оператор возвращает несколько строк, приложение может использовать курсор, объявленный на базе подготовленного оператора. Чтобы привязать входные параметры, курсор нужно открыть с предложением `USING`:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

/* по достижении конца набора результатов прервать цикл while */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

Когда подготовленный оператор больше не нужен, его следует освободить:

```
EXEC SQL DEALLOCATE PREPARE имя;
```

Подробнее оператор `PREPARE` описан в PREPARE. Также обратитесь к Разделу 34.5 за дополнительными сведениями о местозаполнителях и входных параметрах.

34.4. Использование переменных среды

В Разделе 34.3 вы увидели, как можно выполнять SQL-операторы в программе со встраиваемым SQL. Некоторые из этих операторов использовали только фиксированные значения и не давали возможности вставлять в операторы произвольные значения или обрабатывать значения, возвращённые запросом. Операторы такого вида не очень полезны в реальных приложениях. В этом разделе подробно описывается, как можно передавать данные между программой на C и встраиваемыми операторами SQL, используя простой механизм, так называемые *переменные среды*. В программе со встраиваемым SQL мы считаем SQL-операторы *внедрёнными* в код программы на C, *языке среды*. Таким образом, переменные программы на C называются *переменными среды*.

Ещё один способ передать значения данных между сервером Postgres Pro и приложениями ECPG заключается в использовании дескрипторов SQL, как описано в Разделе 34.7.

34.4.1. Обзор

Передавать данные между программой C и SQL-операторами во встраиваемом SQL очень просто. Вместо того, чтобы вставлять данные в оператор, что влечёт дополнительные усложнения, в частности нужно правильно заключать значения в кавычки, можно просто записать имя переменной C в SQL-операторе, предварив его двоеточием. Например:

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

Этот оператор обращается к двум переменным C с именами `v1` и `v2` и также использует обычную строковую константу SQL, показывая тем самым, что можно свободно сочетать разные виды данных.

Этот метод включения переменных C в SQL-операторы работает везде, где SQL-оператор принимает выражение значения.

34.4.2. Секции объявлений

Чтобы передать данные из программы в базу данных, например, в виде параметров запроса, либо получить данные из базы данных в программе, переменные C, которые должны содержать эти данные, нужно объявить в специально помеченных секциях, чтобы препроцессор встраиваемого SQL знал о них.

Секция начинается с:

```
EXEC SQL BEGIN DECLARE SECTION;
```

и заканчивается командой:

```
EXEC SQL END DECLARE SECTION;
```

Между этими строками должны располагаться обычные объявления переменных C, например:

```
int   x = 4;
char  foo[16], bar[16];
```

Как здесь показано, переменной можно присвоить начальное значение. Область видимости переменной определяется расположением секции, в которой она объявляется в программе. Вы также можете объявить переменную следующим образом (при этом неявно создаётся секция объявлений):

```
EXEC SQL int i = 4;
```

Вы можете включать в программу столько секций объявлений, сколько захотите.

Эти объявления выводятся в результирующий файл как объявления обычных переменных C, так что эти переменные не нужно объявлять снова. Переменные, которые не предназначены для использования в командах SQL, можно объявить как обычно вне этих специальных секций.

Определение структуры или объединения тоже должно размещаться в секции `DECLARE`. В противном случае препроцессор не сможет воспринять эти типы, так как не будет знать их определения.

34.4.3. Получение результатов запроса

Теперь вы умеете передавать данные, подготовленные вашей программой, в команду SQL. Но как получить результаты запроса? Для этой цели во встраиваемом SQL есть особые вариации обычных команд `SELECT` и `FETCH`. У этих команд есть специальное предложение `INTO`, определяющее, в какие переменные среды будут помещены получаемые значения. `SELECT` используется для запросов, возвращающих только одну строку, а `FETCH` применяется с курсором для запросов, возвращающих несколько строк.

Пример:

```
/*
 * предполагается существование такой таблицы:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;
```

```
 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

Предложение `INTO` размещается между списком выборки и предложением `FROM`. Число элементов в списке выборки должно равняться числу элементов в списке после `INTO` (также называемом целевым списком).

Следующий пример демонстрирует использование команды `FETCH`:

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
} while (...);
```

Здесь предложение `INTO` размещается после всех остальных обычных предложений.

34.4.4. Сопоставление типов

Когда приложения ECPG передают данные между сервером Postgres Pro и программой на C, например, получая результаты запроса с сервера или выполняя операторы SQL с входными параметрам, эти данные должны преобразовываться из типов Postgres Pro в типы переменных языка среды (а именно типы языка C) и наоборот. Одно из главных качеств ECPG состоит в том, что в большинстве случаев он делает это автоматически.

В этом отношении можно выделить два вида типов данных. К первому относятся простые типы данных Postgres Pro, такие как `integer` и `text`, которые приложение может непосредственно читать и писать. С другими типами данных, такими как `timestamp` и `numeric`, можно работать только через специальные функции; см. Подраздел 34.4.4.2.

В Таблице 34.1 показано, как типы данных Postgres Pro соответствуют типам данных C. Когда нужно передать или получить значение определённого типа данных Postgres Pro, вы должны объявить переменную C соответствующего типа C в секции объявлений.

Таблица 34.1. Соответствие между типами данных Postgres Pro и типами переменных C

Тип данных Postgres Pro	Тип переменной среды C
`smallint`	`short`
`integer`	`int`
`bigint`	`long long int`
`decimal`	`decimal`[a]
`numeric`	`numeric`[a]
`real`	`float`
`double precision`	`double`

Тип данных Postgres Pro	Тип переменной среды C
`smallserial`	`short`
`serial`	`int`
`bigserial`	`long long int`
`oid`	`unsigned int`
`character(` *n*`)`, `varchar(` *n*`)`, `text`	`char[`*n*`+1]`, `VARCHAR[`*n*`+1]`
`name`	`char[NAMEDATALEN]`
`timestamp`	`timestamp`[a]
`interval`	`interval`[a]
`date`	`date`[a]
`boolean`	`bool`[b]
`bytea`	`char *`, `bytea[`*n*`]`

[a] С этим типом можно работать только через специальные функции; см. Подраздел 34.4.4.2.

[b] объявляется в `ecpglib.h` при отсутствии стандартного объявления

34.4.4.1. Работа с символьными строками

Для обработки типов символьных строк SQL, таких как `varchar` и `text`, предлагаются два варианта объявления переменных среды.

Первый способ заключается в использовании `char[]`, массива `char`, как чаще всего и представляются символьные данные в C.

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

Заметьте, что о длине этого массива вы должны подумать сами. Если вы укажете данную переменную среды C в качестве целевой переменной запроса, возвращающего строку длиннее 49 символов, произойдёт переполнение буфера.

В качестве другого подхода можно использовать специальный тип `VARCHAR`, представленный в ECPG. Определение массива типа `VARCHAR` преобразуется в структуру (`struct`) с собственным именем для каждой переменной. Объявление вида:

```
VARCHAR var[180];
```

преобразуется в:

```
struct varchar_var { int len; char arr[180]; } var;
```

Член структуры `arr` содержит строку, включающую завершающий нулевой байт. Таким образом, чтобы сохранить строку в переменной типа `VARCHAR`, эта переменная должна быть объявлена с длиной, учитывающей завершающий нулевой байт. Член структуры `len` содержит длину строки, сохранённой в `arr`, без завершающего нулевого байта. Когда на вход запросу подаётся переменная среды C, у которой `strlen(arr)` отличается от `len`, применяется наименьшее значение.

`VARCHAR` можно записать в верхнем или нижнем регистре, но не в смешанном.

Переменные `char` и `VARCHAR` также могут содержать значения других типов SQL в их строковом представлении.

34.4.4.2. Обработка специальных типов данных

ECPG представляет некоторые особые типы, которые должны помочь вам легко оперировать некоторыми специальными типами данных Postgres Pro. В частности, в нём реализована поддержка типов `numeric`, `decimal`, `date`, `timestamp` и `interval`. Для этих типов нельзя подобрать полезное соответствие с примитивными типами среды (например, `int`, `long long int` или `char[]`), так как они имеют сложную внутреннюю структуру. Приложения, работающие с этими типами, должны объявлять переменные особых типов и работать с ними, применяя функции из библиотеки pgtypes. Эта библиотека, подробно описанная в Разделе 34.6 содержит базовые функции для оперирования этими типами, чтобы вам не требовалось, например, передавать запрос SQL-серверу, когда нужно просто добавить интервал к значению времени.

Эти особые типы данных описаны в следующих подразделах. Чтобы подробнее узнать о функциях в библиотеке pgtypes, обратитесь к Разделу 34.6.

34.4.4.2.1. timestamp, date

Для работы с переменными `timestamp` в приложении ECPG применяется следующая схема.

Сначала в программу нужно включить заголовочный файл, чтобы получить определение типа `timestamp`:

```
#include <pgtypes_timestamp.h>
```

Затем объявите в секции объявлений переменную типа `timestamp`:

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

Прочитав значение в эту переменную, выполняйте действия с ним, используя функции в библиотеке pgtypes. В следующем примере значение `timestamp` преобразуется в текстовый вид (ASCII) с помощью функции `PGTYPEStimestamp_to_asc()`:

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

Этот пример выведет такой результат:

```
ts = 2010-06-27 18:03:56.949343
```

Таким же образом можно работать и с типом DATE. В программу нужно включить `pgtypes_date.h`, объявить переменную типа date, и затем можно будет преобразовать значение DATE в текстовый вид, используя функцию `PGTYPESdate_to_asc()`. Чтобы подробнее узнать о функциях в библиотеке pgtypes, обратитесь к Разделу 34.6.

34.4.4.2.2. interval

Принцип работы с типом `interval` тот же, что и с типами `timestamp` и `date`, однако для значения типа `interval` нужно явно выделить память. Другими словами, блок памяти для этой переменной должен размещаться в области кучи, а не в стеке.

Пример программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    in = PGTYPESinterval_new();
```

```
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

34.4.4.2.3. numeric, decimal

Типы `numeric` и `decimal` обрабатываются так же, как и тип `interval`: вы должны определить указатель, выделить некоторое пространство памяти в куче и обращаться к переменной, используя функции в библиотеке pgtypes. Чтобы подробнее узнать о функциях в библиотеке pgtypes, обратитесь к Разделу 34.6.

Для типа `decimal` никакие специальные функции не реализованы. Для дальнейшей обработки приложение должно преобразовать его в переменную `numeric`, применив функцию из библиотеки pgtypes.

Следующий пример демонстрирует работу с переменными типов `numeric` и `decimal`.

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    num = PGTYPESnumeric_new();
    dec = PGTYPESdecimal_new();

    EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 0));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 1));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 2));

    /* Преобразовать decimal в numeric, чтобы вывести десятичное значение. */
    num2 = PGTYPESnumeric_new();
    PGTYPESnumeric_from_decimal(dec, num2);

    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 0));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 1));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 2));

    PGTYPESnumeric_free(num2);
    PGTYPESdecimal_free(dec);
    PGTYPESnumeric_free(num);
```

```
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

34.4.4.2.4. bytea

Работа с типом `bytea` организуется так же, как и с `VARCHAR`. Определение каждой переменной-массива типа `bytea` преобразуется в именованную структуру. Например, объявление:

```
bytea var[180];
```

преобразуется в:

```
struct bytea_var { int len; char arr[180]; } var;
```

Двоичные данные формата помещаются в поле `arr`. В отличие от типа `VARCHAR`, в данных `bytea` могут быть восприняты значения `'\0'`. Эти данные записываются/считываются и переводятся из шестнадцатеричного формата и обратно средствами ecpglib.

> **Примечание**
>
> Переменные `bytea` можно использовать, только когда параметр bytea_output равен `hex`.

34.4.4.3. Переменные среды для непримитивных типов

В качестве переменных среды также можно использовать массивы, определения типов, структуры и указатели.

34.4.4.3.1. Массивы

Для применения массивов в качестве переменных среды есть два варианта использования. Во-первых, в массиве `char[]` или `VARCHAR[]` можно сохранить текстовую строку, как рассказывалось в Подразделе 34.4.4.1. Во-вторых, в массив можно получить несколько строк из результата запроса, не используя курсор. Чтобы не применяя массивы, обработать результат запроса, состоящий из нескольких строк, нужно использовать курсор и команду `FETCH`. Но с переменными-массивами несколько строк можно получить сразу. Длина определяемого массива должна быть достаточной для размещения всех строк, иначе скорее всего произойдёт переполнение буфера.

Следующий пример сканирует системную таблицу `pg_database` и показывает все OID и имена доступных баз данных:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    /* Получить в массивы сразу несколько строк. */
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;
```

```
    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

Этот пример выводит следующий результат. (Точные значения зависят от локальных обстоятельств.)

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

34.4.4.3.2. Структуры

Для получения значений сразу нескольких столбцов можно применить структуру, имена членов которой совпадают с именами столбцов результата запроса. Структура позволяет обрабатывать значения нескольких столбцов в одной переменной среды C.

Следующий пример получает значения OID, имена и размеры имеющихся баз данных из системной таблицы `pg_database`, используя при этом функцию `pg_database_size()`. В этом примере переменная типа структуры `dbinfo_t` с членами, имена которых соответствуют именам всех столбцов результата `SELECT`, применяется для получения одной строки результата без вовлечения в оператор `FETCH` нескольких переменных среды.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
       long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size
 FROM pg_database;
    EXEC SQL OPEN cur1;

    /* по достижении конца набора результатов прервать цикл while */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* Выбрать несколько столбцов в одну структуру. */
        EXEC SQL FETCH FROM cur1 INTO :dbval;

        /* Напечатать члены структуры. */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname,
 dbval.size);
    }
```

```
    EXEC SQL CLOSE cur1;
```

Этот пример показывает следующий результат. (Точные значения зависят от локальных обстоятельств.)

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012
```

Переменные среды типа структуры «вбирают в себя» столько столбцов, сколько полей содержит структура. Значения дополнительных столбцов можно присвоить другим переменным среды. Например, приведённую выше программу можно видоизменить следующим образом, разместив переменную `size` вне структуры:

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
    } dbinfo_t;

    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size
 FROM pg_database;
    EXEC SQL OPEN cur1;

    /* по достижении конца набора результатов прервать цикл while */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* Выбрать несколько столбцов в одну структуру. */
        EXEC SQL FETCH FROM cur1 INTO :dbval, :size;

        /* Напечатать члены структуры. */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, size);
    }

    EXEC SQL CLOSE cur1;
```

34.4.4.3.3. Определения типов

Чтобы сопоставить новые типы с уже существующими, используйте ключевое слово `typedef`.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

Заметьте, что вы также можете написать:

```
EXEC SQL TYPE serial_t IS long;
```

Это объявление не обязательно должно находиться в секции объявлений, то есть также можно писать определения типов как обычные операторы C.

Любое слово, которое объявляется как `typedef` (определение типа), не может быть использовано в качестве ключевого слова SQL в командах `EXEC SQL` позже в той же программе. Например, это не сработает:

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef int start;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL START TRANSACTION;
```

ECPG сообщит о синтаксической ошибке для `START TRANSACTION`, так как больше не распознаёт `START` как ключевое слово SQL, только как определение типа. (Если у вас есть такой конфликт и переименование определения типа кажется нецелесообразным, можно написать команду SQL, используя динамический SQL.)

Примечание

В выпусках PostgreSQL до 16 версии использование ключевых слов SQL в качестве имён определений типов скорее приводило к синтаксическим ошибкам, связанным с использованием самого определения типа, а не с использованием имени в качестве ключевого слова SQL. Новое поведение с меньшей вероятностью вызовет проблемы при перекомпиляции существующего приложения ECPG в новом выпуске PostgreSQL с новыми ключевыми словами.

34.4.4.3.4. Указатели

Вы можете объявлять указатели на самые распространённые типы. Учтите, однако, что указатели нельзя использовать в качестве целевых переменных запросов без автовыделения. За дополнительными сведениями об автовыделении обратитесь к Разделу 34.7.

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

34.4.5. Обработка непримитивных типов данных SQL

В этом разделе описывается как работать с нескалярными и пользовательскими типами уровня SQL в приложениях ECPG. Заметьте, что этот подход отличается от использования переменных непримитивных типов, описанного в предыдущем разделе.

34.4.5.1. Массивы

Многомерные массивы уровня SQL в ECPG напрямую не поддерживаются, но одномерные массивы уровня SQL могут быть сопоставлены с переменными-массивами среды C и наоборот. Однако учтите, что когда создаётся оператор, ecpg не знает типов столбцов, поэтому не может проверить, вводится ли массив C в соответствующий массив уровня SQL. Обрабатывая результат SQL-оператора, ecpg имеет необходимую информацию и таким образом может убедиться, что с обеих сторон массивы.

Если запрос обращается к отдельным *элементам* массива, это избавляет от необходимости применять массивы в ECPG. В этом случае следует использовать переменную среды C, имеющую тип, который можно сопоставить типу элемента. Например, если типом столбца является массив `integer`, можно использовать переменную среды типа `int`. Аналогично, если тип элемента — `varchar` или `text`, можно использовать переменную типа `char[]` или `VARCHAR[]`.

Предположим, что у нас есть таблица:

```
CREATE TABLE t3 (
    ii integer[]
```

```
);

testdb=> SELECT * FROM t3;
     ii
-------------
 {1,2,3,4,5}
(1 row)
```

Следующая программа получает 4-ый элемент массива и сохраняет его в переменной среды C, имеющей тип `int`:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}

EXEC SQL CLOSE cur1;
```

Этот пример выводит следующий результат:

```
ii=4
```

Чтобы сопоставить несколько элементов массива с несколькими элементами переменной-массивом среды, каждый элемент массива SQL нужно по отдельности связать с каждым элементом массива среды, например:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

Ещё раз обратите внимание, что в этом случае вариант

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;
```

```
while (1)
{
    /* НЕПРАВИЛЬНО */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

не будет работать корректно, так как столбец с типом массива нельзя напрямую сопоставить с переменной-массивом среды.

Можно также применить обходное решение — хранить массивы в их внешнем строковом представлении в переменных среды типа `char[]` или `VARCHAR[]`. Более подробно это представление описывается в Подразделе 8.15.2. Заметьте, это означает, что с таким массивом в программе нельзя будет работать естественным образом (без дополнительного разбора текстового представления).

34.4.5.2. Составные типы

Составные типы в ECPG напрямую не поддерживаются, но есть простое обходное решение. Для решения этой проблемы можно применить те же подходы, что были описаны выше для массивов: обращаться к каждому атрибуту по отдельности или использовать внешнее строковое представление.

Для следующих примеров предполагается, что существует такой тип и таблица:

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'Postgres Pro') );
```

Самое очевидное решение заключается в обращении к каждому атрибуту по отдельности. Следующая программа получает данные из тестовой таблицы, выбирая атрибуты типа `comp_t` по одному:

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/* Указать каждый элемент столбца составного типа в списке SELECT. */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Выбрать каждый элемент столбца составного типа в переменную среды C. */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}

EXEC SQL CLOSE cur1;
```

В развитие этого примера, переменные среды, в которые помещаются результаты команды `FETCH`, можно собрать в одну структуру. Подробнее переменные среды в форме структуры описываются в Подразделе 34.4.4.3.2. Чтобы перейти к структуре, пример можно изменить как показано ниже. Переменные среды, `intval` и `textval`, становятся членами структуры `comp_t`, и эта структура указывается в команде `FETCH`.

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
```

```
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;

/* Поместить каждый элемент составного типа в список SELECT. */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Поместить все значения списка SELECT в одну структуру. */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

Хотя в команде `FETCH` используется структура, имена атрибутов в предложении `SELECT` задаются по одному. Это можно дополнительно улучшить, написав `*`, что будет обозначать все атрибуты значения составного типа.

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Выбрать все значения в списке SELECT в одну структуру. */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

Таким образом составные типы можно сопоставить со структурами практически прозрачно, хотя ECPG и не понимает составные типы.

Наконец, также можно сохранить значения составного типа в их внешнем строковом представлении в переменных среды типа `char[]` или `VARCHAR[]`. Однако при таком подходе нет простой возможности обращаться из программы к полям значения.

34.4.5.3. Пользовательские базовые типы

Определяемые пользователем базовые типы не поддерживаются ECPG напрямую. Для них можно использовать внешнее строковое представление и переменные среды типа `char[]` или `VARCHAR[]`, и это решение действительно будет подходящим и достаточным для большинства типов.

Следующий фрагмент кода демонстрирует использование типа данных `complex` из примера в Разделе 36.13. Внешнее строковое представление этого типа имеет форму `(%f,%f)` и определено в функциях `complex_in()` и `complex_out()` в Разделе 36.13. Следующий пример вставляет значения комплексного типа `(1,1)` и `(3,3)` в столбцы `a` и `b`, а затем выбирает их из таблицы.

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
    EXEC SQL OPEN cur1;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH FROM cur1 INTO :a, :b;
        printf("a=%s, b=%s\n", a.arr, b.arr);
    }

    EXEC SQL CLOSE cur1;
```

Этот пример выводит следующий результат:

```
a=(1,1), b=(3,3)
```

Другое обходное решение состоит в том, чтобы избегать прямого использования пользовательских типов в ECPG, а вместо этого создать функцию или приведение, выполняющее преобразование между пользовательским типом и примитивным типом, который может обработать ECPG. Заметьте, однако, что приведения типов, особенно неявные, нужно добавлять в систему типов очень осторожно.

Например:

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0')' + $2 * complex '(0,1)' $$;
```

После такого определения следующий код

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b),
 create_complex(:c, :d));
```

будет работать так же, как

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

34.4.6. Индикаторы

Приведённые выше примеры никак не обрабатывали значения NULL. На самом деле, в примерах с извлечением данных возникнет ошибка, если они выберут из базы данных значение NULL. Чтобы можно было передавать значения NULL в базу данных или получать их из базы данных, вы должны добавить объявление второй переменной среды C для каждой переменной C, содержащей данные. Эта вторая переменная среды C называется *индикатором* и содержит флаг, показывающий, что в данных передаётся NULL, и при этом значение основной переменной среды C игнорируется. Следующий пример демонстрирует правильную обработку значений NULL:

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

Переменная индикатора `val_ind` будет равна нулю, если значение не NULL, или отрицательному числу, если NULL. (Особенности, связанные с режимом совместимости с Oracle, описаны в Разделе 34.16.)

Индикатор выполняет ещё одну функцию: если он содержит положительное число, это означает, что значение не NULL, но оно было обрезано, когда сохранялось в переменной среды C.

Если препроцессору `ecpg` передаётся аргумент `-r no_indicator`, он работает в режиме «без индикатора». В этом режиме, если переменная индикатора не определена, значения NULL обозначаются (при вводе и выводе) для символьных строк пустой строкой, а для целочисленных типов наименьшим возможным значением этого типа (например, `INT_MIN` для `int`).

34.5. Динамический SQL

Во многих случаях конкретные SQL-операторы, которые должно выполнять приложение, известны в момент написания приложения. В некоторых случаях, однако, SQL-операторы формируются во время выполнения или поступают из внешнего источника. В этих случаях SQL-операторы нельзя внедрить непосредственно в исходный код C, но есть средство, позволяющее вызывать произвольные SQL-операторы, передаваемые в строковой переменной.

34.5.1. Выполнение операторов без набора результатов

Самый простой способ выполнить произвольный SQL-оператор — применить команду `EXECUTE IMMEDIATE`. Например:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

`EXECUTE IMMEDIATE` можно применять для SQL-операторов, которые не возвращают набор результатов (например, DDL, `INSERT`, `UPDATE`, `DELETE`). Выполнять операторы, которые получают данные, (например, `SELECT`) таким образом нельзя. Как выполнять такие операторы, рассказывается в следующем разделе.

34.5.2. Выполнение оператора с входными параметрами

Более эффективно выполнять произвольный SQL-оператор можно, подготовив его один раз, а затем запуская подготовленный оператор столько, сколько нужно. Также можно подготовить обобщённую версию оператора, а затем выполнять специализированные его версии, подставляя в него параметры. Подготавливая оператор, поставьте знаки вопроса там, где позже хотите подставить параметры. Например:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
```

```
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

Когда подготовленный оператор больше не нужен, его следует освободить:

```
EXEC SQL DEALLOCATE PREPARE имя;
```

34.5.3. Выполнение оператора с набором результатов

Для выполнения SQL-оператора с одной строкой результата можно применить команду `EXECUTE`. Чтобы сохранить результат, добавьте предложение `INTO`.

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

Команда `EXECUTE` может содержать предложение `INTO` и/или предложение `USING`, либо не содержать ни того, ни другого.

Если ожидается, что запрос вернёт более одной строки результата, следует применять курсор, как показано в следующем примере. (Подробно курсоры описываются в Подразделе 34.3.2.)

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.usename as dbaname, d.datname "
             "  FROM pg_database d, pg_user u "
             "  WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

34.6. Библиотека pgtypes

Библиотека pgtypes сопоставляет типы базы данных Postgres Pro с их эквивалентами в C, которые можно использовать в программах на C. Она также предлагает функции для выполнения простых вычислений с этими типами в C, то есть без помощи сервера Postgres Pro. Рассмотрите следующий пример:

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
PGTYPESchar_free(out);
```

34.6.1. Символьные строки

Некоторые функции, в частности `PGTYPESnumeric_to_asc`, возвращают указатель на строку в выделенной для неё памяти. Их результаты должны освобождаться функцией `PGTYPESchar_free`, а не `free`. (Это важно только в Windows, где выделение и освобождение памяти в определённых случаях должно производиться одной библиотекой.)

34.6.2. Тип numeric

Тип numeric позволяет производить вычисления с произвольной точностью. Эквивалентный ему тип на сервере Postgres Pro описан в Разделе 8.1. Ввиду того, что переменная имеет произвольную точность, она должна расширяться и сжиматься динамически. Поэтому такие переменные можно создавать только в области кучи, используя функции `PGTYPESnumeric_new` и `PGTYPESnumeric_free`. Тип decimal подобен numeric, но имеет ограниченную точность, и поэтому может размещаться и в области кучи, и в стеке.

Для работы с типом numeric можно использовать следующие функции:

`PGTYPESnumeric_new`

Запрашивает указатель на новую переменную, размещённую в памяти.

```
numeric *PGTYPESnumeric_new(void);
```

`PGTYPESnumeric_free`

Освобождает переменную типа numeric, высвобождая всю её память.

```
void PGTYPESnumeric_free(numeric *var);
```

`PGTYPESnumeric_from_asc`

Разбирает числовой тип из строковой записи.

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

Допускаются в частности следующие форматы: `-2`, `.794`, `+3.44`, `592.49E07` и `-32.84e-4`. Если значение удаётся разобрать успешно, возвращается действительный указатель, в противном случае указатель NULL. На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в `*endptr`. Поэтому в `endptr` свободно можно передать NULL.

`PGTYPESnumeric_to_asc`

Возвращает указатель на строку, выделенную функцией `malloc` и содержащую строковое представление значения `num` числового типа.

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

Числовое значение будет выводиться с заданным в `dscale` количеством цифр после запятой, округлённое при необходимости. Результат нужно освободить функцией `PGTYPESchar_free()`.

`PGTYPESnumeric_add`

Суммирует две числовые переменные и возвращает результат в третьей.

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

Эта функция суммирует переменные `var1` и `var2` в результирующую переменную `result`. Функция возвращает 0 в случае успеха и -1 при ошибке.

`PGTYPESnumeric_sub`

Вычисляет разность двух числовых переменных и возвращает результат в третьей.

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

Эта функция вычитает переменную `var2` из `var1`. Результат операции помещается в переменную `result`. Функция возвращает 0 в случае успеха и -1 при ошибке.

`PGTYPESnumeric_mul`

Перемножает две числовые переменные и возвращает результат в третьей.

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

Эта функция перемножает переменные `var1` и `var2`. Результат операции сохраняется в переменной `result`. Функция возвращает 0 в случае успеха и -1 при ошибке.

`PGTYPESnumeric_div`

Вычисляет частное двух числовых переменных и возвращает результат в третьей.

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

Эта функция делит переменную `var1` на `var2`. Результат операции сохраняется в переменной `result`. Функция возвращает 0 в случае успеха и -1 при ошибке.

`PGTYPESnumeric_cmp`

Сравнивает две числовые переменные.

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

Эта функция производит сравнение двух числовых переменных. При ошибке возвращается `INT_MAX`. В случае успеха функция возвращает одно из трёх возможных значений:

- 1, если `var1` больше `var2`
- -1, если `var1` меньше `var2`
- 0, если `var1` и `var2` равны

`PGTYPESnumeric_from_int`

Преобразует переменную int в переменную numeric.

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

Эта функция принимает целочисленную переменную со знаком типа signed int и сохраняет её значение в переменной `var` типа numeric. Функция возвращает 0 в случае успеха и -1 при ошибке.

`PGTYPESnumeric_from_long`

Преобразует переменную long int в переменную numeric.

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

Эта функция принимает целочисленную переменную со знаком типа signed long int и сохраняет её значение в переменной `var` типа numeric. Функция возвращает 0 в случае успеха и -1 при ошибке.

`PGTYPESnumeric_copy`

Копирует одну числовую переменную в другую.

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

Эта функция копирует значение переменной, на которую указывает `src`, в переменную, на которую указывает `dst`. Она возвращает 0 в случае успеха и -1 при ошибке.

`PGTYPESnumeric_from_double`

Преобразует переменную типа double в переменную numeric.

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

Эта функция принимает переменную типа double и сохраняет преобразованное значение в переменной, на которую указывает `dst`. Она возвращает 0 в случае успеха и -1 при ошибке.

`PGTYPESnumeric_to_double`

Преобразует переменную типа numeric в переменную double.

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

Эта функция преобразует значение типа numeric переменной, на которую указывает `nv`, в переменную типа double, на которую указывает `dp`. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной `errno` дополнительно устанавливается значение `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_to_int`

Преобразует переменную типа numeric в переменную int.

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает `nv`, в целочисленную переменную, на которую указывает `ip`. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной `errno` дополнительно устанавливается значение `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_to_long`

Преобразует переменную типа numeric в переменную long.

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает `nv`, в целочисленную переменную типа long, на которую указывает `lp`. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении или антипереполнении. Если происходит переполнение, в глобальной переменной `errno` дополнительно устанавливается значение `PGTYPES_NUM_OVERFLOW`, а если антипереполнение — `PGTYPES_NUM_UNDERFLOW`.

`PGTYPESnumeric_to_decimal`

Преобразует переменную типа numeric в переменную decimal.

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

Эта функция преобразует значение типа numeric переменной, на которую указывает `src`, в переменную типа decimal, на которую указывает `dst`. Она возвращает 0 в случае успеха и -1 при ошибке, в том числе при переполнении. Если происходит переполнение, в глобальной переменной `errno` дополнительно устанавливается значение `PGTYPES_NUM_OVERFLOW`.

`PGTYPESnumeric_from_decimal`

Преобразует переменную типа decimal в переменную numeric.

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

Эта функция преобразует значение типа decimal переменной, на которую указывает `src`, в переменную типа numeric, на которую указывает `dst`. Она возвращает 0 в случае успеха и -1 при ошибке. Так как тип decimal реализован как ограниченная версия типа numeric, при таком преобразовании переполнение невозможно.

34.6.3. Тип date

Тип date, реализованный в C, позволяет программам работать с данными типа date в SQL. Соответствующий тип сервера Postgres Pro описан в Разделе 8.5.

Для работы с типом date можно использовать следующие функции:

`PGTYPESdate_from_timestamp`

Извлекает часть даты из значения типа timestamp.

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

Эта функция получает в единственном аргументе значение времени типа timestamp и возвращает извлечённую из него дату.

`PGTYPESdate_from_asc`

Разбирает дату из её текстового представления.

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

Эта функция получает строку C char* `str` и указатель на строку C char* `endptr`. На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в `*endptr`. Поэтому в `endptr` свободно можно передать NULL.

Заметьте, что эта функция всегда подразумевает формат дат MDY (месяц-день-год) и никакой переменной для изменения этого формата в ECPG нет.

Все допустимые форматы ввода перечислены в Таблице 34.2.

Таблица 34.2. Допустимые форматы ввода для `PGTYPESdate_from_asc`

Ввод	Результат
`January 8, 1999`	`January 8, 1999`
`1999-01-08`	`January 8, 1999`
`1/8/1999`	`January 8, 1999`
`1/18/1999`	`January 18, 1999`
`01/02/03`	`February 1, 2003`
`1999-Jan-08`	`January 8, 1999`
`Jan-08-1999`	`January 8, 1999`
`08-Jan-1999`	`January 8, 1999`
`99-Jan-08`	`January 8, 1999`
`08-Jan-99`	`January 8, 1999`
`08-Jan-06`	`January 8, 2006`
`Jan-08-99`	`January 8, 1999`
`19990108`	`ISO 8601; January 8, 1999`

Ввод	Результат
`990108`	`ISO 8601; January 8, 1999`
`1999.008`	`год и день года`
`J2451187`	`Юлианский день`
`January 8, 99 BC`	`99 год до нашей эры`

`PGTYPESdate_to_asc`

Возвращает текстовое представление переменной типа date.

```
char *PGTYPESdate_to_asc(date dDate);
```

Эта функция получает в качестве единственного параметра дату `dDate` и выводит её в виде `1999-01-18`, то есть в формате `YYYY-MM-DD`. Результат необходимо освободить функцией `PGTYPESchar_free()`.

`PGTYPESdate_julmdy`

Извлекает значения дня, месяца и года из переменной типа date.

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

Эта функция получает дату `d` и указатель на 3 целочисленных значения `mdy`. Имя переменной указывает на порядок значений: в `mdy[0]` записывается номер месяца, в `mdy[1]` — номер дня, а в `mdy[2]` — год.

`PGTYPESdate_mdyjul`

Образует значение даты из массива 3 целых чисел, задающих день, месяц и год даты.

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

Эта функция получает в первом аргументе массив из 3 целых чисел (`mdy`), а во втором указатель на переменную типа date, в которую будет помещён результат операции.

`PGTYPESdate_dayofweek`

Возвращает число, представляющее день недели для заданной даты.

```
int PGTYPESdate_dayofweek(date d);
```

Эта функция принимает в единственном аргументе переменную `d` типа date и возвращает целое число, выражающее день недели для этой даты.

- 0 — Воскресенье
- 1 — Понедельник
- 2 — Вторник
- 3 — Среда
- 4 — Четверг
- 5 — Пятница
- 6 — Суббота

`PGTYPESdate_today`

Выдаёт текущую дату.

```
void PGTYPESdate_today(date *d);
```

Эта функция получает указатель на переменную (`d`) типа date, в которую будет записана текущая дата.

`PGTYPESdate_fmt_asc`

Преобразует переменную типа date в текстовое представление по маске формата.

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

Эта функция принимает дату для преобразования (`dDate`), маску формата (`fmtstring`) и строку, в которую будет помещено текстовое представление даты (`outbuf`).

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

В строке формата можно использовать следующие коды полей:

- `dd` — Номер дня в месяце.
- `mm` — Номер месяца в году.
- `yy` — Номер года в виде двух цифр.
- `yyyy` — Номер года в виде четырёх цифр.
- `ddd` — Название дня недели (сокращённое).
- `mmm` — Название месяца (сокращённое).

Все другие символы копируются в выводимую строку 1:1.

В Таблице 34.3 перечислены несколько возможных форматов. Это даёт представление, как можно использовать эту функцию. Все строки вывода даны для одной даты: 23 ноября 1959 г.

Таблица 34.3. Допустимые форматы ввода для `PGTYPESdate_fmt_asc`

Формат	**Результат**
`mmddyy`	`112359`
`ddmmyy`	`231159`
`yymmdd`	`591123`
`yy/mm/dd`	`59/11/23`
`yy mm dd`	`59 11 23`
`yy.mm.dd`	`59.11.23`
`.mm.yyyy.dd.`	`.11.1959.23.`
`mmm. dd, yyyy`	`Nov. 23, 1959`
`mmm dd yyyy`	`Nov 23 1959`
`yyyy dd mm`	`1959 23 11`
`ddd, mmm. dd, yyyy`	`Mon, Nov. 23, 1959`
`(ddd) mmm. dd, yyyy`	`(Mon) Nov. 23, 1959`

`PGTYPESdate_defmt_asc`

Преобразует строку C `char*` в значение типа date по маске формата.

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

Эта функция принимает указатель на переменную типа date (`d`), в которую будет помещён результат операции, маску формата для разбора даты (`fmt`) и строку C char*, содержащую текстовое представление даты (`str`). Ожидается, что текстовое представление будет соответствовать маске формата. Однако это соответствие не обязательно должно быть точным. Данная функция анализирует только порядок элементов и ищет в нём подстроки `yy` или `yyyy`, обозначающие позицию года, подстроку `mm`, обозначающую позицию месяца, и `dd`, обозначающую позицию дня.

В Таблица 34.4 перечислены несколько возможных форматов. Это даёт представление, как можно использовать эту функцию.

Таблица 34.4. Допустимые форматы ввода для `rdefmtdate`

Формат	Строка	Результат
`ddmmyy`	`21-2-54`	`1954-02-21`
`ddmmyy`	`2-12-54`	`1954-12-02`
`ddmmyy`	`20111954`	`1954-11-20`
`ddmmyy`	`130464`	`1964-04-13`
`mmm.dd.yyyy`	`MAR-12-1967`	`1967-03-12`
`yy/mm/dd`	`1954, February 3rd`	`1954-02-03`
`mmm.dd.yyyy`	`041269`	`1969-04-12`
`yy/mm/dd`	`In the year 2525, in the month of July, mankind will be alive on the 28th day`	`2525-07-28`
`dd-mm-yy`	`I said on the 28th of July in the year 2525`	`2525-07-28`
`mmm.dd.yyyy`	`9/14/58`	`1958-09-14`
`yy/mm/dd`	`47/03/29`	`1947-03-29`
`mmm.dd.yyyy`	`oct 28 1975`	`1975-10-28`
`mmddyy`	`Nov 14th, 1985`	`1985-11-14`

34.6.4. Тип timestamp

Тип timestamp, реализованный в C, позволяет программам работать с данными типа timestamp в SQL. Соответствующий тип сервера Postgres Pro описан в Разделе 8.5.

Для работы с типом timestamp можно использовать следующие функции:

`PGTYPEStimestamp_from_asc`

Разбирает значение даты/времени из текстового представления в переменную типа timestamp.

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

Эта функция получает строку (`str`), которую нужно разобрать, и указатель на строку C char* (`endptr`). На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в `*endptr`. Поэтому в `endptr` свободно можно передать NULL.

В случае успеха эта функция возвращает разобранное время, а в случае ошибки возвращается `PGTYPESInvalidTimestamp` и в `errno` устанавливается значение `PGTYPES_TS_BAD_TIMESTAMP`. См. замечание относительно `PGTYPESInvalidTimestamp`.

Вообще вводимая строка может содержать допустимое указание даты, пробельные символы и допустимое указание времени в любом сочетании. Заметьте, что часовые пояса ECPG не поддерживает. Эта функция может разобрать их, но не задействует их в вычислениях как это делает, например, сервер Postgres Pro. Указания часового пояса во вводимой строке просто игнорируются.

В Таблица 34.5 приведены несколько примеров вводимых строк.

Таблица 34.5. Допустимые форматы ввода для `PGTYPEStimestamp_from_asc`

Ввод	Результат
`1999-01-08 04:05:06`	`1999-01-08 04:05:06`

Ввод	**Результат**
`January 8 04:05:06 1999 PST`	`1999-01-08 04:05:06`
`1999-Jan-08 04:05:06.789-8`	`1999-01-08 04:05:06.789 (указание часового пояса игнорируется)`
`J2451187 04:05-08:00`	`1999-01-08 04:05:00 (указание часового пояса игнорируется)`

`PGTYPEStimestamp_to_asc`

Преобразует значение даты в строку C char*.

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

Эта функция принимает в качестве единственного аргумента `tstamp` значение типа timestamp и возвращает размещённую в памяти строку, содержащую текстовое представление даты/времени. Результат необходимо освободить функцией `PGTYPESchar_free()`.

`PGTYPEStimestamp_current`

Получает текущее время.

```
void PGTYPEStimestamp_current(timestamp *ts);
```

Эта функция получает текущее время и сохраняет его в переменной типа timestamp, на которую указывает `ts`.

`PGTYPEStimestamp_fmt_asc`

Преобразует переменную типа timestamp в строку C char* по маске формата.

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char
 *fmtstr);
```

Эта функция получает в первом аргументе (`ts`) указатель на переменную типа timestamp, а в последующих указатель на буфер вывода (`output`), максимальную длину строки, которую может принять буфер (`str_len`), и маску формата, с которой будет выполняться преобразование (`fmtstr`).

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

В маске формата можно использовать коды формата, перечисленные ниже. Эти же коды принимает функция `strftime` из библиотеки libc. Любые символы, не относящиеся к кодам формата, будут просто скопированы в буфер вывода.

- `%A` — заменяется локализованным представлением полного названия дня недели.
- `%a` — заменяется локализованным представлением сокращённого названия дня недели.
- `%B` — заменяется локализованным представлением полного названия месяца.
- `%b` — заменяется локализованным представлением сокращённого названия месяца.
- `%C` — заменяется столетием (год / 100) в виде десятичного числа; одиночная цифра предваряется нулём.
- `%c` — заменяется локализованным представлением даты и времени.
- `%D` — равнозначно `%m/%d/%y`.
- `%d` — заменяется днём месяца в виде десятичного числа (01–31).
- `%E*` `%O*` — расширения локали POSIX. Последовательности `%Ec` `%EC` `%Ex` `%EX` `%Ey` `%EY` `%Od` `%Oe` `%OH` `%OI` `%Om` `%OM` `%OS` `%Ou` `%OU` `%OV` `%Ow` `%OW` `%Oy` должны выводить альтернативные представления.

  Кроме того, альтернативные названия месяцев представляет код формата `%OB` (используется отдельно, без упоминания дня).

- `%e` — заменяется днём в виде десятичного числа (1–31); одиночная цифра предваряется пробелом.
- `%F` — равнозначно `%Y-%m-%d`.
- `%G` — заменяется годом в виде десятичного числа (со столетием). При этом годом считается тот, что содержит наибольшую часть недели (дни недели начинаются с понедельника).
- `%g` — заменяется тем же годом, что и `%G`, но в виде десятичного числа без столетия (00–99).
- `%H` — заменяется часами (в 24-часовом формате) в виде десятичного числа (00–23).
- `%h` — равнозначно `%b`.
- `%I` — заменяется часами (в 12-часовом формате) в виде десятичного числа (01–12).
- `%j` — заменяется днём года в виде десятичного числа (001–366).
- `%k` — заменяется часами (в 24-часовом формате) в виде десятичного числа (0–23); одиночная цифра предваряется пробелом.
- `%l` — заменяется часами (в 12-часовом формате) в виде десятичного числа (1–12); одиночная цифра предваряется пробелом.
- `%M` — заменяется минутами в виде десятичного числа (00–59).
- `%m` — заменяется номером месяца в виде десятичного числа (01–12).
- `%n` — заменяется символом новой строки.
- `%O*` — равнозначно `%E*`.
- `%p` — заменяется локализованным представлением «до полудня» или «после полудня» в зависимости от времени.
- `%R` — равнозначно `%H:%M`.
- `%r` — равнозначно `%I:%M:%S %p`.
- `%S` — заменяется секундами в виде десятичного числа (00–60).
- `%s` — заменяется числом секунд с начала эпохи, по мировому времени (UTC).
- `%T` — равнозначно `%H:%M:%S`
- `%t` — заменяется символом табуляции.
- `%U` — заменяется номером недели в году (первым днём недели считается воскресенье) в виде десятичного числа (00–53).
- `%u` — заменяется номером дня недели (первым днём недели считается понедельник) в виде десятичного числа (1–7).
- `%V` — заменяется номером недели в году (первым днём недели считается понедельник) в виде десятичного числа (01–53). Если к недели, включающей 1 января, относятся 4 или больше дней нового года, она считается неделей с номером 1; в противном случае это последняя неделя предыдущего года, а неделей под номером 1 будет следующая.
- `%v` — равнозначно `%e-%b-%Y`.
- `%W` — заменяется номером недели в году (первым днём недели считается понедельник) в виде десятичного числа (00–53).
- `%w` — заменяется номером дня недели (первым днём недели считается воскресенье) в виде десятичного числа (0–6).
- `%X` — заменяется локализованным представлением времени.
- `%x` — заменяется локализованным представлением даты.
- `%Y` — заменяется годом со столетием в виде десятичного числа.

- `%y` — заменяется годом без столетия в виде десятичного числа (00–99).
- `%Z` — заменяется названием часового пояса.
- `%z` — заменяется смещением часового пояса от UTC; ведущий знак плюс обозначает смещение к востоку от UTC, а знак минус — к западу, часы и минуты задаются парами цифр без разделителя между ними (эта форма установлена для даты в *RFC 822*).
- `%+` — заменяется локализованным представлением даты и времени.
- `%-*` — расширение GNU libc. Отключает дополнение чисел по ширине при выводе.
- $_* — расширение GNU libc. Явно включает дополнение пробелами.
- `%0*` — расширение GNU libc. Явно включает дополнение нулями.
- `%%` — заменяется символом `%`.

`PGTYPEStimestamp_sub`

Вычитает одно значение времени из другого и сохраняет результат в переменной типа interval.

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

Эта функция вычитает значение типа timestamp, на которое указывает `ts2`, из значения timestamp, на которое указывает `ts1`, и сохраняет результат в переменной типа interval, на которую указывает `iv`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`PGTYPEStimestamp_defmt_asc`

Разбирает значение типа timestamp из текстового представления с заданной маской формата.

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

Эта функция получает текстовое представление даты/времени в переменной `str`, а также маску формата для разбора в переменной `fmt`. Результат будет сохранён в переменной, на которую указывает `d`.

Если вместо маски формата `fmt` передаётся NULL, эта функция переходит к стандартной маске форматирования, а именно: `%Y-%m-%d %H:%M:%S`.

Данная функция является обратной к функции `PGTYPEStimestamp_fmt_asc`. Обратитесь к её документации, чтобы узнать о возможных вариантах маски формата.

`PGTYPEStimestamp_add_interval`

Добавляет переменную типа interval к переменной типа timestamp.

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

Эта функция получает указатель на переменную `tin` типа timestamp и указатель на переменную `span` типа interval. Она добавляет временной интервал к значению даты/времени и сохраняет полученную дату/время в переменной типа timestamp, на которую указывает `tout`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`PGTYPEStimestamp_sub_interval`

Вычитает переменную типа interval из переменной типа timestamp.

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

Эта функция вычитает значение типа interval, на которое указывает `span`, из значения типа timestamp, на которое указывает `tin`, и сохраняет результат в переменной, на которую указывает `tout`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

34.6.5. Тип interval

Тип interval, реализованный в C, позволяет программам работать с данными типа interval в SQL. Соответствующий тип сервера Postgres Pro описан в Разделе 8.5.

Для работы с типом interval можно использовать следующие функции:

`PGTYPESinterval_new`

Возвращает указатель на новую переменную interval, размещённую в памяти.

```
interval *PGTYPESinterval_new(void);
```

`PGTYPESinterval_free`

Освобождает место, занимаемое ранее размещённой в памяти переменной типа interval.

```
void PGTYPESinterval_free(interval *intvl);
```

`PGTYPESinterval_from_asc`

Разбирает значение типа interval из его текстового представления.

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

Эта функция разбирает входную строку `str` и возвращает указатель на размещённую в памяти переменную типа interval. На данный момент ECPG всегда разбирает строку до конца, так что эта функция не может вернуть адрес первого недопустимого символа в `*endptr`. Поэтому в `endptr` свободно можно передать NULL.

`PGTYPESinterval_to_asc`

Преобразует переменную типа interval в текстовое представление.

```
char *PGTYPESinterval_to_asc(interval *span);
```

Эта функция преобразует переменную типа interval, на которую указывает `span`, в строку C char*. Её вывод выглядит примерно так: `@ 1 day 12 hours 59 mins 10 secs`. Результат необходимо освободить функцией `PGTYPESchar_free()`.

`PGTYPESinterval_copy`

Копирует переменную типа interval.

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

Эта функция копирует переменную типа interval, на которую указывает `intvlsrc`, в переменную, на которую указывает `intvldest`. Заметьте, что для целевой переменной необходимо предварительно выделить память.

34.6.6. Тип decimal

Тип decimal похож на тип numeric, однако его максимальная точность ограничена 30 значащими цифрами. В отличие от типа numeric, который можно создать только в области кучи, тип decimal можно создать и в стеке, и в области кучи (посредством функций `PGTYPESdecimal_new` и `PGTYPESdecimal_free`). Для работы с типом decimal есть много других функций, подключаемых в режиме совместимости с Informix, описанном в Разделе 34.15.

Для работы с типом decimal можно использовать следующие функции (содержащиеся не в библиотеке `libcompat`).

`PGTYPESdecimal_new`

Запрашивает указатель на новую переменную decimal, размещённую в памяти.

```
decimal *PGTYPESdecimal_new(void);
```

`PGTYPESdecimal_free`

Освобождает переменную типа decimal, высвобождая всю её память.

```
void PGTYPESdecimal_free(decimal *var);
```

34.6.7. Значения errno, которые устанавливает pgtypeslib

`PGTYPES_NUM_BAD_NUMERIC`

Аргумент должен содержать переменную типа numeric (либо указывать на переменную типа numeric), но представление этого типа в памяти оказалось некорректным.

`PGTYPES_NUM_OVERFLOW`

Произошло переполнение. Так как тип numeric может принимать значения практически любой точности, при преобразовании этого типа в другие типы возможно переполнение.

`PGTYPES_NUM_UNDERFLOW`

Произошло антипереполнение. Так как тип numeric может принимать значения практически любой точности, при преобразовании переменной этого типа в другие типы возможно антипереполнение.

`PGTYPES_NUM_DIVIDE_ZERO`

Имела место попытка деления на ноль.

`PGTYPES_DATE_BAD_DATE`

Функции `PGTYPESdate_from_asc` передана некорректная строка даты.

`PGTYPES_DATE_ERR_EARGS`

Функции `PGTYPESdate_defmt_asc` переданы некорректные аргументы.

`PGTYPES_DATE_ERR_ENOSHORTDATE`

В строке, переданной функции `PGTYPESdate_defmt_asc`, оказался неправильный компонент даты.

`PGTYPES_INTVL_BAD_INTERVAL`

Функции `PGTYPESinterval_from_asc` передана некорректная строка, задающая интервал, либо функции `PGTYPESinterval_to_asc` передано некорректное значение интервала.

`PGTYPES_DATE_ERR_ENOTDMY`

Обнаружено несоответствие при выводе компонентов день/месяц/год в функции `PGTYPESdate_defmt_asc`.

`PGTYPES_DATE_BAD_DAY`

Функция `PGTYPESdate_defmt_asc` обнаружила некорректное значение дня месяца.

`PGTYPES_DATE_BAD_MONTH`

Функция `PGTYPESdate_defmt_asc` обнаружила некорректное значение месяца.

`PGTYPES_TS_BAD_TIMESTAMP`

Функции `PGTYPEStimestamp_from_asc` передана некорректная строка даты/времени, либо функции `PGTYPEStimestamp_to_asc` передано некорректное значение типа timestamp.

`PGTYPES_TS_ERR_EINFTIME`

Значение типа timestamp, представляющее бесконечность, получено в недопустимом контексте.

34.6.8. Специальные константы pgtypeslib

`PGTYPESInvalidTimestamp`

Значение типа timestamp, представляющее недопустимое время. Это значение возвращает функция `PGTYPEStimestamp_from_asc` при ошибке разбора. Заметьте, что вследствие особенности внутреннего представления типа `timestamp`, значение `PGTYPESInvalidTimestamp` в то же время представляет корректное время (`1899-12-31 23:59:59`). Поэтому для выявления ошибок необходимо, чтобы приложение не только сравнивало результат функции с `PGTYPESInvalidTimestamp`, но и проверяло условие `errno != 0` после каждого вызова `PGTYPEStimestamp_from_asc`.

34.7. Использование областей дескрипторов

Области дескрипторов SQL дают возможности для более сложной обработки результатов операторов `SELECT`, `FETCH` и `DESCRIBE`. Область дескриптора SQL объединяет в одной структуре данные одной строки и элементы метаданных. Эти метаданные особенно полезны при выполнении динамических SQL-операторов, когда характер результирующих столбцов может быть неизвестен заранее. Postgres Pro предлагает два подхода к использованию областей дескрипторов: именованные области SQL-дескрипторов и области SQLDA в структурах C.

34.7.1. Именованные области SQL-дескрипторов

Именованная область SQL-дескриптора состоит из заголовка, содержащего сведения обо всём дескрипторе, и одного или нескольких дескрипторов элементов, которые по сути описывают отдельные столбцы в строке результата.

Прежде чем вы сможете использовать область SQL-дескриптора, её нужно выделить:

```
EXEC SQL ALLOCATE DESCRIPTOR идентификатор;
```

Заданный идентификатор играет роль «имени переменной» области дескриптора. Когда дескриптор оказывается ненужным, его следует освободить:

```
EXEC SQL DEALLOCATE DESCRIPTOR идентификатор;
```

Чтобы воспользоваться областью дескриптора, её нужно указать в качестве целевого объекта в предложении `INTO`, вместо перечисления переменных среды:

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

Если набор результатов пуст, в области дескриптора будут тем не менее содержаться метаданные из запроса, то есть имена полей.

Получить метаданные набора результатов для ещё не выполненных подготовленных запросов можно, воспользовавшись оператором `DESCRIBE`:

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

До PostgreSQL версии 9.0 ключевое слово `SQL` было необязательным, так что предложения `DESCRIPTOR` и `SQL DESCRIPTOR` создавали именованные области SQL-дескрипторов. Теперь оно стало обязательным; без слова `SQL` создаются области SQLDA, см. Подраздел 34.7.2.

В операторах `DESCRIBE` и `FETCH` ключевые слова `INTO` и `USING` действуют примерно одинаково: они указывают вывести набор результатов и метаданные в область дескриптора.

Возникает вопрос: а как же получить данные из области дескриптора? Область дескриптора можно представить как структуру с именованными полями. Чтобы получить значение поля из заголовка и сохранить его в переменной среды C, нужно выполнить команду:

```
EXEC SQL GET DESCRIPTOR имя :переменная_среды = поле;
```

В настоящее время определено только одно поле заголовка: *`COUNT`*, которое говорит, сколько областей дескрипторов элементов существует (то есть, сколько столбцов содержится в результате). Переменная среды C должна иметь целочисленный тип. Чтобы получить поле из области дескриптора элемента, нужно выполнить команду:

```
EXEC SQL GET DESCRIPTOR имя VALUE номер :переменная_среды = поле;
```

В качестве *`num`* можно задать обычное целое или переменную среды C, содержащую целое число. Допустимые поля:

`CARDINALITY` (integer)

: число строк в наборе результатов

`DATA`

: собственно элемент данных (тип данных поля зависит от запроса)

`DATETIME_INTERVAL_CODE` (целое)

: Когда `TYPE` равно `9`, `DATETIME_INTERVAL_CODE` содержит значение `1` для `DATE`, `2` для `TIME`, `3` для `TIMESTAMP`, `4` для `TIME WITH TIME ZONE`, либо `5` для `TIMESTAMP WITH TIME ZONE`.

`DATETIME_INTERVAL_PRECISION` (целое)

: не реализовано

`INDICATOR` (целое)

: индикатор (отмечающий значение NULL или усечение значения)

`KEY_MEMBER` (целое)

: не реализовано

`LENGTH` (целое)

: длина данных в символах

`NAME` (строка)

: имя столбца

`NULLABLE` (целое)

: не реализовано

`OCTET_LENGTH` (целое)

: длина символьного представления данных в байтах

`PRECISION` (целое)

: точность (для типа `numeric`)

`RETURNED_LENGTH` (целое)

: длина данных в символах

`RETURNED_OCTET_LENGTH` (целое)

длина символьного представления данных в байтах

`SCALE` (целое)

масштаб (для типа `numeric`)

`TYPE` (целое)

числовой код типа данных столбца

В операторах `EXECUTE`, `DECLARE` и `OPEN` ключевые слова `INTO` и `USING` действуют по-разному. Область дескриптора также можно сформировать вручную, чтобы передать входные параметры запросу или курсору, а команда `USING SQL DESCRIPTOR` *имя* даёт возможность передать входные аргументы параметризованному запросу. Оператор, формирующий именованную область SQL-дескриптора, выглядит так:

```
EXEC SQL SET DESCRIPTOR имя VALUE номер поле = :переменная_среды;
```

Postgres Pro поддерживает выборку сразу нескольких записей в одном операторе `FETCH` и может сохранить их данные в переменной среды C, если эта переменная — массив. Например:

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

34.7.2. Области дескрипторов SQLDA

Область дескриптора SQLDA представляет собой структуру языка C, в которую можно получить набор результатов и метаданные запроса. Одна такая структура содержит одну запись из набора данных.

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

Заметьте, что ключевое слово `SQL` в этом случае опускается. Замечания относительно применения ключевых слов `INTO` и `USING` в Подразделе 34.7.1 применимы и здесь, с дополнением. В операторе `DESCRIBE` можно полностью опустить ключевое слово `DESCRIPTOR`, если присутствует ключевое слово `INTO`:

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

Общая схема использования SQLDA выглядит так:

1. Подготовить запрос и объявить курсор для него.
2. Объявить SQLDA для строк результата.
3. Объявить SQLDA для входных параметров и инициализировать их (выделить память, задать параметры).
4. Открыть курсор с входной SQLDA.
5. Выбрать строки из курсора и сохранить их в выходной SQLDA.
6. Прочитать значения из выходной SQLDA в переменные среды (и преобразовать при необходимости).
7. Закрыть курсор.

8. Освободить область памяти, выделенную для входной SQLDA.

34.7.2.1. Структура данных SQLDA

Для SQLDA используются три типа данных: `sqlda_t`, `sqlvar_t` и `struct sqlname`.

> **Подсказка**
>
> Структура данных SQLDA в Postgres Pro подобна той, что используется в IBM DB2 Universal Database, так что часть технической информации по SQLDA в DB2 может быть полезна и для понимания устройства SQLDA в Postgres Pro.

34.7.2.1.1. Структура sqlda_t

Тип структуры `sqlda_t` представляет тип собственно SQLDA. Эта структура описывает одну запись. Две или более структур `sqlda_t` могут объединяться в связанный список по указателям в поле `desc_next`, и таким образом образовывать упорядоченный набор строк. Поэтому, когда выбираются две или более строк, приложение может прочитать их, проследуя по указателям `desc_next` во всех узлах `sqlda_t`.

Тип `sqlda_t` определяется так:

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;
```

Его поля имеют следующее назначение:

`sqldaid`

Содержит строковую константу "`SQLDA  `".

`sqldabc`

Содержит размер выделенного пространства в байтах.

`sqln`

Содержит число входных параметров для параметризованного запроса, когда передаётся в операторы `OPEN`, `DECLARE` или `EXECUTE` с ключевым словом `USING`. В структуре, выводимой операторами `SELECT`, `EXECUTE` или `FETCH`, данное значение совпадает с `sqld`.

`sqld`

Содержит число полей в наборе результатов.

`desc_next`

Если запрос выдаёт несколько записей, возвращается несколько связанных структур SQLDA, а `desc_next` содержит указатель на следующую запись в списке.

`sqlvar`

Это массив столбцов в наборе результатов.

34.7.2.1.2. Структура sqlvar_t

Тип структуры `sqlvar_t` содержит значение столбца и метаданные, в частности, тип и длину. Эта структура определяется так:

```
struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

Её поля имеют следующее назначение:

`sqltype`

Содержит идентификатор типа данного поля. Возможные значения перечислены в `enum ECPGttype` в `ecpgtype.h`.

`sqllen`

Содержит двоичную длину поля, например 4 байта для `ECPGt_int`.

`sqldata`

Указывает на данные. Формат данных описан в Подразделе 34.4.4.

`sqlind`

Указывает на индикатор NULL. 0 соответствует значению не NULL, -1 — NULL.

`sqlname`

Имя поля.

34.7.2.1.3. Структура struct sqlname

Структура `struct sqlname` содержит имя столбца. Она включена в `sqlvar_t` в качестве члена. Эта структура определена так:

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

Её поля имеют следующее назначение:

`length`

Содержит длину имени поля.

`data`

Содержит собственно имя поля.

34.7.2.2. Получение набора результатов с применением SQLDA

Чтобы получить набор результатов запроса через SQLDA, нужно проделать примерно следующее:

1. Объявить структуру `sqlda_t` для получения набора результатов.

2. Выполнить команды `FETCH`/`EXECUTE`/`DESCRIBE` для обработки запроса с указанной SQLDA.
3. Определить число записей в наборе результатов, прочитав `sqln`, член структуры `sqlda_t`.
4. Получить значения каждого столбца из элементов `sqlvar[0]`, `sqlvar[1]` и т. д., составляющих массив, включённый в структуру `sqlda_t`.
5. Перейти к следующей строке (структуре `sqlda_t`) по указателю `desc_next`, члену структуры `sqlda_t`.
6. При необходимости повторить эти действия.

Далее показывается, как получить набор результатов через SQLDA.

Сначала объявите структуру `sqlda_t`, в которую будет помещён набор результатов.

```
sqlda_t *sqlda1;
```

Затем укажите эту SQLDA в команде. В данном примере это команда `FETCH`.

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

Обработайте все строки в цикле с переходом по связанному списку.

```
sqlda_t *cur_sqlda;

for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}
```

Внутри этого цикла реализуйте ещё один цикл чтения данных каждого столбца (структуры `sqlvar_t`) в строке.

```
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen  = v.sqllen;
    ...
}
```

Чтобы получить значение столбца, прочитайте значение поля `sqltype`, члена структуры `sqlvar_t`. Затем выберите подходящий способ, в зависимости от типа столбца, копирования данных из поля `sqlvar` в переменную среды C.

```
char var_buf[1024];

switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) - 1 :
 sqllen));
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;
```

```
    ...
}
```

34.7.2.3. Передача значений параметров через SQLDA

Чтобы передать параметры подготовленному запросу через SQLDA, нужно проделать примерно следующее:

1. Создать подготовленный запрос (подготовленный оператор)
2. Объявить структуру sqlda_t в качестве входной SQLDA.
3. Выделить область памяти (структуру sqlda_t) для входной SQLDA.
4. Установить (скопировать) входные значения в выделенной памяти.
5. Открыть курсор, указав входную SQLDA.

Рассмотрим это на примере.

Сначала создайте подготовленный оператор.

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE d.oid
 = s.datid AND (d.datname = ? OR d.oid = ?)";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :query;
```

Затем выделите память для SQLDA и установите число входных параметров в поле `sqln`, члене структуры `sqlda_t`. Когда для подготовленного запроса требуются два или более входных параметров, приложение должно выделить дополнительное место в памяти, размер которого вычисляется как (число параметров - 1) * sizeof(sqlvar_t). В показанном здесь примере выделяется место для двух параметров.

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /* число входных переменных */
```

Выделив память, сохраните значения параметров в массиве `sqlvar[]`. (Этот же массив используется для значений столбцов, когда SQLDA получает набор результатов.) В данном примере передаются два параметра: "`postgres`" (строкового типа) и `1` (целочисленного типа).

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;

int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

Откройте курсор с указанием ранее созданной SQLDA, чтобы входные параметры были переданы подготовленному оператору.

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

Наконец, закончив использование входных SQLDA, необходимо явно освободить выделенную для них память, в отличие от SQLDA, получающих результаты запросов.

```
free(sqlda2);
```

34.7.2.4. Пример приложения, использующего SQLDA

Представленный здесь пример программы показывает, как выбрать из системных каталогов статистику доступа к базам данных, определённых входными параметрами.

Это приложение соединяет записи двух системных таблиц, pg_database и pg_stat_database по OID базы данных, и также выбирает и показывает статистику, принимая два входных параметра (база данных `postgres` и OID `1`).

Сначала создайте SQLDA для ввода параметров и SQLDA для вывода результатов.

```
EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* выходной дескриптор */
sqlda_t *sqlda2; /* входной дескриптор */
```

Затем подключитесь к базе данных, подготовьте оператор и объявите курсор для подготовленного оператора.

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE
 d.oid=s.datid AND ( d.datname=? OR d.oid=? )";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

Затем запишите некоторые значения параметров во входную SQLDA. Выделите память для входной SQLDA и установите количество параметров в `sqln`. Запишите тип, значение и длину значения в поля `sqltype`, `sqldata` и `sqllen` структуры `sqlvar`.

```
    /* Создать структуру SQLDA для входных параметров. */
    sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* число входных переменных */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *)&intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

Подготовив входную SQLDA, откройте курсор с ней.

```
    /* Открыть курсор с входными параметрами. */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

Выберите строки из открытого курсора в выходную SQLDA. (Обычно, чтобы выбрать все строки в наборе результатов, нужно повторять `FETCH` в цикле.)

```
    while (1)
    {
        sqlda_t *cur_sqlda;
```

```
        /* Назначить дескриптор курсору */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

Затем прочитайте выбранные записи из SQLDA, следуя по связанному списку структуры `sqlda_t`.

```
    for (cur_sqlda = sqlda1 ;
         cur_sqlda != NULL ;
         cur_sqlda = cur_sqlda->desc_next)
    {
        ...
```

Прочитайте все столбцы первой записи. Количество столбцов хранится в поле `sqld`, а данные первого столбца в `sqlvar[0]`, оба эти поля — члены структуры `sqlda_t`.

```
        /* Вывести каждый столбец в строке. */
        for (i = 0; i < sqlda1->sqld; i++)
        {
            sqlvar_t v = sqlda1->sqlvar[i];
            char *sqldata = v.sqldata;
            short sqllen  = v.sqllen;

            strncpy(name_buf, v.sqlname.data, v.sqlname.length);
            name_buf[v.sqlname.length] = '\0';
```

Теперь данные столбцов сохранены в переменной `v`. Скопируйте все элементы данных в переменные среды, определив тип столбца по полю `v.sqltype`.

```
            switch (v.sqltype) {
                int intval;
                double doubleval;
                unsigned long long int longlongval;

                case ECPGt_char:
                    memset(&var_buf, 0, sizeof(var_buf));
                    memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ?
 sizeof(var_buf)-1 : sqllen));
                    break;

                case ECPGt_int: /* integer */
                    memcpy(&intval, sqldata, sqllen);
                    snprintf(var_buf, sizeof(var_buf), "%d", intval);
                    break;

                ...

                default:
                    ...
            }

            printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
        }
```

Закончив обработку всех записей, закройте курсор и отключитесь от базы данных.

```
    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;
```

Вся программа показана в Примере 34.1.

Пример 34.1. Пример программы на базе SQLDA

```
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* дескриптор для выходных данных */
sqlda_t *sqlda2; /* дескриптор для входных данных */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE
 d.oid=s.datid AND ( d.datname=? OR d.oid=? )";

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;

    /* Создать структуру SQLDA для входных параметров */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* число входных переменных */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);

    /* Открыть курсор с входными параметрами. */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;

        /* Присвоить дескриптор курсору */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;

        for (cur_sqlda = sqlda1 ;
```

```
             cur_sqlda != NULL ;
             cur_sqlda = cur_sqlda->desc_next)
        {
            int i;
            char name_buf[1024];
            char var_buf[1024];

            /* Напечатать каждый столбец в строке. */
            for (i=0 ; i<cur_sqlda->sqld ; i++)
            {
                sqlvar_t v = cur_sqlda->sqlvar[i];
                char *sqldata = v.sqldata;
                short sqllen  = v.sqllen;

                strncpy(name_buf, v.sqlname.data, v.sqlname.length);
                name_buf[v.sqlname.length] = '\0';

                switch (v.sqltype)
                {
                    case ECPGt_char:
                        memset(&var_buf, 0, sizeof(var_buf));
                        memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqllen ?
 sizeof(var_buf)-1 : sqllen) );
                        break;

                    case ECPGt_int: /* integer */
                        memcpy(&intval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%d", intval);
                        break;

                    case ECPGt_long_long: /* bigint */
                        memcpy(&longlongval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                        break;

                    default:
                    {
                        int i;
                        memset(var_buf, 0, sizeof(var_buf));
                        for (i = 0; i < sqllen; i++)
                        {
                            char tmpbuf[16];
                            snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned char)
 sqldata[i]);
                            strncat(var_buf, tmpbuf, sizeof(var_buf));
                        }
                    }
                        break;
                }

                printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
            }

            printf("\n");
        }
    }

    EXEC SQL CLOSE cur1;
```

```
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

Вывод этой программы должен быть примерно таким (некоторые числа будут меняться).

```
oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
datallowconn = t (type: 1)
dathasloginevt = f (type: 1)
datconnlimit = -1 (type: 5)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
dathasloginevt = f (type: 1)
datconnlimit = -1 (type: 5)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl =  (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

34.8. Обработка ошибок

В этом разделе описывается, как можно обрабатывать исключительные условия и предупреждения в программе со встраиваемым SQL. Для этого предназначены два средства, которые могут дополнять друг друга.

- Можно настроить функции-обработчики для обработки предупреждений и ошибок, воспользовавшись командой `WHENEVER`.
- Подробную информацию об ошибке или предупреждении можно получить через переменную `sqlca`.

34.8.1. Установка обработчиков

Один простой метод перехвата ошибок и предупреждений заключается в назначении определённого действия, которое будет выполняться при некотором условии. В общем виде:

```
EXEC SQL WHENEVER условие действие;
```

Здесь *`условие`* может быть следующим:

`SQLERROR`

Указанное действие вызывается, когда при выполнении SQL-оператора происходит ошибка.

`SQLWARNING`

Указанное действие вызывается, когда при выполнении SQL-оператора выдаётся предупреждение.

`NOT FOUND`

Указанное действие вызывается, когда SQL-оператор получает или обрабатывает ноль строк. (Это обстоятельство не считается ошибкой, но бывает полезно отследить его.)

`действие` может быть следующим:

`CONTINUE`

Это фактически означает, что условие игнорируется. Это поведение по умолчанию.

`GOTO` *`метка`*
`GO TO` *`метка`*

Перейти к указанной метке (используя оператор `goto` языка C).

`SQLPRINT`

Вывести сообщение в устройство стандартного вывода. Это полезно для простых программ или при разработке прототипов. Содержание этого сообщения не настраивается.

`STOP`

Вызвать `exit(1)`, что приведёт к завершению программы.

`DO BREAK`

Выполнить оператор `break` языка C. Этот вариант следует использовать только в циклах или операторах `switch`.

`DO CONTINUE`

Выполнить оператор `continue` языка C. Этот вариант следует использовать только в циклах. Данный оператор передаёт управление в начало цикла.

`CALL` *`имя`* `(`*`аргументы`*`)`
`DO` *`имя`* `(`*`аргументы`*`)`

Вызвать указанные функции C с заданными аргументами. (Эти вызовы имеют смысловые отличия от `CALL` и `DO` в обычной грамматике Postgres Pro.)

В стандарте SQL описаны только действия `CONTINUE` и `GOTO` (и `GO TO`).

Ниже показан простой пример использования этих команд. Эта конструкция выводит простое сообщение при выдаче предупреждения и прерывает программу в случае ошибки:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

Оператор `EXEC SQL WHENEVER` является директивой препроцессора SQL, а не оператором языка C. Устанавливаемое им действие при ошибках или предупреждениях применяется ко всем встраиваемым SQL-операторам ниже точки, где устанавливается обработчик, если только это действие не было изменено после первой команды `EXEC SQL WHENEVER`, и до SQL-оператора, вызвавшего это условие, вне зависимости от хода выполнения программы на C. Поэтому обе следующие программы на C не дадут желаемого эффекта:

```
/*
 * НЕПРАВИЛЬНО
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}
```

```
/*
 * НЕПРАВИЛЬНО
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

34.8.2. sqlca

Для более гибкой обработки ошибок в интерфейсе встраиваемого SQL представлена глобальная переменная с именем `sqlca` (SQL Communication Area, Область сведений SQL), имеющая следующую структуру:

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
```

```
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

(В многопоточной программе каждый поток автоматически получает собственную копию `sqlca`. Это работает подобно стандартной в C глобальной переменной `errno`.)

Структура `sqlca` покрывает и предупреждения, и ошибки. Если в процессе выполнения оператора выдаётся несколько предупреждений или ошибок, `sqlca` будет содержать сведения только о последнем(ей) из них.

Если последний оператор SQL выполняется без ошибки, `sqlca.sqlcode` будет содержать 0, а `sqlca.sqlstate` — `"00000"`. Если выдаётся предупреждение или ошибка, в `sqlca.sqlcode` будет содержаться отрицательное число, а `sqlca.sqlstate` будет отличаться от `"00000"`. Положительное значение `sqlca.sqlcode` устанавливается при нейтральном событии, например, когда последний запрос возвращает ноль строк. Поля `sqlcode` и `sqlstate` представляют две различные схемы кодов ошибок; подробнее они описаны ниже.

Если последний SQL-оператор был успешным, в `sqlca.sqlerrd[1]` содержится OID обработанной строки (если это уместно), а в `sqlca.sqlerrd[2]` количество обработанных или возвращённых строк (если это уместно для команды).

В случае ошибки или предупреждения `sqlca.sqlerrm.sqlerrmc` будет содержать строку, описывающую ошибку. Поле `sqlca.sqlerrm.sqlerrml` содержит длину сообщения об ошибке, которое хранится в `sqlca.sqlerrm.sqlerrmc` (результат функции `strlen()`, который не очень интересен для программиста C). Заметьте, что некоторые сообщения могут не умещаться в массив `sqlerrmc` фиксированного размера; они будут обрезаться.

В случае предупреждения, в `sqlca.sqlwarn[2]` записывается символ `W`. (Во всех других случаях значение будет отличным от `W`.) Символ `W` в `sqlca.sqlwarn[1]` показывает, что значение было обрезано при сохранении в переменной среды C. `W` в `sqlca.sqlwarn[0]` устанавливается, если предупреждение отмечается в каком-либо другом элементе массива.

Поля `sqlcaid`, `sqlabc`, `sqlerrp` и остальные элементы `sqlerrd` и `sqlwarn` в настоящее время не содержат полезной информации.

Структура `sqlca` не определена в стандарте SQL, но реализована в нескольких других СУБД SQL. Принципиально она определяется одинаково, но если вы хотите, чтобы ваши приложения были переносимыми, тщательно изучите различия реализаций.

В следующем примере, демонстрирующем применение `WHENEVER` в сочетании с `sqlca`, выводится содержимое `sqlca` при возникновении ошибки. Это может быть полезно для отладки или в прототипах, пока не реализован более «дружественный пользователю» обработчик ошибок.

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n",
 sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],

 sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
```

```
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0],
 sqlca.sqlwarn[1], sqlca.sqlwarn[2],
                                                          sqlca.sqlwarn[3],
 sqlca.sqlwarn[4], sqlca.sqlwarn[5],
                                                          sqlca.sqlwarn[6],
 sqlca.sqlwarn[7]);
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "===============\n");
}
```

Результат может выглядеть следующим образом (при ошибке, вызванной опечаткой в имени таблицы):

```
==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
===============
```

34.8.3. `SQLSTATE` и `SQLCODE`

Поля `sqlca.sqlstate` и `sqlca.sqlcode` отражают две различные схемы, представляющие коды ошибок. Обе схемы пришли из стандарта SQL, но схема `SQLCODE` была признана устаревшей в редакции SQL-92 и исключена в последующих редакциях. Поэтому в новых приложениях настоятельно рекомендуется использовать `SQLSTATE`.

`SQLSTATE` задаётся в массиве из пяти символов. Эти пять символов содержат цифры или буквы в верхнем регистре, представляющие коды различных условий ошибок и предупреждений. `SQLSTATE` определяется по иерархической схеме: первые два символа обозначают общий класс условия, а следующие три — подкласс общего условия. Успешное состояние обозначается кодом `00000`. По большей части коды `SQLSTATE` определены в стандарте SQL. Сервер Postgres Pro поддерживает коды ошибок `SQLSTATE` естественным образом; поэтому используя во всех приложениях именно эту схему кодов ошибок, можно добиться высокой степени согласованности. За дальнейшими сведениями обратитесь к Приложению A.

`SQLCODE` — устаревшая схема, в которой коды ошибок представлены просто целыми числами. Значение 0 обозначает успешное завершение, положительное значение — успешное завершение с дополнительной информацией, а отрицательное говорит об ошибке. В стандарте SQL определено только положительное значение +100, показывающее, что последняя команда вернула или затронула ноль строк, но отрицательные значения не определены. Таким образом, с этой схемой нельзя рассчитывать на переносимость и она не имеет иерархической структуры. Исторически сложилось, что процессор встраиваемого SQL для Postgres Pro назначает некоторые определённые значения `SQLCODE` для собственного использования; они перечислены ниже с числовыми значениями и символьными именами. Помните, что эти коды несовместимы с другими реализациями SQL. Поэтому для упрощения перевода приложений на схему `SQLSTATE` вместе с этими кодами перечисляются соответствующие значения `SQLSTATE`. Однако однозначного соответствия один-к-одному или один-ко-многим между этими двумя схемами не существует (на самом деле это соответствие многие-ко-многим), поэтому следует свериться со списком `SQLSTATE` в Приложении A в каждом случае.

`SQLCODE` может принимать следующие значения:

0 (`ECPG_NO_ERROR`)

Показывает, что ошибки нет. (SQLSTATE 00000)

100 (`ECPG_NOT_FOUND`)

Это нейтральное условие, показывающее, что последняя команда вернула или обработала ноль строк, либо курсор достиг конца. (SQLSTATE 02000)

Выбирая данные из курсора в цикле, можно проверять этот код, чтобы понять, когда нужно прервать цикл, следующим образом:

```
while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

Но `WHENEVER NOT FOUND DO BREAK` внутри по сути делает это же, поэтому такое явное условие обычно ничем не лучше.

-12 (`ECPG_OUT_OF_MEMORY`)

Указывает, что закончилась виртуальная память. Числовое значение определено как `-ENOMEM`. (SQLSTATE YE001)

-200 (`ECPG_UNSUPPORTED`)

Указывает, что препроцессор сгенерировал код, который не понимает библиотека. Возможно, вы используете несовместимые версии препроцессора и библиотеки. (SQLSTATE YE002)

-201 (`ECPG_TOO_MANY_ARGUMENTS`)

Это означает, что в команде было указано больше переменных среды, чем она ожидает. (SQLSTATE 07001 или 07002)

-202 (`ECPG_TOO_FEW_ARGUMENTS`)

Это означает, что в команде было указано меньше переменных среды, чем она ожидает. (SQLSTATE 07001 или 07002)

-203 (`ECPG_TOO_MANY_MATCHES`)

Это означает, что запрос вернул несколько строк, но оператор был подготовлен только для одной строки результата (например, потому что переданные переменные — не массивы). (SQLSTATE 21000)

-204 (`ECPG_INT_FORMAT`)

Переменная среды C типа `int` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `int`. Для этого преобразования библиотека использует функцию `strtol()`. (SQLSTATE 42804)

-205 (`ECPG_UINT_FORMAT`)

Переменная среды C типа `unsigned int` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `unsigned int`. Для этого преобразования библиотека использует функцию `strtoul()`. (SQLSTATE 42804)

-206 (`ECPG_FLOAT_FORMAT`)

Переменная среды C типа `float` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать во `float`. Для этого преобразования библиотека использует функцию `strtod()`. (SQLSTATE 42804)

-207 (`ECPG_NUMERIC_FORMAT`)

Переменная среды C типа `numeric` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `numeric`. (SQLSTATE 42804)

-208 (`ECPG_INTERVAL_FORMAT`)

Переменная среды C типа `interval` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `interval`. (SQLSTATE 42804)

-209 (`ECPG_DATE_FORMAT`)

Переменная среды C типа `date` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `date`. (SQLSTATE 42804)

-210 (`ECPG_TIMESTAMP_FORMAT`)

Переменная среды C типа `timestamp` и данные в базе имеют разные типы, и в этих данных содержится значение, которое нельзя преобразовать в `timestamp`. (SQLSTATE 42804)

-211 (`ECPG_CONVERT_BOOL`)

Это означает, что переменная среды C имеет тип `bool`, а значение в базе данных отличается от `'t'` или `'f'`. (SQLSTATE 42804)

-212 (`ECPG_EMPTY`)

Серверу Postgres Pro был передан пустой оператор. (Этого обычно не должно происходить в программе со встраиваемым SQL, так что это может указывать на внутреннюю ошибку.) (SQLSTATE YE002)

-213 (`ECPG_MISSING_INDICATOR`)

Возвращено значение NULL, но переменная-индикатор NULL не задана. (SQLSTATE 22002)

-214 (`ECPG_NO_ARRAY`)

Там, где требуется массив, была передана обычная переменная. (SQLSTATE 42804)

-215 (`ECPG_DATA_NOT_ARRAY`)

База данных возвратила обычную переменную там, где требуется значение-массив. (SQLSTATE 42804)

-216 (`ECPG_ARRAY_INSERT`)

Не удалось вставить значение в массив. (SQLSTATE 42804)

-220 (`ECPG_NO_CONN`)

Программа попыталась использовать несуществующее подключение. (SQLSTATE 08003)

-221 (`ECPG_NOT_CONN`)

Программа попыталась использовать подключение, которое существует, но не было открыто. (Это внутренняя ошибка.) (SQLSTATE YE002)

-230 (`ECPG_INVALID_STMT`)

Оператор, который вы пытаетесь выполнить, не был подготовлен. (SQLSTATE 26000)

-239 (`ECPG_INFORMIX_DUPLICATE_KEY`)

Ошибка повторяющегося ключа, нарушение ограничения уникальности (режим совместимости с Informix). (SQLSTATE 23505)

-240 (`ECPG_UNKNOWN_DESCRIPTOR`)

Указанный дескриптор не найден. Оператор, который вы пытаетесь использовать, не был подготовлен. (SQLSTATE 33000)

-241 (`ECPG_INVALID_DESCRIPTOR_INDEX`)

Указанный индекс дескриптора вне диапазона. (SQLSTATE 07009)

-242 (`ECPG_UNKNOWN_DESCRIPTOR_ITEM`)

Запрошен неверный элемент дескриптора. (Это внутренняя ошибка.) (SQLSTATE YE002)

-243 (`ECPG_VAR_NOT_NUMERIC`)

При выполнении динамического оператора база данных возвратила числовое значение, тогда как переменная среды C — не числовая. (SQLSTATE 07006)

-244 (`ECPG_VAR_NOT_CHAR`)

При выполнении динамического оператора база данных возвратила не числовое значение, тогда как переменная среды C — числовая. (SQLSTATE 07006)

-284 (`ECPG_INFORMIX_SUBSELECT_NOT_ONE`)

Результат подзапроса представлен не одной строкой (режим совместимости с Informix). (SQLSTATE 21000)

-400 (`ECPG_PGSQL`)

Ошибка произошла на стороне сервера Postgres Pro. В тексте ошибки содержится сообщение об ошибке от сервера Postgres Pro.

-401 (`ECPG_TRANS`)

Сервер Postgres Pro сообщает, что клиент не может запускать, фиксировать или отменять транзакцию. (SQLSTATE 08007)

-402 (`ECPG_CONNECT`)

Попытка подключения к базе данных была неудачной. (SQLSTATE 08001)

-403 (`ECPG_DUPLICATE_KEY`)

Ошибка повторяющегося ключа, нарушение ограничения уникальности. (SQLSTATE 23505)

-404 (`ECPG_SUBSELECT_NOT_ONE`)

Результат подзапроса представлен не одной строкой. (SQLSTATE 21000)

-602 (`ECPG_WARNING_UNKNOWN_PORTAL`)

Указано неверное имя курсора. (SQLSTATE 34000)

-603 (`ECPG_WARNING_IN_TRANSACTION`)

Транзакция в процессе выполнения. (SQLSTATE 25001)

-604 (`ECPG_WARNING_NO_TRANSACTION`)

Нет активной (выполняющейся) транзакции. (SQLSTATE 25P01)

-605 (`ECPG_WARNING_PORTAL_EXISTS`)

Было указано имя существующего курсора. (SQLSTATE 42P03)

34.9. Директивы препроцессора

Препроцессор `ecpg` поддерживает ряд директив, которые позволяют управлять разбором и обработкой исходных файлов.

34.9.1. Включение файлов

Для включения внешнего файла в программу со встраиваемым SQL, используется конструкция:

```
EXEC SQL INCLUDE имя_файла;
EXEC SQL INCLUDE <имя_файла>;
EXEC SQL INCLUDE "имя_файла";
```

Встретив такую директиву, препроцессор встраиваемого SQL будет искать файл *`имя_файла`*`.h`, обрабатывать его и включать в выходной код C. В результате встраиваемые SQL-операторы во включённом таким образом файле будут обработаны корректно.

Препроцессор `ecpg` будет искать указанный файл в нескольких каталогах в следующем порядке:

- текущий каталог
- `/usr/local/include`
- каталог включаемых файлов Postgres Pro, определённый во время сборки (например, `/usr/local/pgsql/include`)
- `/usr/include`

Но когда используется форма `EXEC SQL INCLUDE "имя_файла"`, просматривается только текущий каталог.

В каждом каталоге препроцессор будет сначала искать файл с заданным именем, а если не обнаружит его, попытается найти файл с добавленным расширением `.h` (если только заданное имя файла уже не содержит это расширение).

Заметьте, что команда `EXEC SQL INCLUDE` *не* равнозначна включению:

```
#include <имя_файла.h>
```

так как во втором случае включаемый файл не проходит через препроцессор SQL-команд. Естественно, директиву C `#include` можно по-прежнему применять для включения других заголовочных файлов.

Примечание

Имя включаемого файла чувствительно к регистру, несмотря на то, что остальная команда `EXEC SQL INCLUDE` подчиняется обычным правилам чувствительности к регистру SQL.

34.9.2. Директивы define и undef

Во встраиваемом SQL есть конструкция, подобная директиве `#define`, известной в C:

```
EXEC SQL DEFINE имя;
EXEC SQL DEFINE имя значение;
```

Используя её, можно определить имя:

```
EXEC SQL DEFINE HAVE_FEATURE;
```

И также можно определить константы:

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

Удалить предыдущее определение позволяет команда `undef`:

```
EXEC SQL UNDEF MYNUMBER;
```

Разумеется, в программе со встраиваемым SQL можно продолжать использовать версии `#define` и `#undef` языка C. Отличие состоит в том, когда вычисляются определяемые значения. Когда применяется команда `EXEC SQL DEFINE`, вычислять определения и подставлять значения будет препроцессор `ecpg`. Например, если написать:

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

подстановку выполнит `ecpg` и компилятор C никогда не увидит имени или идентификатора `MYNUMBER`. Заметьте, что с другой стороны `#define` не подходит для определения константы, которую вы хотите использовать во встраиваемом SQL, так как препроцессор встраиваемого SQL не сможет увидеть это определение.

Если в командной строке препроцессора `ecpg` указано несколько входных файлов, `EXEC SQL DEFINE` и `EXEC SQL UNDEF` не влияют на них: каждый файл начинается только с символов, определённых ключами `-D` в командной строке.

34.9.3. Директивы ifdef, ifndef, elif, else и endif

Для условной компиляции блоков кода можно использовать следующие указания:

`EXEC SQL ifdef *имя*;`

Проверяет *имя* и обрабатывает последующие строки, если *имя* было определено командой `EXEC SQL define *имя*`.

`EXEC SQL ifndef *имя*;`

Проверяет *имя* и обрабатывает последующие строки, если *имя* *не* было определено командой `EXEC SQL define *имя*`.

`EXEC SQL elif *имя*;`

Начинает необязательный альтернативный блок после указания `EXEC SQL ifdef *имя*` или `EXEC SQL ifndef *имя*`. Количество блоков `elif` может быть любым. Строки, следующие за `elif`, будут обрабатываться, если *имя* определено *и* при этом не был обработан ни один из блоков той же конструкции `ifdef`/`ifndef`...`endif`.

`EXEC SQL else;`

Начинает необязательный заключительный блок после указания EXEC SQL ifdef *имя* или EXEC SQL ifndef *имя*. Последующие строки будут обрабатываться, если не был обработан ни один из блоков той же конструкции `ifdef`/`ifndef`...`endif`.

`EXEC SQL endif;`

Завершает конструкцию `ifdef`/`ifndef`...`endif`. Последующие строки обрабатываются обычным образом.

Конструкции `ifdef`/`ifndef`...`endif` могут быть вложенными, на глубину до 127 уровней.

Так будет скомпилирована только одна из трёх команд `SET TIMEZONE`:

```
EXEC SQL ifdef TZVAR;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL endif;
```

34.10. Компиляция программ со встраиваемым SQL

Теперь, когда вы получили представление, как писать программы на C со встраиваемым SQL, вы наверное хотите узнать, как их компилировать. Прежде чем компилировать код C, необходимо пропустить исходный файл через препроцессор встраиваемого SQL в C, который преобразует записанные вами операторы SQL в вызовы специальных функций. После компиляции полученный объектный код нужно скомпоновать со специальной библиотекой, содержащей необходимые функции. Эти функции получают информацию из аргументов, выполняют команды SQL через интерфейс libpq, и помещают результат в аргументы, заданные для вывода.

Программа препроцессора называется `ecpg` и входит в состав обычной инсталляции Postgres Pro. Программам со встраиваемым SQL, как правило, даются имена с расширением `.pgc`. Если вы создали код программы в файле `prog1.pgc`, вы можете обработать его, просто выполнив:

```
ecpg prog1.pgc
```

При этом будет создан файл `prog1.c`. Если имена входных файлов не следуют этому соглашению, имя выходного файла можно задать явно в аргументе `-o`.

Обработанный препроцессором файл можно скомпилировать обычным образом, например, так:

```
cc -c prog1.c
```

В сгенерированные исходные файлы C включаются заголовочные файлы из инсталляции Postgres Pro, поэтому если вы установили Postgres Pro так, что соответствующий каталог не просматривается по умолчанию, вам придётся добавить указание вида `-I/usr/local/pgsql/include` в командную строку компиляции.

Чтобы скомпоновать программу со встраиваемым SQL, необходимо подключить библиотеку `libecpg` примерно так:

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

Возможно, и для этого понадобится добавить в командную строку указание вида `-L/usr/local/pgsql/lib`.

Чтобы узнать пути вашей инсталляции, можно воспользоваться командой `pg_config`или `pkg-config`(в качестве имени пакета нужно указать `libecpg`).

Если вы организуете процесс сборки большого проекта с применением make, может быть удобно включить в ваши сборочные файлы следующее неявное правило:

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

Полный синтаксис вызова команды `ecpg` приведён в описании ecpg.

Библиотека ecpg по умолчанию потокобезопасна. Однако для компиляции клиентского кода могут потребоваться параметры командной строки для настройки многопоточности.

34.11. Библиотечные функции

Библиотека `libecpg` в основном содержит «скрытые» функции, применяемые для реализации функциональности, выражаемой встраиваемыми командами SQL. Но есть также некоторые функции, которые можно вызывать напрямую. Заметьте, что код, задействующий эти функции, будет непереносимым.

- `ECPGdebug(int` *`вкл`*`, FILE *`*`поток`*`)` с первым аргументом, отличным от нуля, включает вывод отладочных сообщений в заданный *`поток`*. Журнал сообщений, полученный таким образом, будет содержать все операторы SQL с заданными входными переменными и результаты, выданные сервером Postgres Pro. Это может быть очень полезно для поиска ошибок в командах SQL.

  ### Примечание

  В Windows, если библиотека ecpg и приложение скомпилированы с разными флагами, эта функция может вызвать крах приложения из-за различий внутреннего представления указателей `FILE`. В частности, флаги многопоточной/однопоточной, выпускаемой/отладочной или статической/динамической сборки должны быть одинаковыми для библиотеки и всех использующих её приложений.

- `ECPGget_PGconn(const char *`*`имя_подключения`*`)` возвращает указатель на подключение к базе данных, имеющее заданное имя. Если аргумент *`имя_подключения`* равен `NULL`, возвращается указатель на текущее подключение. Если определить подключение не удаётся, возвращается

`NULL`. Полученный указатель на подключение, если требуется, можно использовать при вызове любых других функций libpq.

> **Примечание**
>
> Манипулировать подключениями, открытыми средствами ecpg, напрямую через libpq не следует.

- `ECPGtransactionStatus(const char *`*`имя_подключения`*`)` возвращает состояние текущей транзакции для подключения, на которое указывает *`имя_подключения`*. О возвращаемых кодах состояния можно узнать в Раздел 32.2 и в описании входящей в libpq функции `PQtransactionStatus`.
- `ECPGstatus(int` *`номер_строки`*`, const char*` *`имя_подключения`*`)` возвращает true при наличии подключения к базе данных и false в противном случае. В аргументе *`имя_подключения`* можно передать `NULL`, если применяется одно подключение.

34.12. Большие объекты

ECPG не поддерживает большие объекты напрямую, но приложение на базе ECPG может работать с большими объектами, используя предназначенные для этого функции, получив необходимый объект `PGconn` в результате вызова `ECPGget_PGconn()`. (Однако использовать функцию `ECPGget_PGconn()` и напрямую воздействовать на объекты `PGconn` следует очень осторожно; в идеале стоит исключить при этом другие обращения к базе данных через ECPG.)

Подробнее функция `ECPGget_PGconn()` описана в Разделе 34.11. Интерфейс функций для работы с большими объектами рассмотрен в Главе 33.

Функции для работы с большими объектами должны вызываться в блоке транзакций, поэтому если режим автофиксации отключён, необходимо явно выдавать команды `BEGIN`.

В Примере 34.2 приведён пример программы, показывающий, как создать, записать и прочитать большой объект в приложении ECPG.

Пример 34.2. Программа на базе ECPG, работающая с большими объектами

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn     *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
```

```
    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* create */
    loid = lo_create(conn, 0);
    if (loid &lt; 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));

    printf("loid = %d\n", loid);

    /* write test */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_write(conn, fd, buf, buflen);
    if (rc &lt; 0)
        printf("lo_write() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* read test */
    fd = lo_open(conn, loid, INV_READ);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc &lt; 0)
        printf("lo_read() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* check */
    rc = memcmp(buf, buf2, buflen);
    printf("memcmp() = %d\n", rc);

    /* cleanup */
    rc = lo_unlink(conn, loid);
    if (rc &lt; 0)
        printf("lo_unlink() failed: %s", PQerrorMessage(conn));

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

34.13. Приложения на C++

ECPG обеспечивает поддержку языка C++ в ограниченном объёме. Некоторые её особенности описаны в этом разделе.

Препроцессор `ecpg` принимает входной файл, написанный на C (или языке, подобном C) со встраиваемыми командами SQL, преобразует встроенные команды SQL в конструкции языка C и в результате формирует файл `.c`. Объявления библиотечных функций, вызываемых в конструкциях C, которые генерирует `ecpg`, заворачиваются в блоки `extern "C" { ... }` при использовании C++, так что они должны прозрачно работать в C++.

Однако вообще говоря, препроцессор `ecpg` понимает только C; он не воспринимает особый синтаксис и зарезервированные слова языка C++. Поэтому какой-то код SQL, встроенный в код приложения на C++, в котором используются сложные особенности C++, может корректно не обработаться препроцессором или не работать как ожидается.

Надёжный подход к применению внедрённого кода SQL в приложении на C++ заключается в том, чтобы скрыть вызовы ECPG в модуле C, который будет вызываться приложением на C++ для работы с базой данных и который будет скомпонован с остальным кодом C++. Подробнее это описано в Подразделе 34.13.2.

34.13.1. Область видимости переменных среды

Препроцессор `ecpg` имеет понимание области видимости переменных в C. С языком C это довольно просто, так как область видимости переменных определяется их блоками кода. В C++, однако, переменные-члены класса задействуются не в том блоке кода, в каком они объявлены, так что препроцессор `ecpg` не сможет корректно определить область видимости таких переменных.

Например, в следующем случае препроцессор `ecpg` не сможет найти определение переменной `dbname` в методе `test`, так что произойдёт ошибка.

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}
```

При обработке данного кода будет выдано сообщение:

```
ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

(test_cpp.pgc:28: ОШИБКА: переменная "dbname" не объявлена)

Для решения этой проблемы можно немного изменить метод `test` и задействовать в нём локальную переменную для промежуточного хранения. Но предложенный подход нельзя считать хорошим, так как это портит код и снижает производительность.

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :tmp;
    strlcpy(dbname, tmp, sizeof(tmp));

    printf("current_database = %s\n", dbname);
}
```

34.13.2. Разработка приложения на C++ с внешним модулем на C

Если вы поняли технические ограничения препроцессора `ecpg` с C++, вы можете прийти к заключению, что для использования ECPG в приложениях на C++ лучше связывать код C с кодом C++ на стадии компоновки, а не внедрять команды SQL непосредственно в код на C++. В данном разделе показывается, как отделить встраиваемые команды SQL от кода приложения на C++, на простом примере. В этом примере приложение реализуется на C++, а взаимодействие с сервером Postgres Pro построено на C и ECPG.

Для сборки нужно создать три типа файлов: файл на C (`*.pgc`), заголовочный файл и файл на C++:

`test_mod.pgc`

Модуль подпрограмм будет выполнять SQL-команды, встроенные в C. Этот код нужно будет преобразовать в `test_mod.c` с помощью препроцессора.

```
#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL
 COMMIT;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

`test_mod.h`

Заголовочный файл с объявлениями функций в модуле на языке C (`test_mod.pgc`). Он включается в `test_cpp.cpp`. Объявления в этом файле должны заключаться в блок `extern "C"`, так как он будет связываться с модулем C++.

```
#ifdef __cplusplus
extern "C" {
#endif

void db_connect();
void db_test();
void db_disconnect();

#ifdef __cplusplus
}
#endif
```

`test_cpp.cpp`

Основной код приложения, содержащий функцию `main`, а также, в данном примере, класс C++.

```
#include "test_mod.h"

class TestCpp
{
  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    db_connect();
}

void
TestCpp::test()
{
    db_test();
}

TestCpp::~TestCpp()
{
    db_disconnect();
}

int
main(void)
{
    TestCpp *t = new TestCpp();

    t->test();
    return 0;
}
```

Для сборки приложения проделайте следующее. Преобразуйте `test_mod.pgc` в `test_mod.c` с помощью `ecpg`, а затем получите `test_mod.o`, скомпилировав `test_mod.c` компилятором C:

```
ecpg -o test_mod.c test_mod.pgc
```

```
cc -c test_mod.c -o test_mod.o
```

После этого получите `test_cpp.o`, скомпилировав `test_cpp.cpp` компилятором C++:

```
c++ -c test_cpp.cpp -o test_cpp.o
```

Наконец, свяжите полученные объектные файлы, `test_cpp.o` и `test_mod.o`, в один исполняемый файл, выполнив компоновку под управлением компилятора C++:

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

34.14. Команды встраиваемого SQL

В этом разделе описаны все команды, предназначенные специально для встраиваемого SQL. В Справке: «Команды SQL» также описаны обычные команды SQL, которые можно использовать и как встраиваемые, если явно не отмечено обратное.

ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR — выделить область SQL-дескриптора

Синтаксис

```
ALLOCATE DESCRIPTOR имя
```

Описание

`ALLOCATE DESCRIPTOR` выделяет новую именованную область SQL-дескриптора, через которую можно обмениваться данными между сервером Postgres Pro и программой на C.

После использования области дескрипторов должны освобождаться командой `DEALLOCATE DESCRIPTOR`.

Параметры

имя

Имя SQL-дескриптора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

Примеры

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

Совместимость

Команда `ALLOCATE DESCRIPTOR` описана в стандарте SQL.

См. также

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

CONNECT

CONNECT — установить подключение к базе данных

Синтаксис

```
CONNECT TO цель_подключения [ AS имя_подключения ] [ USER пользователь_подключения ]
CONNECT TO DEFAULT
CONNECT пользователь_подключения
DATABASE цель_подключения
```

Описание

Команда `CONNECT` устанавливает подключение клиента к серверу Postgres Pro.

Параметры

цель_подключения

цель_соединения задаёт целевой сервер и базу для подключения в одной из нескольких форм.

[*имя_бд*] [`@`*сервер*] [`:`*порт*]

Подключение по TCP/IP

`unix:postgresql://`*сервер* [`:`*порт*] `/` [*имя_бд*] [`?`*параметр_подключения*]

Подключение через Unix-сокеты

`tcp:postgresql://`*сервер* [`:`*порт*] `/` [*имя_бд*] [`?`*параметр_подключения*]

Подключение по TCP/IP

Строковая константа SQL

содержащая значение в одной из показанных выше форм

переменная среды C

переменная среды C типа `char[]` или `VARCHAR[]`, содержащая значение в одной из показанных выше форм

имя_подключения

Необязательный идентификатор подключения, позволяющий обращаться к этому подключению в других командах. Это может быть идентификатор SQL или переменная среды C.

пользователь_подключения

Имя пользователя для подключения к базе данных.

В этом параметре также можно передать имя и пароль одним из следующих способов: *имя_пользователя*`/`*пароль*, *имя_пользователя* `IDENTIFIED BY` *пароль* или *имя_пользователя* `USING` *пароль*.

В качестве имени пользователя и пароля можно задать идентификаторы SQL, строковые константы или переменные среды.

`DEFAULT`

Использовать все параметры подключения по умолчанию, которые определены библиотекой libpq.

Примеры

Несколько вариантов указания параметров подключения:

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER
 connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY
 connectpw;
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser IDENTIFIED
 BY connectpw;
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser IDENTIFIED
 BY "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING
 "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14 USER
 connectuser;
```

Следующий пример программы демонстрирует применение переменных среды для определения параметров подключения:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    char *dbname     = "testdb";    /* имя базы данных */
    char *user       = "testuser";  /* имя пользователя подключения */
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                    /* строка подключения */
    char ver[256];                  /* буфер для хранения строки версии */
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT pgpro_version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT pgpro_version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);
```

```
    return 0;
}
```

Совместимость

Команда `CONNECT` описана в стандарте SQL, но формат параметров подключения определяется реализацией.

См. также

DISCONNECT, SET CONNECTION

DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR — освободить область SQL-дескриптора

Синтаксис

```
DEALLOCATE DESCRIPTOR имя
```

Описание

`DEALLOCATE DESCRIPTOR` освобождает область именованного SQL-дескриптора.

Параметры

`имя`

Имя дескриптора, подлежащего освобождению, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

Примеры

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

Совместимость

Команда `DEALLOCATE DESCRIPTOR` описана в стандарте SQL.

См. также

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

DECLARE

DECLARE — определить курсор

Синтаксис

```
DECLARE имя_курсора [ BINARY ] [ ASENSITIVE | INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR
 [ { WITH | WITHOUT } HOLD ] FOR подготовленный_оператор
DECLARE имя_курсора [ BINARY ] [ ASENSITIVE | INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR
 [ { WITH | WITHOUT } HOLD ] FOR query
```

Описание

`DECLARE` объявляет курсор для прохода по набору результатов подготовленного оператора. Эта команда несколько отличается от обычной SQL-команды `DECLARE`: тогда как последняя выполняет запрос и подготавливает набор результатов для получения, встраиваемая SQL-команда просто объявляет имя в качестве «переменной цикла» для прохода по набору результатов запроса; фактически запрос выполнится, когда курсор будет открыт командой `OPEN`.

Параметры

имя_курсора

Имя курсора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

подготовленный_оператор

Имя подготовленного запроса, задаваемое SQL-идентификатором или переменной среды.

query

Команда SELECT или VALUES, выдающая строки, которые будут получены через курсор.

Параметры курсора рассматриваются в описании DECLARE.

Примеры

Примеры объявления курсора для запроса:

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT pgpro_version();
```

Пример объявления курсора для подготовленного оператора:

```
EXEC SQL PREPARE stmt1 AS SELECT pgpro_version();
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

Совместимость

Команда `DECLARE` описана в стандарте SQL.

См. также

OPEN, CLOSE, DECLARE

DECLARE STATEMENT

DECLARE STATEMENT — объявляет идентификатор SQL-оператора

Синтаксис

```
EXEC SQL [ AT имя_подключения ] DECLARE имя_оператора STATEMENT
```

Описание

`DECLARE STATEMENT` объявляет идентификатор SQL-оператора. Идентификатор SQL-оператора может быть связан с соединением. Когда этот идентификатор используется динамическими SQL-операторами, они выполняются через связанное с ними подключение. Пространством имён является единица предкомпиляции, и несколько объявлений с одним и тем же идентификатором SQL-оператора не допускаются. Обратите внимание, что если предкомпилятор работает в режиме совместимости с Informix и объявлен некоторый SQL-оператор, в качестве имени курсора нельзя использовать «`database`».

Параметры

`имя_подключения`

Имя подключения к базе данных устанавливается командой `CONNECT`.

Предложение AT можно опустить, но такой оператор не имеет смысла.

`имя_оператора`

Имя идентификатора SQL-оператора, которое может быть либо идентификатором SQL, либо переменной среды C.

Примечания

Эта связь действительна только в том случае, если объявление физически размещается сверху динамического оператора.

Примеры

```
EXEC SQL CONNECT TO postgres AS con1;
EXEC SQL AT con1 DECLARE sql_stmt STATEMENT;
EXEC SQL DECLARE cursor_name CURSOR FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
EXEC SQL OPEN cursor_name;
EXEC SQL FETCH cursor_name INTO :column1;
EXEC SQL CLOSE cursor_name;
```

Совместимость

`DECLARE STATEMENT` — реализованное в PostgreSQL расширение стандарта SQL, которое также может использоваться в других популярных СУБД.

См. также

CONNECT, DECLARE, OPEN

DESCRIBE

DESCRIBE — получить информацию о подготовленном операторе или наборе результатов

Синтаксис

```
DESCRIBE [ OUTPUT ] подготовленный_оператор USING [ SQL ] DESCRIPTOR имя_дескриптора
DESCRIBE [ OUTPUT ] подготовленный_оператор INTO [ SQL ] DESCRIPTOR имя_дескриптора
DESCRIBE [ OUTPUT ] подготовленный_оператор INTO имя_sqlda
```

Описание

`DESCRIBE` получает метаданные о результирующих столбцах, содержащихся в подготовленном операторе, не считывая собственно строки результата.

Параметры

`подготовленный_оператор`

Имя подготовленного оператора. Это может быть идентификатор SQL или переменная среды C.

`имя_дескриптора`

Имя дескриптора, задаётся с учётом регистра. Это может быть идентификатор SQL или переменная среды C.

`имя_sqlda`

Имя переменной SQLDA.

Примеры

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

Совместимость

Команда `DESCRIBE` описана в стандарте SQL.

См. также

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

DISCONNECT

DISCONNECT — закрыть подключение к базе данных

Синтаксис

```
DISCONNECT имя_подключения
DISCONNECT [ CURRENT ]
DISCONNECT ALL
```

Описание

`DISCONNECT` закрывает подключение (или все подключения) к базе данных.

Параметры

`имя_подключения`

Имя подключения к базе данных устанавливается командой `CONNECT`.

`CURRENT`

Закрывает «текущее» подключение, то есть подключение, открытое последним, либо установленное командой `SET CONNECTION`. Текущее подключение подразумевается по умолчанию, если `DISCONNECT` выполняется без аргументов.

`ALL`

Закрывает все открытые подключения.

Примеры

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;

    EXEC SQL DISCONNECT CURRENT;  /* закрывает con3          */
    EXEC SQL DISCONNECT ALL;      /* закрывает con2 и con1 */

    return 0;
}
```

Совместимость

Команда `DISCONNECT` описана в стандарте SQL.

См. также

CONNECT, SET CONNECTION

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE — динамически подготовить и выполнить оператор

Синтаксис

```
EXECUTE IMMEDIATE строка
```

Описание

`EXECUTE IMMEDIATE` немедленно подготавливает и выполняет динамически задаваемый SQL-оператор, не получая при этом строки результата.

Параметры

`строка`

Строковая константа или переменная среды C, содержащая SQL-оператор, который нужно выполнить.

Примечания

Обычно в качестве *`строки`* задаётся ссылка на переменную, содержащую динамически создаваемый SQL-оператор. Использовать здесь строковую константу не очень полезно; с тем же успехом можно написать непосредственно SQL-оператор, не добавляя лишнюю команду `EXECUTE IMMEDIATE`.

Если вы всё же используете строковую константу, имейте в виду, что включаемые в SQL-оператор двойные кавычки нужно записывать в виде восьмеричной спецпоследовательности (`\042`), а не в виде `\"`, как принято в C. Это объясняется тем, что данная строка находится внутри блока `EXEC SQL`, поэтому лексический анализатор ECPG разбирает её в соответствии с правилами SQL, а не C. Содержащиеся в этой строке символы обратной косой черты будут обработаны позже, по правилам C, но последовательность `\"` сразу вызовет ошибку синтаксиса, так как она будет воспринята как завершающая строку.

Примеры

Пример выполнения оператора `INSERT` с применением команды `EXECUTE IMMEDIATE` и переменной среды C `command`:

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: ''r1''', 1,
 'f')");
EXEC SQL EXECUTE IMMEDIATE :command;
```

Совместимость

Команда `EXECUTE IMMEDIATE` описана в стандарте SQL.

GET DESCRIPTOR

GET DESCRIPTOR — получить информацию из области дескриптора SQL

Синтаксис

```
GET DESCRIPTOR имя_дескриптора :cvariable = элемент_заголовка_дескриптора [, ... ]
GET DESCRIPTOR имя_дескриптора VALUE номер_столбца :cvariable = элемент_дескриптора
 [, ... ]
```

Описание

`GET DESCRIPTOR` получает информацию о наборе результатов запроса из области дескриптора SQL и сохраняет её в переменные среды. Область дескриптора обычно предварительно заполняется командами `FETCH` или `SELECT`, чтобы из неё можно было перенести сопутствующую информацию в переменные среды.

Эта команда имеет две формы: первая форма выдаёт элементы из «заголовка» дескриптора, который относится ко всему набору результатов в целом. Например, это число строк. Другая форма, требующая указания в дополнительном параметре номера столбца, выдаёт информацию о конкретном столбце строки. В качестве примеров можно привести имя столбца и фактическое значение в этом столбце.

Параметры

имя_дескриптора

Имя дескриптора.

элемент_заголовка_дескриптора

Идентификатор, определяющий, какой элемент заголовка нужно получить. В настоящее время поддерживается только `COUNT`, позволяющий получить число столбцов в наборе результатов.

номер_столбца

Номер столбца, информацию о котором нужно получить. Нумерация начинается с 1.

элемент_дескриптора

Идентификатор, определяющий, какой элемент информации о столбце нужно получить. Список поддерживаемых элементов приведён в Подразделе 34.7.1.

cvariable

Переменная среды C, в которую будут сохранены данные, полученные из области дескриптора.

Примеры

Пример получения числа столбцов в наборе результатов:

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

Пример получения размера данных в первом столбце:

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
```

Пример получения содержимого данных второго столбца в виде строки:

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

Следующий пример демонстрирует всю процедуру выполнения `SELECT current_database();` и вывода числа столбцов, длины данных в столбце и содержимого столбца:

```
int
```

```
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int  d_count;
    char d_data[1024];
    int  d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;

    /* Объявить, открыть курсор и присвоить ему дескриптор */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

    /* Получить общее число столбцов */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count                 = %d\n", d_count);

    /* Получить размер возвращённого столбца */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);

    /* Выбрать возвращённый столбец в виде текстовой строки */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data                  = %s\n", d_data);

    /* Закрытие */
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

При выполнении этого примера будет получен такой результат:

```
d_count                 = 1
d_returned_octet_length = 6
d_data                  = testdb
```

Совместимость

Команда `GET DESCRIPTOR` описана в стандарте SQL.

См. также

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

OPEN

OPEN — открыть динамический курсор

Синтаксис

```
OPEN имя_курсора
OPEN имя_курсора USING значение [, ... ]
OPEN имя_курсора USING SQL DESCRIPTOR имя_дескриптора
```

Описание

`OPEN` открывает курсор и в дополнение может связывать фактические значения с местозаполнителями в объявлении курсора. Курсор должен быть предварительно объявлен командой `DECLARE`. Команда `OPEN` запускает выполнение запроса на сервере.

Параметры

`имя_курсора`

Имя открываемого курсора. Этот может быть идентификатор SQL или переменная среды.

`значение`

Значение, связываемое с местозаполнителем в курсоре. Это может быть константа SQL, переменная среды C или переменная среды с индикатором.

`имя_дескриптора`

Имя дескриптора, содержащего значения, которые должны быть связаны с местозаполнителями в курсоре. Это может быть идентификатор SQL или переменная среды C.

Примеры

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

Совместимость

Команда `OPEN` описана в стандарте SQL.

См. также

DECLARE, CLOSE

PREPARE

PREPARE — подготовить оператор к выполнению

Синтаксис

```
PREPARE подготовленный_оператор FROM строка
```

Описание

Команда `PREPARE` подготавливает к выполнению динамический оператор, задаваемый в виде строки. Она отличается от обычного SQL-оператора PREPARE, который также можно использовать во встраиваемых командах. Для обоих типов подготовленных операторов применяется команда EXECUTE.

Параметры

`подготовленный_оператор`

Идентификатор для подготовленного запроса.

`строка`

Строковая константа C или переменная среды C, содержащая один из подготавливаемых операторов: SELECT, INSERT, UPDATE или DELETE. Параметры, значения которых будут представлены при выполнении, обозначаются вопросительными знаками (`?`).

Примечания

Обычно в качестве *`строки`* задаётся ссылка на переменную, содержащую динамически создаваемый SQL-оператор. Использовать здесь строковую константу не очень полезно; с тем же успехом можно написать непосредственно SQL-оператор `PREPARE`.

Если вы всё же используете строковую константу, имейте в виду, что включаемые в SQL-оператор двойные кавычки нужно записывать в виде восьмеричной спецпоследовательности (`\042`), а не в виде `\"`, как принято в C. Это объясняется тем, что данная строка находится внутри блока `EXEC SQL`, поэтому лексический анализатор ECPG разбирает её в соответствии с правилами SQL, а не C. Содержащиеся в этой строке символы обратной косой черты будут обработаны позже, по правилам C, но последовательность `\"` сразу вызовет ошибку синтаксиса, так как она будет воспринята как завершающая строку.

Примеры

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

Совместимость

Команда `PREPARE` описана в стандарте SQL.

См. также

EXECUTE

SET AUTOCOMMIT

SET AUTOCOMMIT — установить режим автофиксации для текущего сеанса

Синтаксис

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

Описание

`SET AUTOCOMMIT` устанавливает режим автофиксации для текущего сеанса использования базы данных. По умолчанию программы со встраиваемым SQL работают *не* в режиме автофиксации, так что в определённые моменты нужно явно выполнять `COMMIT`. Эта команда может переключить сеанс в режим автофиксации, когда неявно фиксируется каждый отдельный оператор.

Совместимость

`SET AUTOCOMMIT` — расширение Postgres Pro ECPG.

SET CONNECTION

SET CONNECTION — выбрать подключение к базе данных

Синтаксис

```
SET CONNECTION [ TO | = ] имя_подключения
```

Описание

`SET CONNECTION` устанавливает «текущее» подключение к базе данных, которое будет использоваться командами, не задающими подключение явно.

Параметры

`имя_подключения`

Имя подключения к базе данных устанавливается командой `CONNECT`.

`CURRENT`

Устанавливает заданное подключение текущим подключением (то есть ничего не происходит).

Примеры

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

Совместимость

Команда `SET CONNECTION` описана в стандарте SQL.

См. также

CONNECT, DISCONNECT

SET DESCRIPTOR

SET DESCRIPTOR — внести информацию в область дескриптора SQL

Синтаксис

```
SET DESCRIPTOR имя_дескриптора элемент_заголовка_дескриптора = значение [, ... ]
SET DESCRIPTOR имя_дескриптора VALUE номер элемент_дескриптора = значение [, ...]
```

Описание

`SET DESCRIPTOR` заполняет область SQL-дескриптора значениями. Заполненная область дескриптора обычно применяется для привязывания параметров при выполнении подготовленного запроса.

Эта команда имеет две формы: первая применяется к «заголовку» дескриптора, который не зависит от конкретных данных. Вторая форма устанавливает значения для определённых полей по номерам.

Параметры

имя_дескриптора

Имя дескриптора.

элемент_заголовка_дескриптора

Идентификатор, определяющий, какой элемент заголовка нужно задать. В настоящее время поддерживается только `COUNT`, позволяющий задать число элементов в дескрипторе.

номер

Номер элемента дескриптора, для которого задаётся значение. Нумерация начинается с 1.

элемент_дескриптора

Идентификатор, определяющий, какой элемент нужно установить в дескрипторе. Список поддерживаемых элементов приведён в Подразделе 34.7.1.

значение

Значение, которое нужно поместить в элемент дескриптора. Это может быть константа SQL или переменная среды C.

Примеры

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

Совместимость

Команда `SET DESCRIPTOR` описана в стандарте SQL.

См. также

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

TYPE

TYPE — создать новый тип данных

Синтаксис

```
TYPE имя_типа IS тип_C
```

Описание

Команда `TYPE` определяет новый тип C. Она равнозначна добавлению `typedef` в секции объявлений.

Эта команда принимается, только когда `ecpg` запускается с параметром `-c`.

Параметры

`имя_типа`

Имя нового типа. Это имя должно быть допустимым для типа в языке C.

`тип_C`

Определение типа C.

Примеры

```
EXEC SQL TYPE customer IS
    struct
    {
        varchar name[50];
        int     phone;
    };

EXEC SQL TYPE cust_ind IS
    struct ind
    {
        short   name_ind;
        short   phone_ind;
    };

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

Пример программы, в которой используется `EXEC SQL TYPE`:

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
    struct
    {
        varchar v[256];
        int     i;
    };

EXEC SQL TYPE tt_ind IS
    struct ind {
        short   v_ind;
        short   i_ind;
    };
```

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

Эта программа выдаёт следующее:

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

Совместимость

Команда `TYPE` — расширение Postgres Pro.

VAR

VAR — определить переменную

Синтаксис

```
VAR имя_переменной IS тип_C
```

Описание

Команда `VAR` назначает переменной среды C новый тип данных C. Эта переменная среды должна быть объявлена ранее в секции объявлений.

Параметры

имя_переменной

Имя переменной C.

тип_C

Определение типа C.

Примеры

```
Exec sql begin declare section;
short a;
exec sql end declare section;
EXEC SQL VAR a IS int;
```

Совместимость

Команда `VAR` — расширение Postgres Pro.

WHENEVER

WHENEVER — определить действие, которое должно выполняться, когда при обработке SQL-оператора возникает определённое условие

Синтаксис

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } действие
```

Описание

Устанавливает поведение в случае определённых условий (строки не найдены, выданы предупреждения или ошибки SQL и т. д.), возникающих в ходе выполнения SQL.

Параметры

Описание параметров приведено в Подразделе 34.8.1.

Примеры

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER NOT FOUND DO CONTINUE;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

Типичное применение конструкция `WHENEVER NOT FOUND BREAK` находит в обработке результатов запроса в цикле:

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;

    /* по достижении конца набора результатов прервать цикл while */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;
```

```
    return 0;
}
```

Совместимость

Команда `WHENEVER` описана в стандарте SQL, но большинство действий относятся к расширениям Postgres Pro.

34.15. Режим совместимости с Informix

Препроцессор `ecpg` может работать в так называемом *режиме совместимости с Informix*. Если этот режим включён, `ecpg` старается работать как предкомпилятор Informix для кода Informix E/SQL. Вообще говоря, это позволяет записывать встраиваемые команды SQL, используя знак доллара вместо слов `EXEC SQL`:

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

> **Примечание**
>
> Между `$` и последующей директивой препроцессора (в частности, `include`, `define`, `ifdef` и т. п.) не должно быть пробельных символов. В противном случае препроцессор воспримет следующее слово как имя переменной среды.

Поддерживаются два режима совместимости: `INFORMIX` и `INFORMIX_SE`

При компоновке программ, использующих этот режим совместимости, обязательно подключите библиотеку `libcompat`, поставляемую с ECPG.

Помимо ранее упомянутого синтаксического сахара, режим совместимости с Informix приносит из E/SQL в ECPG набор функций для ввода, вывода и преобразования данных, а также встраиваемые операторы SQL.

Режим совместимости с Informix тесно связан с библиотекой pgtypeslib из ECPG. Библиотека pgtypeslib сопоставляет типы данных SQL с типами данных в ведущей программе на C, а большинство дополнительных функций режима совместимости с Informix позволяют работать с этими типами C. Заметьте, однако, что степень совместимости ограничена. ECPG не пытается копировать поведение Informix; вы можете выполнять примерно те же операции и пользоваться функции с теми же именами и с тем же поведением, но если вы используете Informix, просто заменить одно средство другим на данный момент нельзя. Более того, есть различия и в типах данных. В частности, типы даты и интервала в Postgres Pro не воспринимают диапазоны, как например, `YEAR TO MINUTE`, так что и в ECPG это не будет поддерживаться.

34.15.1. Дополнительные типы

Теперь в режиме Informix без указания `typedef` поддерживается специальный псевдотип Informix "string" для хранения символьной строки, обрезаемой справа. На самом деле, в режиме Informix ECPG откажется обрабатывать исходные файлы, содержащие определение типа `typedef некоторый_тип string;`

```
EXEC SQL BEGIN DECLARE SECTION;
string userid; /* эта переменная будет содержать обрезанные данные */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

34.15.2. Дополнительные/недостающие операторы встраиваемого SQL

`CLOSE DATABASE`

Этот оператор закрывает текущее подключение. Фактически это синоним команды `DISCONNECT CURRENT` в ECPG:

```
$CLOSE DATABASE;                /* закрыть текущее подключение */
EXEC SQL CLOSE DATABASE;
```

`FREE имя_курсора`

Из-за различий в подходах ECPG и ESQL/C Informix (а именно другого разделения на чисто грамматические преобразования и вызовы нижележащей библиотеки времени выполнения) в ECPG нет оператора `FREE имя_курсора`. Это связано с тем, что в ECPG команда `DECLARE CURSOR` не сводится к вызову функции в библиотеке времени выполнения, которая бы принимала имя курсора. Это значит, что курсоры SQL в библиотеке ECPG не требуют обслуживания, оно требуется только на уровне сервера Postgres Pro.

`FREE имя_оператора`

Команда `FREE имя_оператора` является синонимом команды `DEALLOCATE PREPARE имя_оператора`.

34.15.3. Области дескрипторов SQLDA, совместимые с Informix

Режим совместимости с Informix поддерживает структуру, отличную от описанной в Подразделе 34.7.2. См. ниже:

```
struct sqlvar_compat
{
    short   sqltype;
    int     sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short   sqlitype;
    short   sqlilen;
    char   *sqlidata;
    int     sqlxid;
    char   *sqltypename;
    short   sqltypelen;
    short   sqlownerlen;
    short   sqlsourcetype;
    char   *sqlownername;
    int     sqlsourceid;
    char   *sqlilongdata;
    int     sqlflags;
    void   *sqlreserved;
};

struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void  *reserved;
```

```
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

Глобальные свойства:

`sqld`

Число полей в дескрипторе `SQLDA`.

`sqlvar`

Указатель на свойства по полям.

`desc_name`

Не используется, заполняется нулями.

`desc_occ`

Размер структуры в памяти.

`desc_next`

Указатель на следующую структуру SQLDA, если набор результатов содержит больше одной записи.

`reserved`

Неиспользуемый указатель, содержит NULL. Сохраняется для совместимости с Informix.

Свойства, относящиеся к полям, описаны ниже, они хранятся в массиве `sqlvar`:

`sqltype`

Тип поля. Соответствующие константы представлены в `sqltypes.h`

`sqllen`

Длина данных поля.

`sqldata`

Указатель на данные поля. Этот указатель имеет тип `char *`, но он указывает на данные в двоичном формате. Например:

```
int intval;

switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
  ...
}
```

`sqlind`

Указатель на индикатор NULL. Если возвращается командами DESCRIBE или FETCH, это всегда действительный указатель. Если передаётся на вход команде `EXECUTE ... USING sqlda;`, NULL вместо указателя означает, что значение этого поля отлично от NULL. Чтобы обозначить NULL в поле, необходимо корректно установить этот указатель и `sqlitype`. Например:

```
if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
```

```
      printf("value is NULL\n");
```

`sqlname`

Имя поля, в виде строки с завершающим 0.

`sqlformat`

Зарезервировано в Informix, значение `PQfformat` для данного поля.

`sqlitype`

Тип данных индикатора NULL. При получении данных с сервера это всегда SQLSMINT. Когда `SQLDA` используется в параметризованном запросе, данные индикатора обрабатываются в соответствии с указанным здесь типом.

`sqlilen`

Длина данных индикатора NULL.

`sqlxid`

Расширенный тип поля, результат функции `PQftype`.

`sqltypename`
`sqltypelen`
`sqlownerlen`
`sqlsourcetype`
`sqlownername`
`sqlsourceid`
`sqlflags`
`sqlreserved`

Не используются.

`sqlilongdata`

Совпадает с `sqldata`, если `sqllen` превышает 32 Кбайта.

Например:

```
EXEC SQL INCLUDE sqlda.h;

    sqlda_t        *sqlda; /* Это объявление не обязательно должно быть внутри DECLARE
 SECTION */

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL PREPARE mystmt FROM :prep_stmt;

    EXEC SQL DESCRIBE mystmt INTO sqlda;

    printf("# of fields: %d\n", sqlda->sqld);
    for (i = 0; i < sqlda->sqld; i++)
      printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);

    EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
    EXEC SQL OPEN mycursor;
```

```
EXEC SQL WHENEVER NOT FOUND GOTO out;

while (1)
{
  EXEC SQL FETCH mycursor USING sqlda;
}

EXEC SQL CLOSE mycursor;

free(sqlda); /* Освобождать нужно только основную структуру,
              * sqlda и sqlda->sqlvar находятся в одной выделенной области. */
```

34.15.4. Дополнительные функции

`decadd`

Складывает два значения типа decimal.

`int decadd(decimal *arg1, decimal *arg2, decimal *sum);`

Эта функция получает указатель на первый операнд типа decimal (`arg1`), указатель на второй операнд типа decimal (`arg2`) и указатель на переменную типа decimal, в которую будет записана сумма (`sum`). В случае успеха эта функция возвращает 0. `ECPG_INFORMIX_NUM_OVERFLOW` возвращается в случае переполнения, а `ECPG_INFORMIX_NUM_UNDERFLOW` в случае антипереполнения. При любых других ошибках возвращается -1, а в `errno` устанавливается код `errno` из pgtypeslib.

`deccmp`

Сравнивает два значения типа decimal.

`int deccmp(decimal *arg1, decimal *arg2);`

Эта функция получает указатель на первое значение типа decimal (`arg1`), указатель на второе значение типа decimal (`arg2`) и возвращает целое, отражающее результат сравнения этих чисел.

- 1, если значение, на которое указывает `arg1`, больше значения, на которое указывает `var2`
- -1, если значение, на которое указывает `arg1`, меньше значения, на которое указывает `arg2`
- 0, если значение, на которое указывает `arg1`, равно значению, на которое указывает `arg2`

`deccopy`

Копирует значение типа decimal.

`void deccopy(decimal *src, decimal *target);`

Функция принимает в первом аргументе (`src`) указатель на значение decimal, которое должно быть скопировано, а во втором аргументе (`target`) принимает указатель на структуру типа decimal для скопированного значения.

`deccvasc`

Преобразует значение из представления ASCII в тип decimal.

`int deccvasc(char *cp, int len, decimal *np);`

Эта функция получает указатель на строку, содержащую строковое представление числа, которое нужно преобразовать, (`cp`), а также его длину `len`. В `np` передаётся указатель на переменную типа decimal, в которую будет помещён результат преобразования.

Допустимыми являются, например следующие форматы: `-2`, `.794`, `+3.44`, `592.49E07` или `-32.84e-4`.

В случае успеха эта функция возвращает 0. При переполнении или антипереполнении возвращается `ECPG_INFORMIX_NUM_OVERFLOW` или `ECPG_INFORMIX_NUM_UNDERFLOW`, соответственно. Если разобрать ASCII-представление не удаётся, возвращается `ECPG_INFORMIX_BAD_NUMERIC` или `ECPG_INFORMIX_BAD_EXPONENT`, если не удаётся разобрать компонент экспоненты.

`deccvdbl`

Преобразует значение double в значение типа decimal.

```
int deccvdbl(double dbl, decimal *np);
```

Данная функция принимает в первом аргументе (`dbl`) переменную типа double, которая должна быть преобразована. Во втором аргументе (`np`) она принимает указатель на переменную decimal, в которую будет помещён результат операции.

Эта функция возвращает 0 в случае успеха, либо отрицательное значение, если выполнить преобразование не удалось.

`deccvint`

Преобразует значение int в значение типа decimal.

```
int deccvint(int in, decimal *np);
```

Данная функция принимает в первом аргументе (`in`) переменную типа int, которая должна быть преобразована. Во втором аргументе (`np`) она принимает указатель на переменную decimal, в которую будет помещён результат операции.

Эта функция возвращает 0 в случае успеха, либо отрицательное значение, если выполнить преобразование не удалось.

`deccvlong`

Преобразует значение long в значение типа decimal.

```
int deccvlong(long lng, decimal *np);
```

Данная функция принимает в первом аргументе (`lng`) переменную типа long, которая должна быть преобразована. Во втором аргументе (`np`) она принимает указатель на переменную decimal, в которую будет помещён результат операции.

Эта функция возвращает 0 в случае успеха, либо отрицательное значение, если выполнить преобразование не удалось.

`decdiv`

Делит одну переменную типа decimal на другую.

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

Эта функция получает указатели на переменные (`n1` и `n2`) и вычисляет частное `n1`/`n2`. В `result` передаётся указатель на переменную, в которую будет помещён результат операции.

В случае успеха возвращается 0, а при ошибке — отрицательное значение. В случае переполнения или антипереполнения данная функция возвращает `ECPG_INFORMIX_NUM_OVERFLOW` или `ECPG_INFORMIX_NUM_UNDERFLOW`, соответственно. При попытке деления на ноль возвращается `ECPG_INFORMIX_DIVIDE_ZERO`.

`decmul`

Перемножает два значения типа decimal.

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

Эта функция получает указатели на переменные (`n1` и `n2`) и вычисляет произведение `n1`*`n2`. В `result` передаётся указатель на переменную, в которую будет помещён результат операции.

В случае успеха возвращается 0, а при ошибке — отрицательное значение. В случае переполнения или антипереполнения данная функция возвращает `ECPG_INFORMIX_NUM_OVERFLOW` или `ECPG_INFORMIX_NUM_UNDERFLOW`, соответственно.

`decsub`

Вычитает одно значение типа decimal из другого.

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

Эта функция получает указатели на переменные (`n1` и `n2`) и вычисляет разность `n1`-`n2`. В `result` передаётся указатель на переменную, в которую будет помещён результат операции.

В случае успеха возвращается 0, а при ошибке — отрицательное значение. В случае переполнения или антипереполнения данная функция возвращает `ECPG_INFORMIX_NUM_OVERFLOW` или `ECPG_INFORMIX_NUM_UNDERFLOW`, соответственно.

`dectoasc`

Преобразует переменную типа decimal в представление ASCII (в строку C char*).

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

Эта функция получает указатель на переменную типа decimal (`np`), которая будет преобразована в текстовое представление. Аргумент `cp` указывает на буфер, в который будет помещён результат операции. Аргумент `right` определяет, сколько должно выводиться цифр правее десятичной точки. Результат будет округлён до этого числа десятичных цифр. Значение `right`, равное -1, указывает, что выводиться должны все имеющиеся десятичные цифры. Если длина выходного буфера, которую задаёт `len`, недостаточна для помещения в него текстового представления, включая завершающий нулевой байт, в буфере сохраняется один знак `*` и возвращается -1.

Эта функция возвращает -1, если буфер `cp` слишком мал, либо `ECPG_INFORMIX_OUT_OF_MEMORY` при нехватке памяти.

`dectodbl`

Преобразует переменную типа decimal в тип double.

```
int dectodbl(decimal *np, double *dblp);
```

Эта функция получает указатель (`np`) на значение decimal, которое нужно преобразовать, и указатель (`dblp`) на переменную double, в которую будет помещён результат операции.

В случае успеха возвращается 0, или отрицательное значение, если выполнить преобразование не удалось.

`dectoint`

Преобразует переменную типа decimal в тип integer.

```
int dectoint(decimal *np, int *ip);
```

Эта функция получает указатель (`np`) на значение decimal, которое нужно преобразовать, и указатель (`ip`) на целочисленную переменную, в которую будет помещён результат операции.

В случае успеха возвращается 0, или отрицательное значение, если выполнить преобразование не удалось. В случае переполнения возвращается `ECPG_INFORMIX_NUM_OVERFLOW`.

Заметьте, что реализация ECPG отличается от реализации Informix. В Informix целое ограничивается диапазоном -32767 .. 32767, тогда как в ECPG ограничение зависит от архитектуры (`INT_MIN .. INT_MAX`).

`dectolong`

Преобразует переменную типа decimal в тип long.

```
int dectolong(decimal *np, long *lngp);
```

Эта функция получает указатель (`np`) на значение decimal, которое нужно преобразовать, и указатель (`lngp`) на переменную типа long, в которую будет помещён результат операции.

В случае успеха возвращается 0, или отрицательное значение, если выполнить преобразование не удалось. В случае переполнения возвращается `ECPG_INFORMIX_NUM_OVERFLOW`.

Заметьте, что реализация ECPG отличается от реализации Informix. В Informix длинное целое ограничено диапазоном -2 147 483 647 .. 2 147 483 647, тогда как в ECPG ограничение зависит от архитектуры (`-LONG_MAX .. LONG_MAX`).

`rdatestr`

Преобразует дату в строку C char*.

```
int rdatestr(date d, char *str);
```

Эта функция принимает два аргумента. В первом (`d`) передаётся дата, которую нужно преобразовать, а во втором указатель на целевую строку. Результат всегда выводится в формате `yyyy-mm-dd`, так что для этой строки нужно выделить минимум 11 байт (включая завершающий нулевой байт).

Эта функция возвращает 0 в случае успеха, а в случае ошибки — отрицательное значение.

Заметьте, что реализация ECPG отличается от реализации Informix. В Informix формат вывода можно изменить переменными окружения, а в ECPG он фиксирован.

`rstrdate`

Разбирает текстовое представление даты.

```
int rstrdate(char *str, date *d);
```

Эта функция получает текстовое представление (`str`) даты, которую нужно преобразовать, и указатель на переменную типа date (`d`). Для данной функции нельзя задать маску формата. Она использует стандартную маску формата Informix, а именно: `mm/dd/yyyy`. Внутри эта функция вызывает `rdefmtdate`. Таким образом, `rstrdate` не будет быстрее, и если у вас есть выбор, используйте функцию `rdefmtdate`, которая позволяет явно задать маску формата.

Эта функция возвращает те же значения, что и `rdefmtdate`.

`rtoday`

Выдаёт текущую дату.

```
void rtoday(date *d);
```

Эта функция получает указатель на переменную (`d`) типа date, в которую будет записана текущая дата.

Внутри эта функция вызывает `PGTYPESdate_today`.

`rjulmdy`

Извлекает значения дня, месяца и года из переменной типа date.

```
int rjulmdy(date d, short mdy[3]);
```

Эта функция получает дату `d` и указатель на 3 коротких целочисленных значения `mdy`. Имя переменной указывает на порядок значений: в `mdy[0]` записывается номер месяца, в `mdy[1]` — номер дня, а в `mdy[2]` — год.

В текущем состоянии эта функция всегда возвращает 0.

Внутри эта функция вызывает `PGTYPESdate_julmdy`.

`rdefmtdate`

Преобразует символьную строку в значение типа date по маске формата.

```
int rdefmtdate(date *d, char *fmt, char *str);
```

Эта функция принимает указатель на переменную типа date (`d`), в которую будет помещён результат операции, маску формата для разбора даты (`fmt`) и строку C char*, содержащую текстовое представление даты (`str`). Ожидается, что текстовое представление будет соответствовать маске формата. Однако это соответствие не обязательно должно быть точным. Данная функция анализирует только порядок элементов и ищет в нём подстроки `yy` или `yyyy`, обозначающие позицию года, подстроку `mm`, обозначающую позицию месяца, и `dd`, обозначающую позицию дня.

Эта функция возвращает следующие значения:

- 0 — Функция выполнена успешно.
- `ECPG_INFORMIX_ENOSHORTDATE` — Дата не содержит разделителей между днём, месяцем и годом. С таким форматом входная строка должна быть длиной ровно 6 или 8 байт, но это не так.
- `ECPG_INFORMIX_ENOTDMY` — Строка формата не определяет корректно последовательный порядок года, месяца и дня.
- `ECPG_INFORMIX_BAD_DAY` — Во входной строке отсутствует корректное указание дня.
- `ECPG_INFORMIX_BAD_MONTH` — Во входной строке отсутствует корректное указание месяца.
- `ECPG_INFORMIX_BAD_YEAR` — Во входной строке отсутствует корректное указание года.

В реализации этой функции вызывается `PGTYPESdate_defmt_asc`. Примеры вводимых строк приведены в таблице в её описании.

`rfmtdate`

Преобразует переменную типа date в текстовое представление по маске формата.

```
int rfmtdate(date d, char *fmt, char *str);
```

Эта функция принимает дату для преобразования (`d`), маску формата (`fmt`) и строку, в которую будет помещено текстовое представление даты (`str`).

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

Внутри эта функция вызывает `PGTYPESdate_fmt_asc`, примеры форматов можно найти в её описании.

`rmdyjul`

Образует значение даты из массива 3 коротких целых, задающих день, месяц и год даты.

```
int rmdyjul(short mdy[3], date *d);
```

Эта функция получает в первом аргументе массив из 3 коротких целых (`mdy`), а во втором указатель на переменную типа date, в которую будет помещён результат операции.

В настоящее время эта функция всегда возвращает 0.

В реализации этой функции вызывается `PGTYPESdate_mdyjul`.

`rdayofweek`

Возвращает число, представляющее день недели для заданной даты.

```
int rdayofweek(date d);
```

Эта функция принимает в единственном аргументе переменную `d` типа date и возвращает целое число, выражающее день недели для этой даты.

- 0 — Воскресенье
- 1 — Понедельник
- 2 — Вторник
- 3 — Среда
- 4 — Четверг
- 5 — Пятница
- 6 — Суббота

В реализации этой функции вызывается `PGTYPESdate_dayofweek`.

`dtcurrent`

Получает текущее время.

```
void dtcurrent(timestamp *ts);
```

Эта функция получает текущее время и сохраняет его в переменной типа timestamp, на которую указывает `ts`.

`dtcvasc`

Разбирает время из текстового представления в переменную типа timestamp.

```
int dtcvasc(char *str, timestamp *ts);
```

Эта функция получает строку (`str`), которую нужно разобрать, и указатель на переменную типа timestamp, в которую будет помещён результат операции (`ts`).

Эта функция возвращает 0 в случае успеха, а в случае ошибки — отрицательное значение.

Внутри эта функция вызывает `PGTYPEStimestamp_from_asc`. Примеры вводимых строк приведены в таблице в её описании.

`dtcvfmtasc`

Разбирает время из текстового представления в переменную типа timestamp по маске формата.

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

Эта функция получает строку (`inbuf`), которую нужно разобрать, маску формата (`fmtstr`) и указатель на переменную timestamp, в которой будет содержаться результат операции (`dtvalue`).

В реализации этой функции используется `PGTYPEStimestamp_defmt_asc`. Список допустимых кодов формата приведён в её описании.

Эта функция возвращает 0 в случае успеха, а в случае ошибки — отрицательное значение.

`dtsub`

Вычитает одно значение времени из другого и возвращает переменную типа interval.

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

Эта функция вычитает значение timestamp, на которое указывает `ts2`, из значения timestamp, на которое указывает `ts1`, и сохраняет результат в переменной типа interval, на которую указывает `iv`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`dttoasc`

Преобразует переменную типа timestamp в строку C char*.

```
int dttoasc(timestamp *ts, char *output);
```

Эта функция получает указатель (`ts`) на переменную типа timestamp, которую нужно преобразовать, и строку (`output`) для сохранения результата операции. Она преобразует `ts` в текстовое представление согласно стандарту SQL, то есть по маске `YYYY-MM-DD HH:MM:SS`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`dttofmtasc`

Преобразует переменную типа timestamp в строку C char* по маске формата.

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

Эта функция получает в первом аргументе (`ts`) указатель на переменную типа timestamp, а в последующих указатель на буфер вывода (`output`), максимальную длину строки, которую может принять буфер (`str_len`), и маску формата, с которой будет выполняться преобразование (`fmtstr`).

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

Внутри эта функция использует `PGTYPEStimestamp_fmt_asc`. Примеры допустимых масок формата можно найти в её описании.

`intoasc`

Преобразует переменную типа interval в строку C char*.

```
int intoasc(interval *i, char *str);
```

Эта функция получает указатель (`i`) на переменную типа interval, которую нужно преобразовать, и строку (`str`) для сохранения результата операции. Она преобразует `i` в текстовое представление согласно стандарту SQL, то есть по маске `YYYY-MM-DD HH:MM:SS`.

В случае успеха возвращается 0, а в случае ошибки — отрицательное значение.

`rfmtlong`

Преобразует длинное целое в текстовое представление по маске формата.

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

Эта функция принимает значение типа long (`lng_val`), маску формата (`fmt`) и указатель на выходной буфер (`outbuf`). Она преобразует длинное целое в его текстовое представление согласно заданной маске формата.

Маску формата можно составить из следующих символов, определяющих формат:

- `*` (звёздочка) — если в данной позиции будет пусто, заполнить её звёздочкой.
- `&` (амперсанд) — если в данной позиции будет пусто, заполнить её нулём.
- `#` — заменить ведущие нули пробелами.
- `<` — выровнять число в строке по левой стороне.
- `,` (запятая) — сгруппировать числа, содержащие четыре и более цифр, в группы по три цифры через запятую.
- `.` (точка) — этот символ отделяет целую часть числа от дробной.
- `-` (минус) — с отрицательным числом должен выводиться знак минус.
- `+` (плюс) — с положительным числом должен выводиться знак плюс.
- `(` — это символ заменяет знак минус перед отрицательным числом. Сам знак минус выводиться не будет.
- `)` — этот символ заменяет минус и выводится после отрицательного числа.
- `$` — символ денежной суммы.

`rupshift`

Приводит строку к верхнему регистру.

```
void rupshift(char *str);
```

Эта функция получает указатель на строку и приводит в ней каждый символ в нижнем регистре к верхнему регистру.

`byleng`

Возвращает число символов в строке, не считая завершающих пробелов.

```
int byleng(char *str, int len);
```

Эта функция принимает в первом аргументе (`str`) строку фиксированной длины, а во втором (`len`) её длину. Она возвращает число значимых символов, то есть длину строки без завершающих пробелов.

`ldchar`

Копирует строку фиксированной длины в строку с завершающим нулём.

```
void ldchar(char *src, int len, char *dest);
```

Эта функция принимает строку фиксированной длины (`src`), которую нужно скопировать, её длину (`len`) и указатель на целевой буфер в памяти (`dest`). Учтите, что для буфера, на который указывает `dest`, необходимо выделить как минимум `len+1` байт. Данная функция копирует в новую область не больше `len` байт (меньше, если в исходной строке есть завершающие пробелы) и добавляет завершающий 0.

`rgetmsg`

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

Эта функция определена, но не реализована на данный момент!

`rtypalign`

```
int rtypalign(int offset, int type);
```

Эта функция определена, но не реализована на данный момент!

`rtypmsize`

```
int rtypmsize(int type, int len);
```

Эта функция определена, но не реализована на данный момент!

`rtypwidth`

```
int rtypwidth(int sqltype, int sqllen);
```

Эта функция определена, но не реализована на данный момент!

`rsetnull`

Присваивает переменной NULL.

```
int rsetnull(int t, char *ptr);
```

Эта функция получает целое, определяющее тип переменной, и указатель на саму переменную, приведённый к указателю C char*.

Определены следующие типы:

- `CCHARTYPE` — для переменной типа `char` или `char*`
- `CSHORTTYPE` — для переменной типа `short int`
- `CINTTYPE` — для переменной типа `int`

- `CBOOLTYPE` — для переменной типа `boolean`
- `CFLOATTYPE` — для переменной типа `float`
- `CLONGTYPE` — для переменной типа `long`
- `CDOUBLETYPE` — для переменной типа `double`
- `CDECIMALTYPE` — для переменной типа `decimal`
- `CDATETYPE` — для переменной типа `date`
- `CDTIMETYPE` — для переменной типа `timestamp`

Примеры вызова этой функции:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

`risnull`

Проверяет содержимое переменной на NULL.

```
int risnull(int t, char *ptr);
```

Эта функция получает тип проверяемой переменной (`t`), а также указатель на неё (`ptr`). Заметьте, что этот указатель нужно привести к char*. Список возможных типов переменных приведён в описании функции `rsetnull`.

Примеры использования этой функции:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

34.15.5. Дополнительные константы

Заметьте, что все эти константы относятся к ошибкам и все они представлены отрицательными значениями. Из описаний различных констант вы также можете узнать, какими именно числами они представлены в текущей реализации. Однако полагаться на эти числа не следует. Тем не менее вы можете рассчитывать на то, что все эти значения будут отрицательными.

`ECPG_INFORMIX_NUM_OVERFLOW`

Функции возвращают это значение, если при вычислении происходит переполнение. Внутри оно представляется числом -1200 (определение Informix).

`ECPG_INFORMIX_NUM_UNDERFLOW`

Функции возвращают это значение, если при вычислении происходит антипереполнение. Внутри оно представляется числом -1201 (определение Informix).

`ECPG_INFORMIX_DIVIDE_ZERO`

Функции возвращают это значение при попытке деления на ноль. Внутри оно представляется числом -1202 (определение Informix).

`ECPG_INFORMIX_BAD_YEAR`

Функции возвращают это значение, если при разборе даты встретилось некорректное указание года. Внутри оно представляется числом -1204 (определение Informix).

`ECPG_INFORMIX_BAD_MONTH`

Функции возвращают это значение, если при разборе даты встретилось некорректное указание месяца. Внутри оно представляется числом -1205 (определение Informix).

`ECPG_INFORMIX_BAD_DAY`

Функции возвращают это значение, если при разборе даты встретилось некорректное указание дня. Внутри оно представляется числом -1206 (определение Informix).

`ECPG_INFORMIX_ENOSHORTDATE`

Функции возвращают это значение, если процедуре разбора даты требуется короткая запись даты, но строка даты имеет неподходящую длину. Внутри оно представляется числом -1209 (определение Informix).

`ECPG_INFORMIX_DATE_CONVERT`

Функции возвращают это значение, если при форматировании даты происходит ошибка. Внутри оно представляется числом -1210 (определение Informix).

`ECPG_INFORMIX_OUT_OF_MEMORY`

Функции возвращают это значение, если им не хватает памяти для выполнения операций. Внутри оно представляется числом -1211 (определение Informix).

`ECPG_INFORMIX_ENOTDMY`

Функции возвращают это значение, если процедура разбора должна была получить маску формата (например, `mmddyy`), но не все поля были записаны правильно. Внутри оно представляется числом -1212 (определение Informix).

`ECPG_INFORMIX_BAD_NUMERIC`

Функции возвращают это значение, если процедура разбора не может получить числовое значение из текстового представления, потому что оно некорректно, либо если процедура вычисления не может произвести операцию с числовыми переменными из-за недопустимого значения минимум одной из этих переменных. Внутри оно представляется числом -1213 (определение Informix).

`ECPG_INFORMIX_BAD_EXPONENT`

Функции возвращают это значение, если процедура разбора не может воспринять экспоненту в числе. Внутри оно представляется числом -1216 (определение Informix).

`ECPG_INFORMIX_BAD_DATE`

Функции возвращают это значение, если процедура разбора не может разобрать дату. Внутри оно представляется числом -1218 (определение Informix).

`ECPG_INFORMIX_EXTRA_CHARS`

Функции возвращают это значение, если процедуре разбора передаются посторонние символы, которая она не может разобрать. Внутри оно представляется числом -1264 (определение Informix).

34.16. Режим совместимости с Oracle

Препроцессор `ecpg` может работать в так называемом *режиме совместимости с Oracle*. Если этот режим включён, `ecpg` старается работать как Pro*C в Oracle.

В этом режиме изменяются три аспекта работы `ecpg`:

- Массивы символов, принимающие строки символов, дополняются завершающими пробелами до указанной длины.
- Завершаются эти массивы нулевым байтом, а если происходит усечение строки, устанавливается индикатор.
- Когда в массив символов поступает пустая строка, индикатор NULL принимает значение `-1`.

34.17. Внутреннее устройство

В этом разделе рассказывается, как препроцессор ECPG устроен внутри. Эта информация может оказаться полезной для пользователей, желающих понять, как использовать ECPG.

Первые четыре строки, которые `ecpg` записывает в вывод, фиксированы. Первые две строки содержат комментарии, а следующие две директивы включения, подключающие интерфейс к библиотеке. Затем препроцессор прочитывает файл и продолжает запись в вывод. Обычно он просто печатает всё в устройство вывода.

Встречая команду `EXEC SQL`, он вмешивается и изменяет её. Данная команда начинается со слов `EXEC SQL` и заканчивается знаком `;`. Всё между ними воспринимается как оператор SQL и разбирается для подстановки переменных.

Подстановка переменных имеет место, когда символ начинается с двоеточия (`:`). ECPG будет искать переменную с таким именем среди переменных, ранее объявленных в секции `EXEC SQL DECLARE`.

Самая важная функция в библиотеке — `ECPGdo`, которая осуществляет выполнение большинства команд. Она принимает переменное число аргументов (это число легко может достигать 50, и мы надеемся, что это не приведёт к проблемам ни на какой платформе).

Ей передаются следующие аргументы:

Номер строки

Номер исходной строки; используется только в сообщениях об ошибках.

Строка

Команда SQL, которая должна быть выполнена. На её содержимое влияют входные переменные, то есть переменные, добавленные в команду, но неизвестные во время компиляции. Места, в которые должны вставляться переменные, обозначаются знаками `?`.

Входные переменные

Для каждой входной переменной формируются десять аргументов. (См. ниже.)

ECPGt_EOIT

Перечисление (`enum`), показывающее, что больше входных переменных нет.

Выходные переменные

Для каждой входной переменной формируются десять аргументов. (См. ниже.) Эти переменные заполняются данной функцией.

ECPGt_EORT

Перечисление (`enum`), показывающее, что больше выходных переменных нет.

Для каждой переменной, включённой в команду SQL, эта функция принимает десять аргументов:

1. Тип в виде специального символа.
2. Указатель на значение или указатель на указатель.

3. Размер переменной, если она имеет тип `char` или `varchar`.
4. Число элементов в массиве (при выборке данных в массив).
5. Смещение следующего элемента в массиве (при выборке данных в массив).
6. Тип переменной-индикатора в виде специального символа.
7. Указатель на переменную-индикатор.
8. 0
9. Число элементов в массиве индикаторов (при выборке данных в массив).
10. Смещение следующего элемента в массиве индикаторов (при выборке данных в массив).

Заметьте, что не все команды SQL обрабатываются таким образом. Например, команда открытия курсора вида:

```
EXEC SQL OPEN курсор;
```

не копируется в вывод. Вместо этого в позиции команды `OPEN` применяется команда `DECLARE` этого курсора, так как на самом деле курсор открывает она.

Ниже показан полный пример, демонстрирующий результат обработки препроцессором файла `foo.pgc` (детали могут меняться от версии к версии препроцессора):

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

преобразуется в:

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(Отступы здесь добавлены для читаемости, препроцессор их не вставляет.)

Глава 35. Информационная схема

Информационная схема состоит из набора представлений, содержащих информацию об объектах, определённых в текущей базе данных. Информационная схема описана в стандарте SQL и поэтому можно рассчитывать на её переносимость и стабильность — в отличие от системных каталогов, которые привязаны к Postgres Pro, и моделируются, отталкиваясь от реализации. Представления информационной схемы, однако, не содержат информацию о функциях, присущих исключительно Postgres Pro; чтобы получить информацию о них, необходимо обратиться к системным каталогам или другим специфическим представлениям Postgres Pro.

Примечание

Когда из базы данных запрашивается информация об ограничениях, возможна ситуация, когда соответствующий стандарту запрос, который должен возвращать одну строку, возвращает несколько. Это связано с тем, что стандарт SQL требует, чтобы имена ограничений были уникальными в схеме, но в Postgres Pro такого требования нет. Имена ограничений, которые Postgres Pro генерирует автоматически, не должны дублироваться в одной схеме, но сами пользователи могут назначить подобные дублирующиеся имена.

Эта проблема может проявиться при обращении к таким представлениям информационной схемы, как `check_constraint_routine_usage`, `check_constraints`, `domain_constraints` и `referential_constraints`. В некоторых других представлениях она могла бы тоже иметь место, но они содержат имя таблицы, помогающее различить дублирующиеся строки, например: `constraint_column_usage`, `constraint_table_usage`, `table_constraints`.

35.1. Схема

Информационная схема сама по себе — это схема с именем `information_schema`. Данная схема автоматически доступна во всех базах данных. Владельцем этой схемы является начальный пользователь баз данных в кластере, и этот пользователь, естественно, имеет все права в ней, включая возможность её удалить (хотя достигаемая при этом экономия пространства минимальна).

По умолчанию информационная схема отсутствует в пути поиска схем, так что ко всем объектам в ней нужно обращаться по полным именам. Так как имена некоторых объектов в информационной схеме довольно распространённые и могут встречаться и в пользовательских приложениях, будьте осторожны, добавляя информационную схему в путь поиска.

35.2. Типы данных

Столбцы в представлениях информационной схемы имеют специальные типы данных, определённые в информационной схеме. Они определены как простые домены поверх обычных встроенных типов. Задействовать эти типы вне информационной схемы не следует, но тем не менее приложения, выбирающие данные из информационной схеме должны быть готовы работать с ними.

Это следующие типы:

`cardinal_number`

Неотрицательное целое.

`character_data`

Строка символов (без определённого ограничения по длине).

`sql_identifier`

Строка символов. Этот тип применяется для идентификаторов SQL, тогда как тип `character_data` для всех остальных видов текстовых данных.

`time_stamp`

Домен на базе типа `timestamp with time zone` `yes_or_no`

Домен символьной строки, который принимает либо `YES`, либо `NO`. Этот домен применяется для представления булевых данных (истина/ложь, true/false) в информационной схеме. (Информационная схема была введена до появления в стандарте SQL типа `boolean`, поэтому данный домен необходим для сохранения обратной совместимости информационной схемы.)

Все столбцы в информационной схеме имеют один из этих пяти типов.

35.3. information_schema_catalog_name

Таблица `information_schema_catalog_name` всегда содержит одну строку и один столбец с именем текущей базы данных (текущий каталог, в терминологии SQL).

Таблица 35.1. Столбцы `information_schema_catalog_name`

Тип столбца	Описание
`catalog_name sql_identifier`	Имя базы данных, содержащей эту информационную схему

35.4. administrable_role_authorizations

Представление `administrable_role_authorizations` описывает все роли, для которых текущий пользователь является администратором.

Таблица 35.2. Столбцы `administrable_role_authorizations`

Тип столбца	Описание
`grantee sql_identifier`	Имя роли, которой было разрешено участие в целевой роли (может быть текущий пользователь, либо другая роль, в случае вложенного членства)
`role_name sql_identifier`	Имя целевой роли
`is_grantable yes_or_no`	Всегда `YES`

35.5. applicable_roles

Представление `applicable_roles` описывает все роли, права которых может использовать текущий пользователь. Это означает, что существует некоторая цепочка ролей от текущего пользователя к целевой роли. Роль самого пользователя также считается применимой. Набор применимых ролей обычно используется для проверки разрешений.

Таблица 35.3. Столбцы `applicable_roles`

Тип столбца	Описание
`grantee sql_identifier`	Имя роли, которой было разрешено участие в целевой роли (может быть текущий пользователь, либо другая роль, в случае вложенного членства)
`role_name sql_identifier`	Имя целевой роли
`is_grantable yes_or_no`	`YES`, если субъект является администратором для этой роли, или `NO` в противном случае

35.6. `attributes`

Представление `attributes` содержит информацию об атрибутах составных типов данных, определённых в базе. (Заметьте, что представление не даёт информацию о столбцах таблицы, которые иногда называются атрибутами в контекстах Postgres Pro.) В нём показываются только те атрибуты, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторое право для использования типа).

Таблица 35.4. Столбцы `attributes`

Тип столбца	Описание
`udt_catalog sql_identifier`	Имя базы данных, содержащей тип данных (всегда текущая база)
`udt_schema sql_identifier`	Имя схемы, содержащей тип данных
`udt_name sql_identifier`	Имя типа данных
`attribute_name sql_identifier`	Имя атрибута
`ordinal_position cardinal_number`	Порядковый номер атрибута внутри типа данных (нумерация начинается с 1)
`attribute_default character_data`	Выражение по умолчанию для атрибута
`is_nullable yes_or_no`	`YES`, если атрибут может содержать NULL, или `NO`, если он не принимает NULL
`data_type character_data`	Тип данных атрибута, если это встроенный тип, либо `ARRAY`, если это массив (в этом случае обратитесь к представлению `element_types`), иначе — `USER-DEFINED` (в этом случае тип определяется в `attribute_udt_name` и связанных столбцах).
`character_maximum_length cardinal_number`	Если в `data_type` указан тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, либо если максимальная длина не объявлена.
`character_octet_length cardinal_number`	Если в `data_type` указан тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.
`character_set_catalog sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_schema sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_name sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`collation_catalog sql_identifier`	Имя базы данных, содержащей правило сортировки атрибута (это всегда текущая база), либо NULL, если это правило по умолчанию или тип данных атрибута несортируемый
`collation_schema sql_identifier`	Имя схемы, содержащей правило сортировки атрибута, либо NULL, если это правило по умолчанию или тип данных атрибута несортируемый
`collation_name sql_identifier`	

Тип столбца	**Описание**
	Имя правила сортировки атрибута, либо NULL, если это правило по умолчанию или атрибут несортируемый
`numeric_precision` `cardinal_number`	Если в `data_type` указан числовой тип, этот столбец содержит точность (объявленную или неявную) типа для этого атрибута. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix` . Для всех других типов данных этот столбец содержит NULL.
`numeric_precision_radix` `cardinal_number`	Если в `data_type` указан числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах `numeric_precision` и `numeric_scale` . Возможные варианты: 2 или 10. Для всех других типов данных этот столбец содержит NULL.
`numeric_scale` `cardinal_number`	Если в `data_type` указан точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для этого атрибута. Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix` . Для всех других типов данных этот столбец содержит NULL.
`datetime_precision` `cardinal_number`	Если в `data_type` указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа этого атрибута, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.
`interval_type` `character_data`	Если в `data_type` указан тип интервала, этот столбец определяет, какие поля принимает интервал в этом атрибуте, например: `YEAR TO MONTH`, `DAY TO SECOND` и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
`interval_precision` `cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro (см. поле `datetime_precision` , определяющее точность в долях секунды для типов интервалов)
`attribute_udt_catalog` `sql_identifier`	Имя базы данных, в которой определён тип данных атрибута (всегда текущая база)
`attribute_udt_schema` `sql_identifier`	Имя схемы, в которой определён тип данных атрибута
`attribute_udt_name` `sql_identifier`	Имя типа данных атрибута
`scope_catalog` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`scope_schema` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`scope_name` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`maximum_cardinality` `cardinal_number`	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
`dtd_identifier` `sql_identifier`	Идентификатор дескриптора типа данных атрибута, уникальный среди всех дескрипторов типа данных, относящихся к составному типу. Он в основном полезен для соедине-

Тип столбца Описание	
	ния с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
`is_derived_reference_attribute` `yes_or_no`	Относится к функциональности, отсутствующей в Postgres Pro

Обратитесь также к описанию Раздел 35.17, представлению с похожей структурой, за дополнительной информацией о некоторых столбцах.

35.7. `character_sets`

Представление `character_sets` описывает наборы символов, доступные в текущей базе данных. Так как Postgres Pro не поддерживает несколько наборов символов в одной базе данных, этот набор показывает только один набор, соответствующий кодировке базы.

Примите к сведению следующие термины, принятые в стандарте SQL:

совокупность символов

> Абстрактная коллекция символов, например `UNICODE`, `UCS` или `LATIN1`. Не существует в виде SQL-объекта, но показывается в этом представлении.

форма кодировки символов

> Кодировка некоторой совокупности символов. Для большинства устаревших совокупностей используется только одна кодировка, так что эта кодировка не имеет отдельного имени (например, `LATIN2` — форма кодировки, применимая к совокупности `LATIN2`). Но например, Unicode имеет формы кодировки `UTF8`, `UTF16` и т. д. (не все они поддерживаются в Postgres Pro Standard). Формы кодировки не существуют в виде SQL-объектов, но показываются в этом представлении.

набор символов

> Именованный SQL-объект, определяющий совокупность и кодировку символов, а также правило сортировки по умолчанию. Предопределённый набор символов обычно называется так же, как форма кодировки, но пользователи могут определить другие имена. Например, набору символов `UTF8` обычно соответствует совокупность символов `UCS`, форма кодировки `UTF8` и некоторое правило сортировки по умолчанию.

Вы можете считать, что «кодировка» в Postgres Pro определяет набор или форму кодировки символов. Она имеет такое же имя и может быть только одной в определённой базе.

Таблица 35.5. Столбцы `character_sets`

Тип столбца Описание	
`character_set_catalog` `sql_identifier`	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL.
`character_set_schema` `sql_identifier`	Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL.
`character_set_name` `sql_identifier`	Имя набора символов, в настоящее время в качестве этого имени показывается имя кодировки базы данных
`character_repertoire` `sql_identifier`	Совокупность символов — `UCS` для кодировки `UTF8`, либо просто имя кодировки
`form_of_use` `sql_identifier`	Форма кодировки символов, то же, что и кодировка базы данных

Тип столбца **Описание**
`default_collate_catalog sql_identifier` Имя базы данных, содержащей правило сортировки по умолчанию (всегда текущая база, если это правило установлено)
`default_collate_schema sql_identifier` Имя схемы, содержащей правило сортировки по умолчанию
`default_collate_name sql_identifier` Имя правила сортировки по умолчанию. Правилом сортировки по умолчанию считается правило, соответствующее параметрам `COLLATE` и `CTYPE` текущей базы данных. Если такого правила нет, данный столбец и связанные столбцы схемы и каталога содержат NULL.

35.8. `check_constraint_routine_usage`

Представление `check_constraint_routine_usage` описывает подпрограммы (функции и процедуры), участвующие в проверках ограничений. В нём показываются только те подпрограммы, которые принадлежат текущей активной роли.

Таблица 35.6. Столбцы `check_constraint_routine_usage`

Тип столбца **Описание**
`constraint_catalog sql_identifier` Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema sql_identifier` Имя схемы, содержащей ограничение
`constraint_name sql_identifier` Имя ограничения
`specific_catalog sql_identifier` Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier` Имя схемы, содержащей функцию
`specific_name sql_identifier` «Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.

35.9. `check_constraints`

Представление `check_constraints` показывает все ограничения-проверки, либо определённые для таблицы или домена, либо принадлежащие текущей активной роли. (Владелец таблицы или домена является владельцем ограничения.)

В стандарте SQL ограничения NOT NULL рассматриваются как проверочные ограничения с выражением `CHECK (`*`имя_столбца`*` IS NOT NULL)`. Поэтому ограничения NOT NULL также включены в это представление, а не в отдельное.

Таблица 35.7. Столбцы `check_constraints`

Тип столбца **Описание**
`constraint_catalog sql_identifier` Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema sql_identifier` Имя схемы, содержащей ограничение
`constraint_name sql_identifier`

Тип столбца **Описание**
Имя ограничения
`check_clause character_data` Выражение проверки для ограничения

35.10. `collations`

Представление `collations` показывает правила сортировки, доступные в текущей базе данных.

Таблица 35.8. Столбцы `collations`

Тип столбца **Описание**
`collation_catalog sql_identifier` Имя базы данных, содержащей правило сортировки (всегда текущая база)
`collation_schema sql_identifier` Имя схемы, содержащей правило сортировки
`collation_name sql_identifier` Имя правила сортировки
`pad_attribute character_data` Всегда `NO PAD` (Альтернативный вариант `PAD SPACE` в Postgres Pro не поддерживается.)

35.11. `collation_character_set_applicability`

Представление `collation_character_set_applicability` показывает, к каким наборам символов применимы доступные правила сортировки. В Postgres Pro в базе данных может быть только один набор символов (см. описание в Разделе 35.7), так что это представление даёт не так много полезной информации.

Таблица 35.9. Столбцы `collation_character_set_applicability`

Тип столбца **Описание**
`collation_catalog sql_identifier` Имя базы данных, содержащей правило сортировки (всегда текущая база)
`collation_schema sql_identifier` Имя схемы, содержащей правило сортировки
`collation_name sql_identifier` Имя правила сортировки
`character_set_catalog sql_identifier` Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL
`character_set_schema sql_identifier` Наборы символов в настоящее время не представлены в виде объектов схемы, так что этот столбец содержит NULL
`character_set_name sql_identifier` Имя набора символов

35.12. `column_column_usage`

Представление `column_column_usage` описывает все генерируемые столбцы, которые зависят от других базовых столбцов в той же таблице. В нём показываются только таблицы, принадлежащие текущей активной роли.

Таблица 35.10. Столбцы `column_column_usage`

Тип столбца Описание
`table_catalog sql_identifier` Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema sql_identifier` Имя схемы, содержащей таблицу
`table_name sql_identifier` Имя таблицы
`column_name sql_identifier` Имя базового столбца, от которого зависит генерируемый
`dependent_column sql_identifier` Имя генерируемого столбца

35.13. `column_domain_usage`

Представление `column_domain_usage` описывает все столбцы (таблиц или представлений), которые используют какие-либо домены, определённые в базе данных и принадлежащие текущей активной роли.

Таблица 35.11. Столбцы `column_domain_usage`

Тип столбца Описание
`domain_catalog sql_identifier` Имя базы данных, содержащей домен (всегда текущая база)
`domain_schema sql_identifier` Имя схемы, содержащей домен
`domain_name sql_identifier` Имя домена
`table_catalog sql_identifier` Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema sql_identifier` Имя схемы, содержащей таблицу
`table_name sql_identifier` Имя таблицы
`column_name sql_identifier` Имя столбца

35.14. `column_options`

Представление `column_options` показывает все параметры, определённые для столбцов сторонней таблицы в текущей базе данных. В нём отражаются только те столбцы сторонних таблиц, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.12. Столбцы `column_options`

Тип столбца Описание
`table_catalog sql_identifier` Имя базы данных, содержащей стороннюю таблицу (всегда текущая база)
`table_schema sql_identifier` Имя схемы, содержащей стороннюю таблицу

Тип столбца **Описание**
`table_name` `sql_identifier` Имя сторонней таблицы
`column_name` `sql_identifier` Имя столбца
`option_name` `sql_identifier` Имя параметра
`option_value` `character_data` Значение параметра

35.15. `column_privileges`

Представление `column_privileges` описывает все права, назначенные текущей активной роли или текущей активной ролью для столбцов. Оно содержит отдельную строку для каждой комбинации столбца, праводателя и правообладателя.

Если право даётся для всей таблицы, оно будет показываться как право для каждого столбца, но только для типов прав, применимых к столбцам: `SELECT`, `INSERT`, `UPDATE`, `REFERENCES`.

Таблица 35.13. Столбцы `column_privileges`

Тип столбца **Описание**
`grantor` `sql_identifier` Имя роли, давшей право (праводатель)
`grantee` `sql_identifier` Имя роли, которой было дано право (правообладатель)
`table_catalog` `sql_identifier` Имя базы данных, содержащей таблицу с этим столбцом (всегда текущая база)
`table_schema` `sql_identifier` Имя схемы, содержащей таблицу с этим столбцом
`table_name` `sql_identifier` Имя таблицы с этим столбцом
`column_name` `sql_identifier` Имя столбца
`privilege_type` `character_data` Тип права: `SELECT`, `INSERT`, `UPDATE` или `REFERENCES`
`is_grantable` `yes_or_no` `YES`, если право может передаваться, или `NO` в противном случае

35.16. `column_udt_usage`

Представление `column_udt_usage` описывает все столбцы, которые используют типы данных, принадлежащие текущей активной роли. Заметьте, что в Postgres Pro встроенные типы данных не отличаются от определённых пользователем, так что в этом представлении выводятся и они. За подробностями обратитесь к Разделу 35.17.

Таблица 35.14. Столбцы `column_udt_usage`

Тип столбца **Описание**
`udt_catalog` `sql_identifier`

Тип столбца	Описание
	Имя базы данных, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
`udt_schema sql_identifier`	Имя схемы, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
`udt_name sql_identifier`	Имя типа данных столбца (если применимо, нижележащий тип домена)
`table_catalog sql_identifier`	Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema sql_identifier`	Имя схемы, содержащей таблицу
`table_name sql_identifier`	Имя таблицы
`column_name sql_identifier`	Имя столбца

35.17. columns

Представление `columns` содержит информацию обо всех столбцах таблиц (или столбцах представлений) в базе данных. Системные столбцы (`ctid` и т. д.) в нём не отображаются. В нём показываются только те столбцы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.15. Столбцы `columns`

Тип столбца	Описание
`table_catalog sql_identifier`	Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema sql_identifier`	Имя схемы, содержащей таблицу
`table_name sql_identifier`	Имя таблицы
`column_name sql_identifier`	Имя столбца
`ordinal_position cardinal_number`	Порядковый номер столбца в таблице (нумерация начинается с 1)
`column_default character_data`	Выражение по умолчанию для столбца
`is_nullable yes_or_no`	`YES`, если столбец может содержать NULL, или `NO`, если он не принимает NULL. Не будет принимать NULL столбец с ограничением NOT NULL, но возможны и другие варианты.
`data_type character_data`	Тип данных столбца, если это встроенный тип, либо `ARRAY`, если это массив (в этом случае обратитесь к представлению `element_types`), иначе — `USER-DEFINED` (в этом случае тип определяется в `udt_name` и связанных столбцах). Если столбец основан на домене, данный столбец показывает нижележащий тип домена (а сам домен показывается в `domain_name` и связанных столбцах).
`character_maximum_length cardinal_number`	

Тип столбца	**Описание**
	Если в `data_type` указан тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, либо если максимальная длина не объявлена.
`character_octet_length` `cardinal_number`	Если в `data_type` указан тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.
`numeric_precision` `cardinal_number`	Если в `data_type` указан числовой тип, этот столбец содержит точность (объявленную или неявную) типа для целевого столбца. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix`. Для всех других типов данных этот столбец содержит NULL.
`numeric_precision_radix` `cardinal_number`	Если в `data_type` указан числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах `numeric_precision` и `numeric_scale`. Возможные варианты: 2 или 10. Для всех других типов данных этот столбец содержит NULL.
`numeric_scale` `cardinal_number`	Если в `data_type` указан точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для целевого столбца. Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix`. Для всех других типов данных этот столбец содержит NULL.
`datetime_precision` `cardinal_number`	Если в `data_type` указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа для целевого столбца, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.
`interval_type` `character_data`	Если в `data_type` указан тип интервала, этот столбец определяет, какие поля принимает интервал в целевом столбце, например: `YEAR TO MONTH`, `DAY TO SECOND` и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
`interval_precision` `cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro (см. поле `datetime_precision`, определяющее точность в долях секунды для типов интервалов)
`character_set_catalog` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_schema` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_name` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`collation_catalog` `sql_identifier`	Имя базы данных, содержащей правило сортировки столбца (это всегда текущая база), либо NULL, если это правило по умолчанию или тип данных столбца несортируемый
`collation_schema` `sql_identifier`	Имя схемы, содержащей правило сортировки столбца, либо NULL, если это правило по умолчанию или тип данных столбца несортируемый

Тип столбца	**Описание**
`collation_name sql_identifier`	Имя правила сортировки столбца, либо NULL, если это правило по умолчанию или тип данных столбца несортируемый
`domain_catalog sql_identifier`	Если целевой столбец имеет тип домена, этот столбец содержит имя базы данных, в которой определён домен (всегда текущая база), иначе — NULL.
`domain_schema sql_identifier`	Если целевой столбец имеет тип домена, этот столбец содержит имя схемы, в которой определён домен, иначе — NULL.
`domain_name sql_identifier`	Если целевой столбец имеет тип домена, этот столбец содержит имя домена, иначе — NULL.
`udt_catalog sql_identifier`	Имя базы данных, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
`udt_schema sql_identifier`	Имя схемы, в которой определён тип (если применимо, нижележащий тип домена) столбца (всегда текущая база)
`udt_name sql_identifier`	Имя типа данных столбца (если применимо, нижележащий тип домена)
`scope_catalog sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`scope_schema sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`scope_name sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`maximum_cardinality cardinal_number`	Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
`dtd_identifier sql_identifier`	Идентификатор дескриптора типа данных столбца, уникальный среди всех дескрипторов типов, относящихся к таблице. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
`is_self_referencing yes_or_no`	Относится к функциональности, отсутствующей в Postgres Pro
`is_identity yes_or_no`	Если целевой столбец является столбцом идентификации, значение `YES`, иначе — `NO`.
`identity_generation character_data`	Если целевой столбец является столбцом идентификации, значение `ALWAYS` или `BY DEFAULT`, отражающее определение столбца.
`identity_start character_data`	Если целевой столбец является столбцом идентификации, начальное значение внутренней последовательности, иначе — NULL.
`identity_increment character_data`	Если целевой столбец является столбцом идентификации, шаг внутренней последовательности, иначе — NULL.
`identity_maximum character_data`	

Тип столбца **Описание**
Если целевой столбец является столбцом идентификации, максимальное значение внутренней последовательности, иначе — NULL.
`identity_minimum character_data` Если целевой столбец является столбцом идентификации, минимальное значение внутренней последовательности, иначе — NULL.
`identity_cycle yes_or_no` Если целевой столбец является столбцом идентификации, `YES` показывает, что внутренняя последовательность зацикливается, `NO` — не зацикливается, иначе — NULL.
`is_generated character_data` `ALWAYS`, если целевой столбец является генерируемым, иначе — `NEVER`.
`generation_expression character_data` Генерирующее выражение, если целевой столбец является генерируемым, иначе — NULL.
`is_updatable yes_or_no` `YES`, если столбец допускает изменение, или `NO` в противном случае (столбцы в базовых таблицах всегда изменяемые, но в представлениях — не обязательно)

Так как типы данных могут определяться в SQL множеством способов и Postgres Pro добавляет дополнительные варианты, представление типов в информационной схеме может быть довольно сложным. Столбец `data_type` предназначен для идентификации нижележащего встроенного типа столбца. В Postgres Pro это означает, что данный тип определён в схеме системного каталога `pg_catalog`. Этот столбец может быть полезным, если приложение способно особым образом воспринимать встроенные типы (например, форматировать числовые типы по-другому или задействовать данные в столбцах точности). Столбцы `udt_name`, `udt_schema` и `udt_catalog` всегда указывают на нижележащий тип данных столбца, даже если столбец основан на домене. (Так как в Postgres Pro встроенные типы не отличаются от определённых пользователем, в этом представлении выводятся и они. Это расширение стандарта SQL.) Эти столбцы должны учитываться, когда приложению нужно обрабатывать данные в зависимости от типа, так как в этом случае не важно, основан ли столбец на домене. Если столбец основан на домене, на него указывают столбцы `domain_name`, `domain_schema` и `domain_catalog`. Если вы хотите связать столбцы с их типами данных и обработать домены как отдельные типы, вы можете записать `coalesce(domain_name, udt_name)` и т. п.

35.18. `constraint_column_usage`

Представление `constraint_column_usage` описывает все столбцы в текущей базе данных, связанные с некоторым ограничением. В нём показываются только столбцы таблиц, принадлежащих текущей активной роли. Для ограничений-проверок это представление содержит столбцы, задействованные в выражении проверки. Для ограничений NOT NULL это представление содержит столбец, для которого определено ограничение. Для ограничений внешнего ключа оно содержит столбцы, на которые ссылается внешний ключ, а для ограничений уникального или первичного ключа — ограничиваемые столбцы.

Таблица 35.16. Столбцы `constraint_column_usage`

Тип столбца **Описание**
`table_catalog sql_identifier` Имя базы данных, содержащей таблицу со столбцом, задействованным в некотором ограничении (всегда текущая база)
`table_schema sql_identifier` Имя схемы, содержащей таблицу со столбцом, задействованным в некотором ограничении

Тип столбца **Описание**
`table_name sql_identifier` Имя таблицы со столбцом, задействованным в некотором ограничении
`column_name sql_identifier` Имя столбца, задействованного в некотором ограничении
`constraint_catalog sql_identifier` Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema sql_identifier` Имя схемы, содержащей ограничение
`constraint_name sql_identifier` Имя ограничения

35.19. `constraint_table_usage`

Представление `constraint_table_usage` описывает все таблицы в текущей базе данных, связанные с некоторым ограничением и принадлежащие текущей активной роли. (Это отличается от представления `table_constraints`, в котором показываются все ограничения таблиц с таблицами, для которых они определены.) Для ограничений внешнего ключа это представление показывает таблицу, на которую ссылается ограничение. Для ограничений уникальности или первичного ключа в этом представлении показывается таблица, которой принадлежит ограничение. Ограничения-проверки и ограничения NOT NULL в нём не отражаются.

Таблица 35.17. Столбцы `constraint_table_usage`

Тип столбца **Описание**
`table_catalog sql_identifier` Имя базы данных, которая содержит таблицу, задействованную некоторым ограничением (всегда текущая база)
`table_schema sql_identifier` Имя схемы, которая содержит таблицу, задействованную некоторым ограничением
`table_name sql_identifier` Имя таблицы, задействованной некоторым ограничением
`constraint_catalog sql_identifier` Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema sql_identifier` Имя схемы, содержащей ограничение
`constraint_name sql_identifier` Имя ограничения

35.20. `data_type_privileges`

Представление `data_type_privileges` описывает все дескрипторы типов данных, к которым имеет доступ текущий пользователь, являясь их владельцем или имея некоторые права для них. Дескриптор типа данных формируется, когда тип данных задействуется в определении столбца таблицы, домена или функции (в качестве типа параметра или результата), и хранит некоторую информацию о том, как этот тип используется в данном случае (например, объявленную максимальную длину, если это применимо). Каждому дескриптору типа данных назначается уникальный идентификатор, уникальный среди всех дескрипторов типов, назначаемых одному объекту (таблица, домен, функция). Это представление, вероятно, не полезно для приложений, но на его базе определены некоторые другие представления в информационной схеме.

Таблица 35.18. Столбцы `data_type_privileges`

Тип столбца	Описание
`object_catalog sql_identifier`	Имя базы данных, содержащей описываемый объект (всегда текущая база)
`object_schema sql_identifier`	Имя схемы, содержащей описываемый объект
`object_name sql_identifier`	Имя описываемого объекта
`object_type character_data`	Тип описываемого объекта: `TABLE` (дескриптор типа данных относится к столбцу этой таблицы), `DOMAIN` (дескриптор типа данных относится к домену) или `ROUTINE` (дескриптор типа данных относится к типу данных параметра или результата функции).
`dtd_identifier sql_identifier`	Идентификатор дескриптора типа данных, уникальный среди дескрипторов типов для этого же объекта.

35.21. domain_constraints

Представление `domain_constraints` показывает все ограничения, принадлежащие доменам, определённым в текущей базе данных. В нём отражаются только те домены, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.19. Столбцы `domain_constraints`

Тип столбца	Описание
`constraint_catalog sql_identifier`	Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema sql_identifier`	Имя схемы, содержащей ограничение
`constraint_name sql_identifier`	Имя ограничения
`domain_catalog sql_identifier`	Имя базы данных, содержащей домен (всегда текущая база)
`domain_schema sql_identifier`	Имя схемы, содержащей домен
`domain_name sql_identifier`	Имя домена
`is_deferrable yes_or_no`	`YES`, если ограничение откладываемое, или `NO` в противном случае
`initially_deferred yes_or_no`	`YES`, если ограничение откладываемое и отложенное изначально, или `NO` в противном случае

35.22. domain_udt_usage

Представление `domain_udt_usage` описывает все домены, которые используют типы данных, принадлежащие текущей активной роли. Заметьте, что в Postgres Pro встроенные типы данных не отличаются от определённых пользователем, так что в этом представлении выводятся и они.

Таблица 35.20. Столбцы `domain_udt_usage`

Тип столбца Описание
`udt_catalog` `sql_identifier` Имя базы данных, в которой определён тип данных домена (всегда текущая база)
`udt_schema` `sql_identifier` Имя схемы, в которой определён тип данных домена
`udt_name` `sql_identifier` Имя типа данных домена
`domain_catalog` `sql_identifier` Имя базы данных, содержащей домен (всегда текущая база)
`domain_schema` `sql_identifier` Имя схемы, содержащей домен
`domain_name` `sql_identifier` Имя домена

35.23. domains

Представление `domains` показывает все *домены*, определённые в текущей базе данных. В нём показываются только те домены, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.21. Столбцы `domains`

Тип столбца Описание
`domain_catalog` `sql_identifier` Имя базы данных, содержащей домен (всегда текущая база)
`domain_schema` `sql_identifier` Имя схемы, содержащей домен
`domain_name` `sql_identifier` Имя домена
`data_type` `character_data` Тип данных домена, если это встроенный тип, либо `ARRAY`, если это массив (в этом случае обратитесь к представлению `element_types`), иначе — `USER-DEFINED` (в этом случае тип определяется в `udt_name` и связанных столбцах).
`character_maximum_length` `cardinal_number` Если домен имеет тип текстовой или битовой строки, это поле задаёт её объявленную максимальную длину; NULL для всех других типов данных, или если максимальная длина не объявлена.
`character_octet_length` `cardinal_number` Если домен имеет тип символьной строки, это поле задаёт её максимально возможный размер в октетах (байтах); NULL для всех других типов данных. Максимальный размер в октетах зависит от объявленной максимальной длины в символах (см. выше) и от кодировки сервера.
`character_set_catalog` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`character_set_schema` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`character_set_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro

Тип столбца **Описание**
`collation_catalog sql_identifier` Имя базы данных, содержащей правило сортировки домена (это всегда текущая база), либо NULL, если это правило по умолчанию или тип домена несортируемый
`collation_schema sql_identifier` Имя схемы, содержащей правило сортировки домена, либо NULL, если это правило по умолчанию или тип домена несортируемый
`collation_name sql_identifier` Имя правила сортировки домена, либо NULL, если это правило по умолчанию или тип домена несортируемый
`numeric_precision cardinal_number` Если домен имеет числовой тип, этот столбец содержит точность (объявленную или неявную) типа для этого домена. Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix` . Для всех других типов данных этот столбец содержит NULL.
`numeric_precision_radix cardinal_number` Если домен имеет числовой тип, в этом столбце определяется, по какому основанию задаются значения в столбцах `numeric_precision` и `numeric_scale` . Возможные варианты: 2 и 10. Для всех других типов данных этот столбец содержит NULL.
`numeric_scale cardinal_number` Если домен имеет точный числовой тип, этот столбец содержит масштаб (объявленный или неявный) типа для этого домена. Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix` . Для всех других типов данных этот столбец содержит NULL.
`datetime_precision cardinal_number` Если в `data_type` указан тип даты, времени, отметки времени или интервала, этот столбец содержит точность (объявленную или неявную) в долях секунды типа для этого домена, то есть число десятичных цифр, сохраняемых после десятичной точки в значении секунд. Для всех других типов данных этот столбец содержит NULL.
`interval_type character_data` Если в `data_type` указан тип интервала, этот столбец определяет, какие поля принимает интервал в домене, например: `YEAR TO MONTH`, `DAY TO SECOND` и т. д. Если ограничения для полей не заданы (то есть, интервал принимает все поля), и для любых других типов данных это поле содержит NULL.
`interval_precision cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro (см. поле `datetime_precision` , определяющее точность в долях секунды для типов интервалов)
`domain_default character_data` Выражение по умолчанию для домена
`udt_catalog sql_identifier` Имя базы данных, в которой определён тип данных домена (всегда текущая база)
`udt_schema sql_identifier` Имя схемы, в которой определён тип данных домена
`udt_name sql_identifier` Имя типа данных домена
`scope_catalog sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`scope_schema sql_identifier`

Тип столбца **Описание**
Относится к функциональности, отсутствующей в Postgres Pro
`scope_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`maximum_cardinality` `cardinal_number` Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
`dtd_identifier` `sql_identifier` Идентификатор дескриптора типа данных целевого домена, уникальный среди всех дескрипторов типов, относящихся к домену (что тривиально, так как домен содержит только один дескриптор типа). Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)

35.24. `element_types`

Представление `element_types` показывает дескрипторы типов элементов массива. Когда столбец таблицы, атрибут составного типа, параметр или результат функции объявлены с типом массива, соответствующее представление информационной схемы будет содержать только `ARRAY` в столбце `data_type`. Чтобы получить информацию о типе элемента массива, можно связать соответствующее представление с данным. Например, просмотреть столбцы таблицы с типами данных и типами элементов массива (если это применимо) можно так:

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type,
 e.collection_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

Это представление показывает только те объекты, к которым имеет доступ текущий пользователь, являясь владельцем или имея некоторые права.

Таблица 35.22. Столбцы `element_types`

Тип столбца **Описание**
`object_catalog` `sql_identifier` Имя базы данных, содержащей объект, связанный с описываемым массивом (всегда текущая база)
`object_schema` `sql_identifier` Имя схемы, содержащей объект, связанный с описываемым массивом
`object_name` `sql_identifier` Имя объекта, связанного с описываемым массивом
`object_type` `character_data` Тип объекта, связанного с описываемым массивом: `TABLE` (массив задействован в столбце этой таблицы), `USER-DEFINED TYPE` (массив задействован в атрибуте составного типа), `DOMAIN` (массив задействован в домене), `ROUTINE` (массив задействован в типе данных параметра или результата функции).
`collection_type_identifier` `sql_identifier` Идентификатор дескриптора типа данных для описываемого массива. Его можно использовать для соединения со столбцами `dtd_identifier` других представлений информационной схемы.
`data_type` `character_data`

Тип столбца	**Описание**
	Тип данных элементов массива, если это встроенный тип, либо `USER-DEFINED` (в этом случае тип определяется в `udt_name` и связанных столбцах).
`character_maximum_length` `cardinal_number`	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
`character_octet_length` `cardinal_number`	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
`character_set_catalog` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_schema` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_name` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`collation_catalog` `sql_identifier`	Имя базы данных, содержащей правило сортировки типа элемента (это всегда текущая база), либо NULL, если это правило по умолчанию или тип элемента несортируемый
`collation_schema` `sql_identifier`	Имя схемы, содержащей правило сортировки типа элемента, либо NULL, если это правило по умолчанию или тип элемента несортируемый
`collation_name` `sql_identifier`	Имя правила сортировки типа элемента, либо NULL, если это правило по умолчанию или тип элемента несортируемый
`numeric_precision` `cardinal_number`	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
`numeric_precision_radix` `cardinal_number`	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
`numeric_scale` `cardinal_number`	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
`datetime_precision` `cardinal_number`	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
`interval_type` `character_data`	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
`interval_precision` `cardinal_number`	Всегда NULL, так как эта информация неприменима к типам элементов массива в Postgres Pro
`udt_catalog` `sql_identifier`	Имя базы данных, в которой определён тип данных элемента (всегда текущая база)
`udt_schema` `sql_identifier`	Имя схемы, в которой определён тип данных элемента
`udt_name` `sql_identifier`	Имя типа данных элемента
`scope_catalog` `sql_identifier`	

Тип столбца **Описание**
Относится к функциональности, отсутствующей в Postgres Pro
`scope_schema sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`scope_name sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`maximum_cardinality cardinal_number` Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
`dtd_identifier sql_identifier` Идентификатор дескриптора типа данных элемента. В настоящее время бесполезен.

35.25. `enabled_roles`

Представление `enabled_roles` описывает «доступные роли». Список доступных ролей рекурсивно определяется как роль текущего пользователя плюс роли, данные доступным ролям, с автоматическим наследованием. Другими словами, это роли, которые даны текущему пользователю непосредственно или косвенно, посредством автоматического наследования.

Для проверки разрешений применяется набор «применимых ролей», который может быть шире набора доступных ролей. Поэтому в общем случае вместо этого представления лучше использовать представление `applicable_roles` (оно описывается в Разделе 35.5).

Таблица 35.23. Столбцы `enabled_roles`

Тип столбца **Описание**
`role_name sql_identifier` Имя целевой роли

35.26. `foreign_data_wrapper_options`

Представление `foreign_data_wrapper_options` показывает все параметры, определённые для обёрток сторонних данных в текущей базе. В нём отражаются только те обёртки сторонних данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.24. Столбцы `foreign_data_wrapper_options`

Тип столбца **Описание**
`foreign_data_wrapper_catalog sql_identifier` Имя базы данных, в которой определена обёртка сторонних данных (всегда текущая база)
`foreign_data_wrapper_name sql_identifier` Имя обёртки сторонних данных
`option_name sql_identifier` Имя параметра
`option_value character_data` Значение параметра

35.27. `foreign_data_wrappers`

Представление `foreign_data_wrappers` показывает все обёртки сторонних данных, определённые в текущей базе. В нём показываются только те обёртки сторонних данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.25. Столбцы `foreign_data_wrappers`

Тип столбца **Описание**
`foreign_data_wrapper_catalog` `sql_identifier` Имя базы данных, в которой определена обёртка сторонних данных (всегда текущая база)
`foreign_data_wrapper_name` `sql_identifier` Имя обёртки сторонних данных
`authorization_identifier` `sql_identifier` Имя владельца стороннего сервера
`library_name` `character_data` Имя файла библиотеки, реализующей эту обёртку сторонних данных
`foreign_data_wrapper_language` `character_data` Язык, на котором реализована эта обёртка сторонних данных

35.28. `foreign_server_options`

Представление `foreign_server_options` показывает все параметры, определённые для сторонних серверов в текущей базе данных. В нём отражаются только те сторонние серверы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.26. Столбцы `foreign_server_options`

Тип столбца **Описание**
`foreign_server_catalog` `sql_identifier` Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
`foreign_server_name` `sql_identifier` Имя стороннего сервера
`option_name` `sql_identifier` Имя параметра
`option_value` `character_data` Значение параметра

35.29. `foreign_servers`

Представление `foreign_servers` показывает все сторонние серверы, определённые в текущей базе данных. В нём показываются только те сторонние серверы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.27. Столбцы `foreign_servers`

Тип столбца **Описание**
`foreign_server_catalog` `sql_identifier` Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
`foreign_server_name` `sql_identifier` Имя стороннего сервера
`foreign_data_wrapper_catalog` `sql_identifier` Имя базы данных, в которой определена обёртка сторонних данных, используемая сторонним сервером (всегда текущая база)
`foreign_data_wrapper_name` `sql_identifier`

Тип столбца Описание
Имя обёртки сторонних данных, используемой сторонним сервером
`foreign_server_type` `character_data` Информация о типе стороннего сервера, если она была указана при его создании
`foreign_server_version` `character_data` Информация о версии стороннего сервера, если она была указана при его создании
`authorization_identifier` `sql_identifier` Имя владельца стороннего сервера

35.30. `foreign_table_options`

Представление `foreign_table_options` показывает все параметры, определённые для сторонних таблиц в текущей базе данных. В нём отражаются только те сторонние таблицы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.28. Столбцы `foreign_table_options`

Тип столбца Описание
`foreign_table_catalog` `sql_identifier` Имя базы данных, содержащей стороннюю таблицу (всегда текущая база)
`foreign_table_schema` `sql_identifier` Имя схемы, содержащей стороннюю таблицу
`foreign_table_name` `sql_identifier` Имя сторонней таблицы
`option_name` `sql_identifier` Имя параметра
`option_value` `character_data` Значение параметра

35.31. `foreign_tables`

Представление `foreign_tables` показывает все сторонние таблицы, определённые в текущей базе данных. В нём показываются только те сторонние таблицы, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.29. Столбцы `foreign_tables`

Тип столбца Описание
`foreign_table_catalog` `sql_identifier` Имя базы данных, в которой определена сторонняя таблица (всегда текущая база)
`foreign_table_schema` `sql_identifier` Имя схемы, содержащей стороннюю таблицу
`foreign_table_name` `sql_identifier` Имя сторонней таблицы
`foreign_server_catalog` `sql_identifier` Имя базы данных, в которой определён сторонний сервер (всегда текущая база)
`foreign_server_name` `sql_identifier` Имя стороннего сервера

35.32. key_column_usage

Представление `key_column_usage` описывает все столбцы в текущей базе, с которыми связано какое-либо ограничение уникальности, либо ограничение первичного или внешнего ключа. Ограничения-проверки в этом представлении не показываются. В нём показываются только те столбцы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.30. Столбцы `key_column_usage`

Тип столбца	Описание
`constraint_catalog sql_identifier`	Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema sql_identifier`	Имя схемы, содержащей ограничение
`constraint_name sql_identifier`	Имя ограничения
`table_catalog sql_identifier`	Имя базы данных, содержащей таблицу со столбцом, подчиняющимся этому ограничению (всегда текущая база)
`table_schema sql_identifier`	Имя схемы, содержащей таблицу со столбцом, подчиняющимся этому ограничению
`table_name sql_identifier`	Имя таблицы со столбцом, подчиняющимся этому ограничению
`column_name sql_identifier`	Имя столбца, подчиняющегося этому ограничению
`ordinal_position cardinal_number`	Порядковый номер столбца в ключе ограничения (нумерация начинается с 1)
`position_in_unique_constraint cardinal_number`	Для ограничения внешнего ключа это порядковый номер целевого столбца в его ограничении уникальности (нумерация начинается с 1); в противном случае NULL

35.33. parameters

Представление `parameters` содержит информацию о параметрах (аргументах) всех функций в текущей базе данных. В нём отражаются только функции, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.31. Столбцы `parameters`

Тип столбца	Описание
`specific_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier`	Имя схемы, содержащей функцию
`specific_name sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.
`ordinal_position cardinal_number`	Порядковый номер параметра в списке аргументов функции (нумерация начинается с 1)
`parameter_mode character_data`	`IN` для входного параметра, `OUT` для выходного, `INOUT` — для входного и выходного параметра.

Тип столбца	**Описание**
`is_result` `yes_or_no`	Относится к функциональности, отсутствующей в Postgres Pro
`as_locator` `yes_or_no`	Относится к функциональности, отсутствующей в Postgres Pro
`parameter_name` `sql_identifier`	Имя параметра, либо NULL, если параметр безымянный
`data_type` `character_data`	Тип данных параметра, если это встроенный тип, либо `ARRAY`, если это массив (в этом случае обратитесь к представлению `element_types`), иначе — `USER-DEFINED` (в этом случае тип определяется в `udt_name` и связанных столбцах).
`character_maximum_length` `cardinal_number`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`character_octet_length` `cardinal_number`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`character_set_catalog` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_schema` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_name` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`collation_catalog` `sql_identifier`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`collation_schema` `sql_identifier`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`collation_name` `sql_identifier`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`numeric_precision` `cardinal_number`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`numeric_precision_radix` `cardinal_number`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`numeric_scale` `cardinal_number`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`datetime_precision` `cardinal_number`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`interval_type` `character_data`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`interval_precision` `cardinal_number`	Всегда NULL, так как эта информация не применима к типам параметров в Postgres Pro
`udt_catalog` `sql_identifier`	Имя базы данных, в которой определён тип данных параметра (всегда текущая база)
`udt_schema` `sql_identifier`	Имя схемы, в которой определён тип данных параметра
`udt_name` `sql_identifier`	Имя типа данных параметра
`scope_catalog` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro

Тип столбца Описание
`scope_schema sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`scope_name sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`maximum_cardinality cardinal_number` Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
`dtd_identifier sql_identifier` Идентификатор дескриптора типа данных параметра, уникальный среди всех дескрипторов типов, относящихся к функции. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
`parameter_default character_data` Выражение параметра по умолчанию, либо NULL, если такого выражения нет или функция не принадлежит текущей активной роли.

35.34. `referential_constraints`

Представление `referential_constraints` содержит все ссылочные ограничения (внешнего ключа) в текущей базе данных. В нём показываются только ограничения, в которых ссылающаяся таблица доступна текущему пользователю на запись (он является её владельцем или имеет не только право `SELECT`).

Таблица 35.32. Столбцы `referential_constraints`

Тип столбца Описание
`constraint_catalog sql_identifier` Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema sql_identifier` Имя схемы, содержащей ограничение
`constraint_name sql_identifier` Имя ограничения
`unique_constraint_catalog sql_identifier` Имя базы данных, содержащей ограничение уникальности или первичный ключ, на которые ссылается ограничение внешнего ключа (всегда текущая база)
`unique_constraint_schema sql_identifier` Имя схемы, содержащей ограничение уникальности или первичный ключ, на которые ссылается ограничение внешнего ключа
`unique_constraint_name sql_identifier` Имя ограничения уникальности или первичного ключа, на которые ссылается ограничение внешнего ключа
`match_option character_data` Тип совпадения для ограничения внешнего ключа: `FULL`, `PARTIAL` или `NONE`.
`update_rule character_data` Правило изменения для ограничения внешнего ключа: `CASCADE`, `SET NULL`, `SET DEFAULT`, `RESTRICT` или `NO ACTION`.
`delete_rule character_data` Правило удаления для ограничения внешнего ключа: `CASCADE`, `SET NULL`, `SET DEFAULT`, `RESTRICT` или `NO ACTION`.

35.35. role_column_grants

Представление `role_column_grants` описывает все назначенные для столбцов права, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в `column_privileges`. Единственное существенное отличие этого представления от `column_privileges` состоит в том, что в данном представлении опускаются столбцы, которые доступны текущему пользователю косвенно через роль `PUBLIC`.

Таблица 35.33. Столбцы `role_column_grants`

Тип столбца	Описание
`grantor sql_identifier`	Имя роли, давшей право (праводатель)
`grantee sql_identifier`	Имя роли, которой было дано право (правообладатель)
`table_catalog sql_identifier`	Имя базы данных, содержащей таблицу с этим столбцом (всегда текущая база)
`table_schema sql_identifier`	Имя схемы, содержащей таблицу с этим столбцом
`table_name sql_identifier`	Имя таблицы с этим столбцом
`column_name sql_identifier`	Имя столбца
`privilege_type character_data`	Тип права: `SELECT`, `INSERT`, `UPDATE` или `REFERENCES`
`is_grantable yes_or_no`	`YES`, если право может передаваться, или `NO` в противном случае

35.36. role_routine_grants

Представление `role_routine_grants` описывает все назначенные для функций права, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в `routine_privileges`. Единственное существенное отличие этого представления от `routine_privileges` состоит в том, что в данном представлении опускаются функции, которые доступны текущему пользователю косвенно через роль `PUBLIC`.

Таблица 35.34. Столбцы `role_routine_grants`

Тип столбца	Описание
`grantor sql_identifier`	Имя роли, давшей право (праводатель)
`grantee sql_identifier`	Имя роли, которой было дано право (правообладатель)
`specific_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier`	Имя схемы, содержащей функцию
`specific_name sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.
`routine_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)

Тип столбца Описание
`routine_schema sql_identifier` Имя схемы, содержащей функцию
`routine_name sql_identifier` Имя функции (может дублироваться в случае перегрузки)
`privilege_type character_data` Всегда `EXECUTE` (единственный тип прав для функций)
`is_grantable yes_or_no` `YES`, если право может передаваться, или `NO` в противном случае

35.37. `role_table_grants`

Представление `role_table_grants` описывает все назначенные для таблиц и представлений права, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в `table_privileges`. Единственное существенное отличие этого представления от `table_privileges` состоит в том, что в данном представлении опускаются таблицы, которые доступны текущему пользователю косвенно через роль `PUBLIC`.

Таблица 35.35. Столбцы `role_table_grants`

Тип столбца Описание
`grantor sql_identifier` Имя роли, давшей право (праводатель)
`grantee sql_identifier` Имя роли, которой было дано право (правообладатель)
`table_catalog sql_identifier` Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema sql_identifier` Имя схемы, содержащей таблицу
`table_name sql_identifier` Имя таблицы
`privilege_type character_data` Тип права: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES` или `TRIGGER`
`is_grantable yes_or_no` `YES`, если право может передаваться, или `NO` в противном случае
`with_hierarchy yes_or_no` В стандарте SQL имеется отдельное подчинённое разрешение `WITH HIERARCHY OPTION`, позволяющее выполнять определённые операции в иерархии наследования таблиц. В Postgres Pro так действует право `SELECT`, так что в этом столбце выводится `YES` для права `SELECT`, а для других — `NO`.

35.38. `role_udt_grants`

Представление `role_udt_grants` предназначено для отображения прав `USAGE`, назначенных для пользовательских типов, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в `udt_privileges`. Единственное существенное отличие этого представления от `udt_privileges` состоит в том, что в данном представлении опускаются объекты, которые доступны текущему пользователю косвенно через роль `PUBLIC`. Так как с типами данных не связываются действующие права в Postgres Pro (только `PUBLIC` неявно даётся право их использования), это представление пустое.

Таблица 35.36. Столбцы `role_udt_grants`

Тип столбца Описание
`grantor sql_identifier` Имя роли, которая дала это право
`grantee sql_identifier` Имя роли, которой было дано это право
`udt_catalog sql_identifier` Имя базы данных, содержащей тип (всегда текущая база)
`udt_schema sql_identifier` Имя схемы, содержащей тип
`udt_name sql_identifier` Имя типа
`privilege_type character_data` Всегда `TYPE USAGE`
`is_grantable yes_or_no` `YES`, если право может передаваться, или `NO` в противном случае

35.39. `role_usage_grants`

Представление `role_usage_grants` описывает права `USAGE`, назначенные для объектов различных типов, в которых праводателем или правообладателем является текущая активная роль. Дополнительную информацию можно найти в `usage_privileges`. Единственное существенное отличие этого представления от `usage_privileges` состоит в том, что в данном представлении опускаются объекты, которые доступны текущему пользователю косвенно через роль `PUBLIC`.

Таблица 35.37. Столбцы `role_usage_grants`

Тип столбца Описание
`grantor sql_identifier` Имя роли, которая дала это право
`grantee sql_identifier` Имя роли, которой было дано это право
`object_catalog sql_identifier` Имя базы данных, содержащей объект (всегда текущая база)
`object_schema sql_identifier` Имя схемы, содержащей объект, если это применимо, иначе пустая строка
`object_name sql_identifier` Имя объекта
`object_type character_data` `COLLATION` или `DOMAIN` или `FOREIGN DATA WRAPPER` или `FOREIGN SERVER` или `SEQUENCE`
`privilege_type character_data` Всегда `USAGE`
`is_grantable yes_or_no` `YES`, если право может передаваться, или `NO` в противном случае

35.40. `routine_column_usage`

Представление `routine_column_usage` идентифицирует все столбцы, используемые функцией или процедурой, либо в теле SQL, либо в стандартных выражениях параметров. (Это представление поддерживается только для тел SQL без кавычек, но не для тел или функций на других языках.) Столбец выводится только в том случае, если его таблица принадлежит текущей активной роли.

Таблица 35.38. Столбцы `routine_column_usage`

Тип столбца	Описание
`specific_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier`	Имя схемы, содержащей функцию
`specific_name sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.
`routine_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`routine_schema sql_identifier`	Имя схемы, содержащей функцию
`routine_name sql_identifier`	Имя функции (может дублироваться в случае перегрузки)
`table_catalog sql_identifier`	Имя базы данных, содержащей таблицу, задействованную в функции (всегда текущая база)
`table_schema sql_identifier`	Имя схемы, содержащей таблицу, задействованную в функции
`table_name sql_identifier`	Имя таблицы, задействованной в функции
`column_name sql_identifier`	Имя столбца, задействованного в функции

35.41. `routine_privileges`

Представление `routine_privileges` описывает все права, назначенные текущей активной роли или текущей активной ролью для функций. Оно содержит отдельный столбец для каждой комбинации функции, праводателя и правообладателя.

Таблица 35.39. Столбцы `routine_privileges`

Тип столбца	Описание
`grantor sql_identifier`	Имя роли, давшей право (праводатель)
`grantee sql_identifier`	Имя роли, которой было дано право (правообладатель)
`specific_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier`	Имя схемы, содержащей функцию
`specific_name sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.
`routine_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`routine_schema sql_identifier`	

Тип столбца **Описание**
Имя схемы, содержащей функцию
`routine_name` `sql_identifier` Имя функции (может дублироваться в случае перегрузки)
`privilege_type` `character_data` Всегда `EXECUTE` (единственный тип прав для функций)
`is_grantable` `yes_or_no` `YES`, если право может передаваться, или `NO` в противном случае

35.42. `routine_routine_usage`

Представление `routine_routine_usage` идентифицирует все функции или процедуры, которые используются другой (или той же) функцией или процедурой, либо в теле SQL, либо в стандартных выражениях параметров. (Это представление поддерживается только для тел SQL без кавычек, но не для тел или функций на других языках.) Запись включается в представление только в том случае, если используемая функция принадлежит текущей активной роли. (Это ограничение не распространяется на использующую функцию.)

Обратите внимание, что в записях этого представления для обеих подпрограмм задаётся однозначное имя подпрограммы, несмотря на то, что имена столбцов используются не так, как в других представлениях информационной схемы, посвящённых подпрограммам. Этого требует стандарт SQL, хотя по логике так не должно быть. За дополнительными сведениями об однозначных именах обратитесь к Разделу 35.45.

Таблица 35.40. Столбцы `routine_routine_usage`

Тип столбца **Описание**
`specific_catalog` `sql_identifier` Имя базы данных, содержащей задействовавшую её функцию (всегда текущая база)
`specific_schema` `sql_identifier` Имя схемы, содержащей задействовавшую её функцию
`specific_name` `sql_identifier` «Однозначное имя» задействовавшей функции.
`routine_catalog` `sql_identifier` Имя базы данных, содержащей функцию, задействованную в первой функции (всегда текущая база)
`routine_schema` `sql_identifier` Имя схемы, содержащей функцию, задействованную в первой функции
`routine_name` `sql_identifier` «Однозначное имя» функции, задействованной в первой функции.

35.43. `routine_sequence_usage`

Представление `routine_sequence_usage` идентифицирует все последовательности, используемые функцией или процедурой, либо в теле SQL, либо в стандартных выражениях параметров. (Это представление поддерживается только для тел SQL без кавычек, но не для тел или функций на других языках.) Последовательность выводится только в случае, если она принадлежит текущей активной роли.

Таблица 35.41. Столбцы `routine_sequence_usage`

Тип столбца	**Описание**
`specific_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier`	Имя схемы, содержащей функцию
`specific_name sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.
`routine_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`routine_schema sql_identifier`	Имя схемы, содержащей функцию
`routine_name sql_identifier`	Имя функции (может дублироваться в случае перегрузки)
`schema_catalog sql_identifier`	Имя базы данных, содержащей последовательность, задействованную в функции (всегда текущая база)
`sequence_schema sql_identifier`	Имя схемы, содержащей последовательность, задействованную в функции
`sequence_name sql_identifier`	Имя последовательности, задействованной в функции

35.44. `routine_table_usage`

Представление `routine_table_usage` предназначено для идентификации всех таблиц, задействованных в функции или процедуре. В настоящее время Postgres Pro не собирает эту информацию.

Таблица 35.42. Столбцы `routine_table_usage`

Тип столбца	**Описание**
`specific_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier`	Имя схемы, содержащей функцию
`specific_name sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.
`routine_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`routine_schema sql_identifier`	Имя схемы, содержащей функцию
`routine_name sql_identifier`	Имя функции (может дублироваться в случае перегрузки)
`table_catalog sql_identifier`	Имя базы данных, содержащей таблицу, задействованную в функции (всегда текущая база)
`table_schema sql_identifier`	Имя схемы, содержащей таблицу, задействованную в функции
`table_name sql_identifier`	Имя таблицы, задействованной в функции

35.45. `routines`

Представление `routines` отображает все функции и процедуры в текущей базе данных. В нём показываются только те функции и процедуры, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.43. Столбцы `routines`

Тип столбца	Описание
`specific_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier`	Имя схемы, содержащей функцию
`specific_name sql_identifier`	«Однозначное имя» функции. Это имя однозначным образом идентифицирует функцию в схеме, даже если реальное имя функции перегружено. Формат однозначных имён не определён, так что его следует использовать только для сравнения с другими экземплярами однозначных имён подпрограмм.
`routine_catalog sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`routine_schema sql_identifier`	Имя схемы, содержащей функцию
`routine_name sql_identifier`	Имя функции (может дублироваться в случае перегрузки)
`routine_type character_data`	`FUNCTION` для функций, `PROCEDURE` для процедур
`module_catalog sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`module_schema sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`module_name sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`udt_catalog sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`udt_schema sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`udt_name sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`data_type character_data`	Тип данных результата функции, если это встроенный тип, либо `ARRAY`, если это массив (в этом случае обратитесь к представлению `element_types`), иначе — `USER-DEFINED` (в этом случае тип определяется в `type_udt_name` и связанных столбцах). Для процедуры данное поле содержит NULL.
`character_maximum_length cardinal_number`	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`character_octet_length cardinal_number`	Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`character_set_catalog sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_schema sql_identifier`	

Тип столбца **Описание**
Относится к функциональности, отсутствующей в Postgres Pro
`character_set_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`collation_catalog` `sql_identifier` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`collation_schema` `sql_identifier` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`collation_name` `sql_identifier` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`numeric_precision` `cardinal_number` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`numeric_precision_radix` `cardinal_number` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`numeric_scale` `cardinal_number` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`datetime_precision` `cardinal_number` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`interval_type` `character_data` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`interval_precision` `cardinal_number` Всегда NULL, так как эта информация неприменима к типам результатов в Postgres Pro
`type_udt_catalog` `sql_identifier` Имя базы данных, в которой определён тип данных результата функции (всегда текущая база). Для процедуры данное поле содержит NULL.
`type_udt_schema` `sql_identifier` Имя схемы, в которой определён тип данных результата функции. Для процедуры данное поле содержит NULL.
`type_udt_name` `sql_identifier` Имя типа данных результата функции. Для процедуры данное поле содержит NULL.
`scope_catalog` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`scope_schema` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`scope_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`maximum_cardinality` `cardinal_number` Всегда NULL, так как массивы имеют неограниченную максимальную ёмкость в Postgres Pro
`dtd_identifier` `sql_identifier` Идентификатор дескриптора типа данных результата функции, уникальный среди всех дескрипторов типов, относящихся к функции. Он в основном полезен для соединения с другими экземплярами таких идентификаторов. (Конкретный формат идентификатора не определён и не гарантируется, что он останется неизменным в будущих версиях.)
`routine_body` `character_data` Если функция написана на SQL, это поле содержит `SQL`, иначе `EXTERNAL`.
`routine_definition` `character_data`

Тип столбца	**Описание**
	Исходный текст функции (NULL, если функция не принадлежит текущей активной роли). (Согласно стандарту SQL, этот столбец актуален, только если в `routine_body` указано `SQL`, но в Postgres Pro он будет содержать любой исходный текст, заданный при создании функции.)
`external_name` `character_data`	Если это функция на C, этот столбец содержит внешнее имя (объектный символ) функции, иначе — NULL. (Это будет то же значение, что содержит столбец `routine_definition`.)
`external_language` `character_data`	Язык, на котором написана функция
`parameter_style` `character_data`	Всегда `GENERAL` (В стандарте SQL определены и другие стили параметров, но в Postgres Pro они отсутствуют.)
`is_deterministic` `yes_or_no`	Если функция объявлена как постоянная (IMMUTABLE) (в стандарте SQL она называется детерминированной), этот столбец содержит `YES`, иначе — `NO`. (Получить другие уровни переменности функций, имеющиеся в Postgres Pro, через информационную схему нельзя.)
`sql_data_access` `character_data`	Всегда `MODIFIES`, что означает, что функция может модифицировать данные SQL. Для Postgres Pro эта информация бесполезна.
`is_null_call` `yes_or_no`	Если функция автоматически возвращает NULL, когда один из аргументов NULL, этот столбец содержит `YES`, иначе — `NO`. Для процедуры он содержит NULL.
`sql_path` `character_data`	Относится к функциональности, отсутствующей в Postgres Pro
`schema_level_routine` `yes_or_no`	Всегда `YES` (Другое значение было бы у методов пользовательских типов, но в Postgres Pro их нет.)
`max_dynamic_result_sets` `cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro
`is_user_defined_cast` `yes_or_no`	Относится к функциональности, отсутствующей в Postgres Pro
`is_implicitly_invocable` `yes_or_no`	Относится к функциональности, отсутствующей в Postgres Pro
`security_type` `character_data`	Если функция выполняется с правами вызывающего пользователя, этот столбец содержит `INVOKER`, а если с правами пользователя, создавшего её, то — `DEFINER`.
`to_sql_specific_catalog` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`to_sql_specific_schema` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`to_sql_specific_name` `sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`as_locator` `yes_or_no`	Относится к функциональности, отсутствующей в Postgres Pro
`created` `time_stamp`	Относится к функциональности, отсутствующей в Postgres Pro

Тип столбца **Описание**
`last_altered` `time_stamp` Относится к функциональности, отсутствующей в Postgres Pro
`new_savepoint_level` `yes_or_no` Относится к функциональности, отсутствующей в Postgres Pro
`is_udt_dependent` `yes_or_no` В настоящее время всегда `NO`. Альтернативное значение `YES` связано с возможностями, отсутствующими в Postgres Pro.
`result_cast_from_data_type` `character_data` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_as_locator` `yes_or_no` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_char_max_length` `cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_char_octet_length` `cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_char_set_catalog` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_char_set_schema` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_char_set_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_collation_catalog` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_collation_schema` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_collation_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_numeric_precision` `cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_numeric_precision_radix` `cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_numeric_scale` `cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_datetime_precision` `cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_interval_type` `character_data` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_interval_precision` `cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_type_udt_catalog` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_type_udt_schema` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_type_udt_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_scope_catalog` `sql_identifier`

Тип столбца **Описание**
Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_scope_schema` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_scope_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_maximum_cardinality` `cardinal_number` Относится к функциональности, отсутствующей в Postgres Pro
`result_cast_dtd_identifier` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro

35.46. `schemata`

Представление `schemata` показывает все схемы в текущей базе данных, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.44. Столбцы `schemata`

Тип столбца **Описание**
`catalog_name` `sql_identifier` Имя базы данных, содержащей схему (всегда текущая база)
`schema_name` `sql_identifier` Имя схемы
`schema_owner` `sql_identifier` Имя владельца схемы
`default_character_set_catalog` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`default_character_set_schema` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`default_character_set_name` `sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`sql_path` `character_data` Относится к функциональности, отсутствующей в Postgres Pro

35.47. `sequences`

Представление `sequences` показывает все последовательности, определённые в текущей базе данных. В нём показываются только те последовательности, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.45. Столбцы `sequences`

Тип столбца **Описание**
`sequence_catalog` `sql_identifier` Имя базы данных, содержащей последовательность (всегда текущая база)
`sequence_schema` `sql_identifier` Имя схемы, содержащей последовательность
`sequence_name` `sql_identifier` Имя последовательности
`data_type` `character_data`

Тип столбца	**Описание**
	Тип данных последовательности.
`numeric_precision cardinal_number`	Этот столбец содержит точность (объявленную или неявную) типа данных последовательности (см. выше). Точность определяет число значащих цифр. Она может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix` .
`numeric_precision_radix cardinal_number`	Этот столбец определяет, по какому основанию задаются значения в столбцах `numeric_precision` и `numeric_scale` . Возможные варианты: 2 и 10.
`numeric_scale cardinal_number`	Этот столбец содержит масштаб (объявленный или неявный) типа данных последовательности (см. выше). Масштаб определяет число значащих цифр справа от десятичной точки. Он может выражаться в десятичных (по основанию 10) или двоичных (по основанию 2) цифрах, согласно столбцу `numeric_precision_radix` .
`start_value character_data`	Начальное значение последовательности
`minimum_value character_data`	Минимальное значение последовательности
`maximum_value character_data`	Максимальное значение последовательности
`increment character_data`	Шаг увеличения последовательности
`cycle_option yes_or_no`	`YES`, если последовательность зацикливается, или `NO` в противном случае

Заметьте, что, согласно стандарту SQL, начальное, минимальное, максимальное значение и шаг выдаются в виде символьных строк.

35.48. `sql_features`

Таблица `sql_features` содержит информацию о формальной функциональности, описанной в стандарте и поддерживаемой Postgres Pro. Эта же информация представлена в Приложении D. Там вы можете найти и дополнительные сведения.

Таблица 35.46. Столбцы `sql_features`

Тип столбца	**Описание**
`feature_id character_data`	Строка идентификатора функциональности
`feature_name character_data`	Описательное название функциональности
`sub_feature_id character_data`	Строка идентификатора подчинённой возможности, либо строка нулевой длины, если это не подчинённая возможность
`sub_feature_name character_data`	Описательное название подчинённой возможности, либо строка нулевой длины, если это не подчинённая возможность
`is_supported yes_or_no`	`YES`, если функциональность полностью поддерживается текущей версией Postgres Pro, либо `NO` в противном случае

Тип столбца **Описание**
`is_verified_by` `character_data` Всегда NULL, так как группа разработки Postgres Pro не проводит формальное тестирование функционального соответствия
`comments` `character_data` Необязательный комментарий о поддерживаемом состоянии функциональности

35.49. `sql_implementation_info`

Таблица `sql_implementation_info` содержит информацию о различных аспектах, которые в стандарте SQL оставлены на усмотрение реализации. Эта информация в основном предназначена для применения в контексте интерфейса ODBC; пользователям других интерфейсов она вряд ли будет нужна. Поэтому отдельные элементы особенностей реализации здесь не описываются; о них можно прочитать в описании интерфейса ODBC.

Таблица 35.47. Столбцы `sql_implementation_info`

Тип столбца **Описание**
`implementation_info_id` `character_data` Строка идентификатора элемента особенности реализации
`implementation_info_name` `character_data` Описательное название элемента особенности реализации
`integer_value` `cardinal_number` Значение элемента особенности реализации, либо NULL, если его значение содержится в столбце `character_value`
`character_value` `character_data` Значение элемента особенности реализации, либо NULL, если его значение содержится в столбце `integer_value`
`comments` `character_data` Необязательный комментарий, относящийся к элементу особенности реализации

35.50. `sql_parts`

Таблица `sql_parts` содержит информацию о различных частях стандарта SQL, поддерживаемых Postgres Pro.

Таблица 35.48. Столбцы `sql_parts`

Тип столбца **Описание**
`feature_id` `character_data` Строка идентификатора, содержащая номер части
`feature_name` `character_data` Описательное название части
`is_supported` `yes_or_no` `YES`, если часть полностью поддерживается текущей версией Postgres Pro, либо `NO` в противном случае
`is_verified_by` `character_data` Всегда NULL, так как группа разработки Postgres Pro не проводит формальное тестирование функционального соответствия
`comments` `character_data` Необязательный комментарий о поддерживаемом состоянии части

35.51. `sql_sizing`

Таблица `sql_sizing` содержит информацию о различных ограничениях размера и максимальных значениях в Postgres Pro. Эта информация в основном предназначена для применения в контексте интерфейса ODBC; пользователям других интерфейсов она вряд ли будет нужна. Поэтому отдельные элементы размеров здесь не описываются; о них можно прочитать в описании интерфейса ODBC.

Таблица 35.49. Столбцы `sql_sizing`

Тип столбца Описание
`sizing_id cardinal_number` Идентификатор элемента размеров
`sizing_name character_data` Описательное название элемента размеров
`supported_value cardinal_number` Значение элемента размеров, или 0, если размер неограниченный или не может быть определён, либо NULL, если функциональность, к которой относится размер, не поддерживается
`comments character_data` Необязательный комментарий, относящийся к элементу размеров

35.52. `table_constraints`

Представление `table_constraints` показывает все ограничения, принадлежащие таблицам, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права, кроме `SELECT`).

Таблица 35.50. Столбцы `table_constraints`

Тип столбца Описание
`constraint_catalog sql_identifier` Имя базы данных, содержащей ограничение (всегда текущая база)
`constraint_schema sql_identifier` Имя схемы, содержащей ограничение
`constraint_name sql_identifier` Имя ограничения
`table_catalog sql_identifier` Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema sql_identifier` Имя схемы, содержащей таблицу
`table_name sql_identifier` Имя таблицы
`constraint_type character_data` Тип ограничения: `CHECK` (включает ограничения NOT NULL), `FOREIGN KEY`, `PRIMARY KEY` или `UNIQUE`
`is_deferrable yes_or_no` `YES`, если ограничение откладываемое, или `NO` в противном случае
`initially_deferred yes_or_no` `YES`, если ограничение откладываемое и отложенное изначально, или `NO` в противном случае

Тип столбца	**Описание**
`enforced` `yes_or_no`	Относится к функциональности, отсутствующей в Postgres Pro (в настоящее время всегда равно `YES`)
`nulls_distinct` `yes_or_no`	Для ограничений уникальности — `YES`, если значения NULL считаются различными для данного ограничения, или `NO`, если они считаются одинаковыми; для других типов ограничений — NULL.

35.53. `table_privileges`

Представление `table_privileges` описывает все права, назначенные текущей активной роли или текущей активной ролью для таблиц и представлений. Оно содержит отдельную строку для каждой комбинации таблицы, праводателя и правообладателя.

Таблица 35.51. Столбцы `table_privileges`

Тип столбца	**Описание**
`grantor` `sql_identifier`	Имя роли, давшей право (праводатель)
`grantee` `sql_identifier`	Имя роли, которой было дано право (правообладатель)
`table_catalog` `sql_identifier`	Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema` `sql_identifier`	Имя схемы, содержащей таблицу
`table_name` `sql_identifier`	Имя таблицы
`privilege_type` `character_data`	Тип права: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES` или `TRIGGER`
`is_grantable` `yes_or_no`	`YES`, если право может передаваться, или `NO` в противном случае
`with_hierarchy` `yes_or_no`	В стандарте SQL имеется отдельное подчинённое разрешение `WITH HIERARCHY OPTION`, позволяющее выполнять определённые операции в иерархии наследования таблиц. В Postgres Pro так действует право `SELECT`, так что в этом столбце выводится `YES` для права `SELECT`, а для других — `NO`.

35.54. `tables`

Представление `tables` показывает все таблицы и представления, определённые в текущей базе данных. В нём показываются только те таблицы и представления, к которым имеет доступ текущий пользователь (являясь их владельцем или имея некоторые права).

Таблица 35.52. Столбцы `tables`

Тип столбца	**Описание**
`table_catalog` `sql_identifier`	Имя базы данных, содержащей таблицу (всегда текущая база)
`table_schema` `sql_identifier`	

Тип столбца **Описание**
Имя схемы, содержащей таблицу
`table_name sql_identifier` Имя таблицы
`table_type character_data` Тип таблицы: `BASE TABLE` для постоянных базовых таблиц (таблиц обычного типа), `VIEW` для представлений, `FOREIGN` для сторонних таблиц, либо `LOCAL TEMPORARY` для временных таблиц
`self_referencing_column_name sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`reference_generation character_data` Относится к функциональности, отсутствующей в Postgres Pro
`user_defined_type_catalog sql_identifier` Если таблица является типизированной, это имя базы данных, содержащей нижележащий тип данных (всегда текущая база), иначе — NULL.
`user_defined_type_schema sql_identifier` Если таблица является типизированной, это имя схемы, содержащей нижележащий тип данных, иначе — NULL.
`user_defined_type_name sql_identifier` Если таблица является типизированной, это имя типа данных, иначе — NULL.
`is_insertable_into yes_or_no` `YES`, если в эту таблицу можно добавлять данные, или `NO` в противном случае (Базовые таблицы всегда допускают добавление данных, но представления — не обязательно.)
`is_typed yes_or_no` `YES`, если эта таблица является типизированной, иначе — `NO`
`commit_action character_data` Ещё не реализовано

35.55. `transforms`

Представление `transforms` содержит информацию о трансформациях, определённых в текущей базе данных. Более точно, оно содержит строку для каждой функции, задействованной в трансформации (функцию «из SQL» или «в SQL»).

Таблица 35.53. Столбцы `transforms`

Тип столбца **Описание**
`udt_catalog sql_identifier` Имя базы данных, содержащей тип, для которого предназначена трансформация (всегда текущая база)
`udt_schema sql_identifier` Имя схемы, содержащей тип, для которого предназначена трансформация
`udt_name sql_identifier` Имя типа, для которого предназначена трансформация
`specific_catalog sql_identifier` Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema sql_identifier` Имя схемы, содержащей функцию
`specific_name sql_identifier`

Тип столбца	Описание
	«Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.
`group_name sql_identifier`	Стандарт SQL позволяет определять «группы» трансформаций и выбирать группу во время выполнения. Postgres Pro это не поддерживает. Вместо этого трансформации привязываются к языкам. В качестве компромисса со стандартом, это поле содержит язык, для которого предназначена трансформация.
`transform_type character_data`	`FROM SQL` или `TO SQL`

35.56. `triggered_update_columns`

Для триггеров в текущей базе данных, установленных для списка столбцов (например, `UPDATE OF column1, column2`), представление `triggered_update_columns` показывает эти столбцы. Триггеры, для которых не задаётся список столбцов, в этом представлении не отражаются. В нём показываются только столбцы, доступные текущему пользователю (как владельцу или имеющему некоторые права, кроме `SELECT`).

Таблица 35.54. Столбцы `triggered_update_columns`

Тип столбца	Описание
`trigger_catalog sql_identifier`	Имя базы данных, содержащей триггер (всегда текущая база)
`trigger_schema sql_identifier`	Имя схемы, содержащей триггер
`trigger_name sql_identifier`	Имя триггера
`event_object_catalog sql_identifier`	Имя базы данных, содержащей таблицу, для которой определён триггер (всегда текущая база)
`event_object_schema sql_identifier`	Имя схемы, содержащей таблицу, для которой определён триггер
`event_object_table sql_identifier`	Имя таблицы, для которой определён триггер
`event_object_column sql_identifier`	Имя столбца, для которого определён триггер

35.57. `triggers`

Представление `triggers` показывает все триггеры, определённые в текущей базе данных для таблиц и представлений, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права, кроме `SELECT`).

Таблица 35.55. Столбцы `triggers`

Тип столбца	Описание
`trigger_catalog sql_identifier`	Имя базы данных, содержащей триггер (всегда текущая база)
`trigger_schema sql_identifier`	Имя схемы, содержащей триггер

Тип столбца **Описание**
`trigger_name sql_identifier` Имя триггера
`event_manipulation character_data` Событие, вызывающие срабатывание триггера (`INSERT`, `UPDATE` или `DELETE`)
`event_object_catalog sql_identifier` Имя базы данных, содержащей таблицу, для которой определён триггер (всегда текущая база)
`event_object_schema sql_identifier` Имя схемы, содержащей таблицу, для которой определён триггер
`event_object_table sql_identifier` Имя таблицы, для которой определён триггер
`action_order cardinal_number` Порядок срабатывания триггеров, имеющих одинаковые свойства `event_manipulation` , `action_timing` и `action_orientation` . В Postgres Pro триггеры срабатывают по порядку их имён, что и отражается в этом столбце.
`action_condition character_data` Условие `WHEN` триггера, либо NULL, если его нет (так же NULL, если таблица не принадлежит текущей активной роли)
`action_statement character_data` Оператор, выполняемый триггером (в настоящее время всегда `EXECUTE FUNCTION` *функция*`(...)`)
`action_orientation character_data` Определяет, срабатывает ли триггер для каждой обрабатываемой строки или только для каждого оператора (`ROW` или `STATEMENT`)
`action_timing character_data` Момент срабатывания триггера (`BEFORE` (до), `AFTER` (после) или `INSTEAD OF` (вместо))
`action_reference_old_table sql_identifier` Имя «старой» переходной таблицы либо NULL, если её нет
`action_reference_new_table sql_identifier` Имя «новой» переходной таблицы либо NULL, если её нет
`action_reference_old_row sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`action_reference_new_row sql_identifier` Относится к функциональности, отсутствующей в Postgres Pro
`created time_stamp` Относится к функциональности, отсутствующей в Postgres Pro

Триггеры в Postgres Pro несовместимы со стандартом в двух аспектах, которые влияют на их представление в информационной схеме. Во-первых, имена триггеров являются локальными для каждой таблицы в Postgres Pro, а не независимыми объектами схемы. Таким образом, в одной схеме могут быть дублирующиеся имена триггеров, если они принадлежат разным таблицам. (Значения `trigger_catalog` и `trigger_schema` на самом деле относятся к таблице, для которой определён триггер.) Во-вторых, триггеры в Postgres Pro могут срабатывать при нескольких событиях (например, `ON INSERT OR UPDATE`), тогда как стандарт SQL допускает только одно событие. Если триггер настроен на несколько событий, он представляется в информационной схеме в виде нескольких строк, по одной для каждого типа события. Вследствие этих двух особенностей, первичный ключ в представлении `triggers` на самом деле `(trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation)`, а не `(trigger_catalog, trigger_schema, trigger_name)`, как должно быть согласно стандарту SQL. Однако если определять триггеры в строгом соответствии со стандартом SQL (чтобы имена триггеров были уникальны в схеме и каждый триггер связывался только с одним событием), это расхождение никак не проявится.

> **Примечание**
>
> До PostgreSQL 9.1 в этом представлении столбцы `action_timing`, `action_reference_old_table`, `action_reference_new_table`, `action_reference_old_row` и `action_reference_new_row` назывались `condition_timing`, `condition_reference_old_table`, `condition_reference_new_table`, `condition_reference_old_row` и `condition_reference_new_row`, соответственно. Старые имена были продиктованы стандартом SQL:1999. Новые имена соответствуют стандарту SQL:2003 и более поздним.

35.58. `udt_privileges`

Представление `udt_privileges` описывает права `USAGE`, назначенные текущей активной роли или текущей активной ролью для пользовательских типов. Оно содержит отдельную строку для каждой комбинации типа, праводателя и правообладателя. Это представление показывает только составные типы (почему, говорится в Разделе 35.60); права, затрагивающие домены, описаны в Разделе 35.59.

Таблица 35.56. Столбцы `udt_privileges`

Тип столбца **Описание**
`grantor` `sql_identifier` Имя роли, давшей право (праводатель)
`grantee` `sql_identifier` Имя роли, которой было дано право (правообладатель)
`udt_catalog` `sql_identifier` Имя базы данных, содержащей тип (всегда текущая база)
`udt_schema` `sql_identifier` Имя схемы, содержащей тип
`udt_name` `sql_identifier` Имя типа
`privilege_type` `character_data` Всегда `TYPE USAGE`
`is_grantable` `yes_or_no` `YES`, если право может передаваться, или `NO` в противном случае

35.59. `usage_privileges`

Представление `usage_privileges` описывает права `USAGE`, назначенные текущей активной роли или текущей активной ролью для различных типов объектов. В Postgres Pro на данный момент это правила сортировки, домены, обёртки сторонних данных, сторонние серверы и последовательности. Оно содержит отдельную строку для каждой комбинации объекта, праводателя и правообладателя.

Так как с правилами сортировки в Postgres Pro не связаны никакие действующие права, в этом представлении показываются фиктивные права `USAGE`, якобы назначенные владельцем базы роли `PUBLIC` для всех правил сортировки. Для других типов объектов, однако, в нём показываются фактические права.

В Postgres Pro для последовательностей могут назначаться права `SELECT` и `UPDATE`, в дополнение к `USAGE`. Но это нестандартные права и поэтому в информационной схеме они не видны.

Таблица 35.57. Столбцы `usage_privileges`

Тип столбца Описание
`grantor sql_identifier` Имя роли, давшей право (праводатель)
`grantee sql_identifier` Имя роли, которой было дано право (правообладатель)
`object_catalog sql_identifier` Имя базы данных, содержащей объект (всегда текущая база)
`object_schema sql_identifier` Имя схемы, содержащей объект, если это применимо, иначе пустая строка
`object_name sql_identifier` Имя объекта
`object_type character_data` `COLLATION` или `DOMAIN` или `FOREIGN DATA WRAPPER` или `FOREIGN SERVER` или `SEQUENCE`
`privilege_type character_data` Всегда `USAGE`
`is_grantable yes_or_no` `YES`, если право может передаваться, или `NO` в противном случае

35.60. `user_defined_types`

Представление `user_defined_types` в данное время показывает все составные типы, определённые в текущей базе данных. В нём показываются только те типы, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

SQL знает два вида пользовательских типов: структурные типы (также называются составными типами в Postgres Pro) и отдельные типы (не реализованы в Postgres Pro). Для уверенности в будущем, нужно обратиться к столбцу `user_defined_type_category`, чтобы различить их. Другие пользовательские типы, как например, базовые типы и перечисления, относящиеся к расширениям Postgres Pro, в этом представлении не показываются. О доменах можно узнать в Разделе 35.23.

Таблица 35.58. Столбцы `user_defined_types`

Тип столбца Описание
`user_defined_type_catalog sql_identifier` Имя базы данных, содержащей тип (всегда текущая база)
`user_defined_type_schema sql_identifier` Имя схемы, содержащей тип
`user_defined_type_name sql_identifier` Имя типа
`user_defined_type_category character_data` На данный момент всегда `STRUCTURED`
`is_instantiable yes_or_no` Относится к функциональности, отсутствующей в Postgres Pro
`is_final yes_or_no` Относится к функциональности, отсутствующей в Postgres Pro
`ordering_form character_data` Относится к функциональности, отсутствующей в Postgres Pro
`ordering_category character_data`

Тип столбца	Описание
	Относится к функциональности, отсутствующей в Postgres Pro
`ordering_routine_catalog sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`ordering_routine_schema sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`ordering_routine_name sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`reference_type character_data`	Относится к функциональности, отсутствующей в Postgres Pro
`data_type character_data`	Относится к функциональности, отсутствующей в Postgres Pro
`character_maximum_length cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro
`character_octet_length cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_catalog sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_schema sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`character_set_name sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`collation_catalog sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`collation_schema sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`collation_name sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`numeric_precision cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro
`numeric_precision_radix cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro
`numeric_scale cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro
`datetime_precision cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro
`interval_type character_data`	Относится к функциональности, отсутствующей в Postgres Pro
`interval_precision cardinal_number`	Относится к функциональности, отсутствующей в Postgres Pro
`source_dtd_identifier sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro
`ref_dtd_identifier sql_identifier`	Относится к функциональности, отсутствующей в Postgres Pro

35.61. user_mapping_options

Представление `user_mapping_options` показывает все параметры, заданные для сопоставлений пользователей в текущей базе данных. В нём отражаются только сопоставления пользователей, установленные для сторонних серверов, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.59. Столбцы `user_mapping_options`

Тип столбца **Описание**
`authorization_identifier sql_identifier` Имя сопоставляемого пользователя, либо `PUBLIC`, если это сопоставление для всех
`foreign_server_catalog sql_identifier` Имя базы данных, в которой определён сторонний сервер, задействованный в сопоставлении (всегда текущая база)
`foreign_server_name sql_identifier` Имя стороннего сервера, задействованного в сопоставлении
`option_name sql_identifier` Имя параметра
`option_value character_data` Значение параметра. Этот столбец будет содержать не NULL, только если описывается сопоставление текущего пользователя, либо это сопоставление для `PUBLIC`, а текущий пользователь — владелец стороннего сервера или суперпользователь. Данное ограничение введено для защиты информации о пароле, сохранённой в параметрах сопоставления.

35.62. user_mappings

Представление `user_mappings` показывает все сопоставления пользователей, определённые в текущей базе данных. В нём показываются только сопоставления, установленные для сторонних серверов, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.60. Столбцы `user_mappings`

Тип столбца **Описание**
`authorization_identifier sql_identifier` Имя сопоставляемого пользователя, либо `PUBLIC`, если это сопоставление для всех
`foreign_server_catalog sql_identifier` Имя базы данных, в которой определён сторонний сервер, задействованный в сопоставлении (всегда текущая база)
`foreign_server_name sql_identifier` Имя стороннего сервера, задействованного в сопоставлении

35.63. view_column_usage

Представление `view_column_usage` описывает все столбцы, задействованные в выражении запроса представления (операторе `SELECT`, определяющем представление). Столбец выводится в этом списке, только если содержащая его таблица принадлежит текущей активной роли.

Примечание

Столбцы системных таблиц в этом представлении не отображаются. Это может быть исправлено позже.

Таблица 35.61. Столбцы `view_column_usage`

Тип столбца	Описание
`view_catalog` `sql_identifier`	Имя базы данных, содержащей представление (всегда текущая база)
`view_schema` `sql_identifier`	Имя схемы, содержащей представление
`view_name` `sql_identifier`	Имя представления
`table_catalog` `sql_identifier`	Имя базы данных, содержащей таблицу со столбцом, задействованным в представлении (всегда текущая база)
`table_schema` `sql_identifier`	Имя схемы, содержащей таблицу со столбцом, задействованным в представлении
`table_name` `sql_identifier`	Имя таблицы со столбцом, задействованным в представлении
`column_name` `sql_identifier`	Имя столбца, задействованного в представлении

35.64. `view_routine_usage`

Представление `view_routine_usage` описывает все подпрограммы (функции и процедуры), используемые в выражении запроса представления (операторе `SELECT`, определяющем представление). Подпрограмма выводится в этом списке, только если она принадлежит текущей активной роли.

Таблица 35.62. Столбцы `view_routine_usage`

Тип столбца	Описание
`table_catalog` `sql_identifier`	Имя базы данных, содержащей представление (всегда текущая база)
`table_schema` `sql_identifier`	Имя схемы, содержащей представление
`table_name` `sql_identifier`	Имя представления
`specific_catalog` `sql_identifier`	Имя базы данных, содержащей функцию (всегда текущая база)
`specific_schema` `sql_identifier`	Имя схемы, содержащей функцию
`specific_name` `sql_identifier`	«Однозначное имя» функции. Подробнее оно описано в Разделе 35.45.

35.65. `view_table_usage`

Представление `view_table_usage` описывает все таблицы, задействованные в выражении запроса представления (операторе `SELECT`, определяющем представление). В этом представлении показываются только таблицы, принадлежащие текущей активной роли.

Примечание

Системные таблицы в этом представлении не отображаются. Это может быть исправлено позже.

Таблица 35.63. Столбцы `view_table_usage`

Тип столбца Описание
`view_catalog sql_identifier` Имя базы данных, содержащей представление (всегда текущая база)
`view_schema sql_identifier` Имя схемы, содержащей представление
`view_name sql_identifier` Имя представления
`table_catalog sql_identifier` Имя базы данных, содержащей таблицу, задействованную в представлении (всегда текущая база)
`table_schema sql_identifier` Имя схемы, содержащей таблицу, задействованную в представлении
`table_name sql_identifier` Имя таблицы, задействованной в представлении

35.66. `views`

Представление `views` показывает все представления, определённые в текущей базе данных. В нём показываются только представления, к которым имеет доступ текущий пользователь (являясь владельцем или имея некоторые права).

Таблица 35.64. Столбцы `views`

Тип столбца Описание
`table_catalog sql_identifier` Имя базы данных, содержащей представление (всегда текущая база)
`table_schema sql_identifier` Имя схемы, содержащей представление
`table_name sql_identifier` Имя представления
`view_definition character_data` Выражение запроса, определяющее представление (NULL, если представление не принадлежит текущей активной роли)
`check_option character_data` `CASCADED` или `LOCAL`, если для представления определена характеристика `CHECK OPTION`, и `NONE` в противном случае
`is_updatable yes_or_no` `YES`, если представление допускает изменение (команды `UPDATE` и `DELETE`), или `NO` в противном случае
`is_insertable_into yes_or_no` `YES`, если представление допускает добавление данных (команду `INSERT`), или `NO` в противном случае

Тип столбца **Описание**
`is_trigger_updatable` `yes_or_no` `YES`, если для представления определён триггер `INSTEAD OF UPDATE`, или `NO` в противном случае
`is_trigger_deletable` `yes_or_no` `YES`, если для представления определён триггер `INSTEAD OF DELETE`, или `NO` в противном случае
`is_trigger_insertable_into` `yes_or_no` `YES`, если для представления определён триггер `INSTEAD OF INSERT`, или `NO` в противном случае

Часть V. Серверное программирование

Эта часть документации посвящена расширению функциональности сервера путём реализации собственных функций, типов данных, триггеров и т. д. Это довольно сложные темы, для освоения которых рекомендуется предварительно изучить и понять всю остальную документацию для пользователей Postgres Pro. В последних главах этой части описываются языки программирования на стороне сервера, поддерживаемые дистрибутивом Postgres Pro, и рассматриваются общие вопросы, связанные с программированием на стороне сервера. Но прежде чем погружаться в этот материал, важно изучить предыдущие разделы Главы 36 (где освещаются функции).

Глава 36. Расширение SQL

В следующих разделах мы обсудим, как в Postgres Pro можно расширять язык запросов SQL, добавляя собственные:

- функции (начиная с Раздела 36.3)
- агрегатные функции (начиная с Раздела 36.12)
- типы данных (начиная с Раздела 36.13)
- операторы (начиная с Раздела 36.14)
- классы операторов для индексов (начиная с Раздела 36.16)
- пакеты связанных объектов (начиная с Раздела 36.17)

36.1. Как реализована расширяемость

Postgres Pro является расширяемым благодаря тому, что его работа управляется каталогами. Если вы знакомы с традиционными реляционными системами баз данных, вы знаете, что они хранят информацию о базах, таблицах, столбцах и т. д., в структурах, которые обычно называются системными каталогами. (В некоторых системах они называются словарями данных.) Эти каталоги представляются пользователю в виде таблиц, подобных любым другим, но СУБД ведёт в них свои внутренние записи. Ключевое отличие Postgres Pro от обычных реляционных СУБД состоит в том, что Postgres Pro хранит в этих каталогах намного больше информации: информацию не только о таблицах и столбцах, но также о типах данных, функциях, методах доступа и т. д. Эти таблицы могут быть изменены пользователями, а так как Postgres Pro в своих действиях руководствуется этими таблицами, это означает, что пользователи могут расширять Postgres Pro. Обычные же СУБД можно расширять, только модифицируя жёстко запрограммированные процедуры в исходном коде или загружая модули, специально разработанные производителем СУБД.

Кроме того, сервер Postgres Pro может динамически загружать в свой процесс код, написанный пользователем. То есть, пользователь может подключить файл с объектным кодом (например, разделяемую библиотеку), который реализует новый тип или функцию, а Postgres Pro загрузит его по мере надобности. Код, написанный на SQL, добавляется на сервер ещё проще. Эта способность менять своё поведение «на лету» делает Postgres Pro исключительно подходящим для быстрого прототипирования новых приложений и структур хранения.

36.2. Система типов Postgres Pro

Типы данных Postgres Pro делятся на базовые, типы-контейнеры, составные, доменные и псевдотипы.

36.2.1. Базовые типы

Базовые типы — это типы, такие как `integer`, которые реализуются ниже уровня языка SQL (обычно на низкоуровневом языке, например C). В общих чертах они соответствуют так называемым абстрактным типам данных. Postgres Pro может работать с такими типами только через функции, предоставленные пользователем, и понимать их поведение только в той степени, в какой его опишет пользователь. Встроенные базовые типы описываются в Главе 8.

Типы-перечисления (enum) можно считать подкатегорией базовых типов. Они отличаются от других типов тем, что их можно создавать просто командами SQL, обходясь без низкоуровневого программирования. За подробностями обратитесь к Разделу 8.7.

36.2.2. Типы-контейнеры

В Postgres Pro есть три вида «типов-контейнеров», то есть типов, которые могут содержать в себе несколько значений других типов. Это массивы, составные типы и диапазоны.

Массивы могут содержать множество значений, имеющих один тип. Тип массива автоматически создаётся для каждого базового и составного типа, диапазона и домена, но не для массивов — массивы массивов не существуют. Для системы типов многомерные массивы не отличаются от одномерных. За дополнительными сведениями обратитесь к Разделу 8.15.

Составные типы, или типы строк, образуются при создании любой таблицы. С помощью команды CREATE TYPE также можно определить «независимый» составной тип, не связанный с таблицей. Составной тип представляет собой просто список типов с определёнными именами полей. Значением составного типа является строка таблицы или запись из значений полей. За дополнительными сведениями обратитесь к Разделу 8.16.

Диапазонный тип может содержать два значения одного типа, которые определяют нижнюю и верхнюю границу диапазона. Диапазонные типы создаются пользователем, хотя существует и несколько встроенных. За дополнительными сведениями обратитесь к Разделу 8.17.

36.2.3. Домены

Домен основывается на определённом нижележащем типе и во многих аспектах взаимозаменяем с ним. Однако домен может иметь ограничения, уменьшающие множество допустимых для него значений относительно нижележащего типа. Домены создаются SQL-командой CREATE DOMAIN. За дополнительными сведениями обратитесь к Разделу 8.18.

36.2.4. Псевдотипы

Для специальных целей существует также несколько «псевдотипов». Псевдотипы нельзя задействовать в столбцах таблицы или в типах-контейнерах, но их можно использовать в объявлениях аргументов и результатов функций. Это даёт возможность выделить в системе типов специальные классы функций. Все существующие псевдотипы перечислены в Таблице 8.27.

36.2.5. Полиморфные типы

Особый интерес представляет подмножество псевдотипов, *полиморфные типы*, которые применяются в объявлениях *полиморфных функций*. Используя такие типы, можно объявить всего одну функцию, которая будет работать с разными типами данных, определяя конкретные типы в зависимости от того, значения каких типов были переданы ей при вызове. Полиморфные типы перечислены в Таблице 36.1. Некоторые примеры их использования показаны в Подразделе 36.5.11.

Таблица 36.1. Полиморфные типы

Имя	Семейство	Описание
`anyelement`	Простое	Указывает, что функция принимает любой тип
`anyarray`	Простое	Указывает, что функция принимает любой тип массива
`anynonarray`	Простое	Указывает, что функция принимает любой тип, отличный от массива
`anyenum`	Простое	Указывает, что функция принимает любой тип-перечисление (см. Раздел 8.7)
`anyrange`	Простое	Указывает, что функция принимает любой диапазонный тип (см. Раздел 8.17)
`anymultirange`	Простое	Указывает, что функция принимает любой мультидиапазонный тип (см. Раздел 8.17)
`anycompatible`	Общее	Указывает, что функция принимает любой тип, с автоматическим приведением нескольких аргументов к общему типу
`anycompatiblearray`	Общее	Указывает, что функция принимает любой тип массива, с автоматиче-

Имя	Семейство	Описание
		ским приведением нескольких аргументов к общему типу
`anycompatiblenonarray`	Общее	Указывает, что функция принимает любой тип, отличный от массива, с автоматическим приведением нескольких аргументов к общему типу
`anycompatiblerange`	Общее	Указывает, что функция принимает любой диапазонный тип, с автоматическим приведением нескольких аргументов к общему типу
`anycompatiblemultirange`	Общее	Указывает, что функция принимает любой мультидиапазонный тип данных и может автоматически приводить различные аргументы к общему типу данных

Полиморфные аргументы и результаты связаны друг с другом и сводятся к конкретным типам данным при разборе запроса, вызывающего полиморфную функцию. Когда полиморфных аргументов несколько, фактические типы данных входных значений должны совмещаться, как описано далее. Если тип результата функции полиморфный или у неё имеются выходные параметры полиморфных типов, фактические типы этих результатов выводятся из типов полиморфных входных значений, как описано ниже.

Для «простого» семейства полиморфных типов действуют следующие правила совмещения и выведения типов:

В каждой позиции (в аргументах или возвращаемом значении), объявленной как `anyelement`, может передаваться любой фактический тип данных, но в каждом конкретном вызове все эти фактические типы должны быть *одинаковыми*. Аналогичным образом, в каждой позиции, объявленной как `anyarray`, может передаваться любой тип данных массива, но все фактические типы должны совпадать. Так же и во всех позициях, объявленных как `anyrange`, должен передаваться одинаковый диапазонный тип, а в объявленных как `anymultirange` — одинаковый мультидиапазонный тип.

Более того, если некоторые позиции объявлены как `anyarray`, а другие как `anyelement`, то фактическим типом в позициях `anyarray` должен быть массив, элементы которого имеют тот же тип, что и значения в позициях `anyelement`. Псевдотип `anynonarray` обрабатывается так же, как `anyelement`, но с дополнительным ограничением — фактический тип не должен быть типом массива. Псевдотип `anyenum` тоже обрабатывается как `anyelement`, но его фактические типы ограничиваются перечислениями.

Подобным образом, если одни позиции объявлены как `anyrange`, а другие как `anyelement` или `anyarray`, фактическим типом в позициях `anyrange` должен быть диапазон, подтип которого совпадает с типом элементов в позициях `anyelement` и с типом, передаваемым в позициях `anyarray`. Если есть позиции, объявленные `anymultirange`, их фактический мультидиапазонный тип должен содержать диапазоны, соответствующие объявленным параметрам `anyrange`, и базовые элементы, соответствующие параметрам, объявленным `anyelement` и `anyarray`.

Таким образом, когда с полиморфным типом объявлено несколько аргументов, в итоге допускаются только определённые комбинации фактических типов. Например, функция, объявленная как `equal(anyelement, anyelement)`, примет в аргументах любые два значения, но только если их типы данных совпадают.

Когда с полиморфным типом объявлено возвращаемое значение функции, так же полиморфным должен быть минимум один аргумент, и фактический тип результата при конкретном вызове определится по типу фактически переданного полиморфного аргумента (или аргументов). Например, если бы отсутствовал механизм обращения к элементам массива, его можно было бы реализовать, создав функцию `subscript(anyarray, integer) returns anyelement`. С таким объявлением первым фактическим аргументом должен быть массив, и из него будет выведен правильный тип результата при разборе запроса. В качестве другого примера можно привести функцию `f(anyarray) returns anyenum`, которая будет принимать только массивы перечислений.

В большинстве случаев при разборе функции фактический тип данных для полиморфного результата может быть выведен из аргументов, имеющих другой полиморфный тип из того же семейства; например, подтип `anyarray` может выводиться из `anyelement` и наоборот. Исключение представляет полиморфный результат типа `anyrange` — для него требуется аргумент типа `anyrange`; вывести его фактический тип из типа аргументов `anyarray` или `anyelement` нельзя. Это объясняется тем, что на одном подтипе могут базироваться несколько диапазонных типов.

Заметьте, что `anynonarray` и `anyenum` представляют не отдельные типы переменных; это те же типы, что и `anyelement`, но с дополнительными ограничениями. Например, объявление функции `f(anyelement, anyenum)` равнозначно объявлению `f(anyenum, anyenum)`: оба фактических аргумента должны быть одинаковыми типами-перечислениями.

Для «общего» семейства полиморфных типов работают примерно те же правила совмещения и выведения типов, что и для «простого» семейства, но есть одно важно отличие: фактические типы аргументов не должны совпадать, если они могут быть неявно приведены к некоторому общему типу. Этот общий тип выбирается по тем же правилам, что применяются в `UNION` и подобных конструкциях (см. Раздел 10.5). При выборе общего типа учитываются фактические типы аргументов `anycompatible` и `anycompatiblenonarray`, типы элементов в аргументах `anycompatiblearray`, подтипы диапазонов в аргументах `anycompatiblerange` и подтипы мультидиапазонов в аргументах `anycompatiblemultirange`. Если присутствует тип `anycompatiblenonarray`, общим типом не должен быть тип массива. После того как общий тип определён, аргументы `anycompatible` и `anycompatiblenonarray` автоматически приводятся к этому типу, а аргументы `anycompatiblearray` приводятся к типу-массиву с элементами этого типа.

Так как невозможно выбрать диапазонный тип, зная только его подтип, при использовании `anycompatiblerange` и/или `anycompatiblemultirange` необходимо, чтобы все аргументы, объявленные с этим типом, имели один диапазонный и/или мультидиапазонный тип, а его подтип соответствовал выбранному общему типу, что позволяет обойтись без приведения типов для диапазонных значений. Как и в случае с типами `anyrange` и `anymultirange`, типы `anycompatiblerange` и `anycompatiblemultirange` можно использовать в качестве типа результата функции, только если у неё имеется аргумент того же типа (`anycompatiblerange` или `anycompatiblemultirange`).

Заметьте, что типа `anycompatibleenum` не существует. Такой тип был бы не очень полезным, так как никаких неявных приведений к типам-перечислениям обычно нет, то есть не существует способа найти общий тип для двух различных перечислений.

Полиморфные семейства «простое» и «общее» представляют два независимых набора переменных типов. Рассмотрите, например, объявление:

```
CREATE FUNCTION myfunc(a anyelement, b anyelement,
                       c anycompatible, d anycompatible)
RETURNS anycompatible AS ...
```

Когда эта функция вызывается, первые два аргумента обязательно должны иметь один и тот же тип. Последние два аргумента должны быть приводимыми к общему типу, причём этот тип может не совпадать с типом первых двух аргументов. Этот общий тип станет типом результата.

Функции с переменным числом аргументом (описанные в Подразделе 36.5.6) тоже могут быть полиморфными: для этого их последний параметр описывается как `VARIADIC anyarray` или `VARIADIC anycompatiblearray`. Для целей сопоставления аргументов и определения фактического типа результата такая функция представляется так же, как если бы в ней явно объявлялось нужное число параметров `anynonarray` или `anycompatiblenonarray`.

36.3. Пользовательские функции

В Postgres Pro представлены функции четырёх видов:

- функции на языке запросов (функции, написанные на SQL) (Раздел 36.5)
- функции на процедурных языках (функции, написанные, например, на PL/pgSQL или PL/Tcl) (Раздел 36.8)
- внутренние функции (Раздел 36.9)
- функции на языке C (Раздел 36.10)

Функции любых видов могут принимать в качестве аргументов (параметров) базовые типы, составные типы или их сочетания. Кроме того, любые функции могут возвращать значения базового или составного типа. Также можно определить функции, возвращающие наборы базовых или составных значений.

Функции многих видов могут также принимать или возвращать определённые псевдотипы (например, полиморфные типы), но доступные средства для работы с ними различаются. За подробностями обратитесь к описанию конкретного вида функций.

Проще всего определить функции на языке SQL, поэтому сначала мы рассмотрим их. Многие концепции, касающиеся функций на SQL, затем распространятся и на другие виды функций.

Изучая эту главу, будет полезно обращаться к странице справки по команде `CREATE FUNCTION`, чтобы лучше понимать примеры.

36.4. Пользовательские процедуры

Процедура — объект базы данных, подобный функции, но имеющий следующие отличия:

- Процедуры определяются командой `CREATE PROCEDURE`, а не `CREATE FUNCTION`.
- Процедуры, в отличие от функций, не возвращают значение; поэтому в `CREATE PROCEDURE` отсутствует предложение `RETURNS`. Однако процедуры могут выдавать данные в вызывающий код через выходные параметры.
- Функции вызываются как часть запроса или команды DML, а процедуры вызываются отдельно командой `CALL`.
- Процедура, в отличие от функции, может фиксировать или откатывать транзакции во время её выполнения (а затем автоматически начинать новую транзакцию), если вызывающая команда `CALL` находится не в явном блоке транзакции.
- Некоторые атрибуты функций (например, `STRICT`) неприменимы к процедурам. Эти атрибуты влияют на вызов функций в запросах и не имеют отношения к процедурам.

В следующих разделах рассказывается о том, как определять пользовательские функции, что также применимо к процедурам, с учётом вышеперечисленных особенностей.

Функции и процедуры в совокупности также называются *подпрограммами*. Существуют команды, такие как `ALTER ROUTINE` и `DROP ROUTINE`, которые способны работать и с функциями, и с процедурами, не требуя указания точного вида объекта. Однако заметьте, что команды `CREATE ROUTINE` нет.

36.5. Функции на языке запросов (SQL)

SQL-функции выполняют произвольный список операторов SQL и возвращают результат последнего запроса в списке. В простом случае (не с множеством) будет возвращена первая строка результата последнего запроса. (Помните, что понятие «первая строка» в наборе результатов с несколькими строками определено точно, только если присутствует `ORDER BY`.) Если последний запрос вообще не вернёт строки, будет возвращено значение NULL.

Кроме того, можно объявить SQL-функцию как возвращающую множество (то есть, несколько строк), указав в качестве возвращаемого типа функции `SETOF` *`некий_тип`*, либо объявив её с указанием `RETURNS TABLE(`*`столбцы`*`)`. В этом случае будут возвращены все строки результата последнего запроса. Подробнее это описывается ниже.

Тело SQL-функции должно представлять собой список SQL-операторов, разделённых точкой с запятой. Точка с запятой после последнего оператора может отсутствовать. Если только функция не объявлена как возвращающая `void`, последним оператором должен быть `SELECT`, либо `INSERT`, `UPDATE`, `DELETE` или `MERGE` с предложением `RETURNING`.

Любой набор команд на языке SQL можно скомпоновать вместе и обозначить как функцию. Помимо запросов `SELECT`, эти команды могут включать запросы, изменяющие данные (`INSERT`, `UPDATE` и `DELETE` и `MERGE`), а также другие SQL-команды. (В SQL-функциях нельзя использовать команды управления транзакциями, например `COMMIT`, `SAVEPOINT`, и некоторые вспомогательные команды, в частности `VACUUM`.) Однако последней командой должна быть `SELECT` или команда с предложением `RETURNING`, возвращающая результат с типом возврата функции. Если же вы хотите определить функцию SQL, выполняющую действия, но не возвращающую полезное значение, вы можете объявить её как возвращающую тип `void`. Например, эта функция удаляет строки с отрицательным жалованьем из таблицы `emp`:

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

Если записать то же самое в виде процедуры, то можно не указывать возвращаемый тип. Например:

```
CREATE PROCEDURE clean_emp() AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

CALL clean_emp();
```

В таких простых случаях разница между процедурой и функцией, возвращающей `void`, в основном стилистическая. Однако по сравнению с функциями процедуры предоставляют дополнительную функциональность, например возможности управления транзакциями. Кроме того, процедуры соответствуют стандарту SQL, а возвращающие `void` функции специфичны для Postgres Pro.

> **Примечание**
>
> Прежде чем начинается выполнение команд, разбирается всё тело SQL-функции. Когда SQL-функция содержит команды, модифицирующие системные каталоги (например, `CREATE TABLE`), действие таких команд не будет проявляться на стадии анализа последующих команд этой функции. Так, например, команды `CREATE TABLE foo (...); INSERT INTO foo VALUES(...);` не будут работать, как ожидается, если их упаковать в одну SQL-функцию, так как `foo` не будет существовать к моменту разбору команды `INSERT`. В подобных ситуациях вместо SQL-функции рекомендуется использовать PL/pgSQL.

Синтаксис команды `CREATE FUNCTION` требует, чтобы тело функции было записано как строковая константа. Обычно для этого удобнее всего заключать строковую константу в доллары (см. Подраздел 4.1.2.4). Если вы решите использовать обычный синтаксис с заключением строки в апострофы, вам придётся дублировать апострофы (`'`) и обратную косую черту (`\`) (предполагается синтаксис спецпоследовательностей) в теле функции (см. Подраздел 4.1.2.1).

36.5.1. Аргументы SQL-функций

К аргументам SQL-функции можно обращаться в теле функции по именам или номерам. Ниже приведены примеры обоих вариантов.

Чтобы использовать имя, объявите аргумент функции как именованный, а затем просто пишите это имя в теле функции. Если имя аргумента совпадает с именем какого-либо столбца в текущей SQL-команде внутри функции, имя столбца будет иметь приоритет. Чтобы всё же перекрыть имя столбца, дополните имя аргумента именем самой функции, то есть запишите его в виде *`имя_функции`*`.`*`имя_аргумента`*. (Если и это имя будет конфликтовать с полным именем столбца, снова выиграет имя столбца. Неоднозначности в этом случае вы можете избежать, выбрав другой псевдоним для таблицы в SQL-команде.)

Старый подход с нумерацией позволяет обращаться к аргументам, применяя запись `$`*`n`*: `$1` обозначает первый аргумент, `$2` — второй и т. д. Это будет работать и в том случае, если данному аргументу назначено имя.

Если аргумент имеет составной тип, то для обращения к его атрибутам можно использовать запись с точкой, например: *`аргумент`*`.`*`поле`* или `$1.`*`поле`*. И опять же, при этом может потребоваться дополнить имя аргумента именем функции, чтобы сделать имя аргумента однозначным.

Аргументы SQL-функции могут использоваться только как значения данных, но не как идентификаторы. Например, это приемлемо:

```
INSERT INTO mytable VALUES ($1);
```

а это не будет работать:

```
INSERT INTO $1 VALUES (42);
```

> **Примечание**
>
> Возможность обращаться к аргументам SQL-функций по именам появилась в PostgreSQL 9.2. В функциях, которые должны работать со старыми серверами, необходимо применять запись `$`*`n`*.

36.5.2. Функции SQL с базовыми типами

Простейшая возможная функция SQL не имеет аргументов и просто возвращает базовый тип, например `integer`:

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

-- Альтернативная запись строковой константы:
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

Заметьте, что мы определили псевдоним столбца в теле функции для её результата (дали ему имя `result`), но этот псевдоним не виден снаружи функции. Вследствие этого, столбец результата получил имя `one`, а не `result`.

Практически так же легко определяются функции SQL, которые принимают в аргументах базовые типы:

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

Мы также можем отказаться от имён аргументов и обращаться к ним по номерам:

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

Вот более полезная функция, которую можно использовать, чтобы дебетовать банковский счёт:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT 1;
$$ LANGUAGE SQL;
```

Пользователь может выполнить эту функцию, чтобы дебетовать счёт 17 на 100 долларов, так:

```
SELECT tf1(17, 100.0);
```

В этом примере мы выбрали имя `accountno` для первого аргумента, но это же имя имеет столбец в таблице `bank`. В команде `UPDATE` имя `accountno` относится к столбцу `bank.accountno`, так для обращения к аргументу нужно записать `tf1.accountno`. Конечно, мы могли бы избежать этого, выбрав другое имя для аргумента.

На практике обычно желательно получать от функции более полезный результат, чем константу 1, поэтому более реалистично такое определение:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

Эта функция изменяет баланс и возвращает полученное значение. То же самое можно сделать в одной команде, применив `RETURNING`:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

Если последняя команда `SELECT` или `RETURNING` в SQL-функции возвращает не в точности объявленный тип результата, Postgres Pro автоматически приведёт возвращаемое значение к нужному типу, если это возможно с применением приведения присваивания или неявным образом. В противном случае вы должны применить явное приведение. Например, предположим, что мы захотели изменить возвращаемый тип в предыдущей функции `add_em` на `float8`. Это можно сделать так:

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
```

Сумма типа `integer` может быть неявно приведена к типу `float8`, поэтому достаточно сменить только тип результата. (Подробнее о приведениях говорится в Главе 10 и описании CREATE CAST.)

36.5.3. Функции SQL с составными типами

В функциях с аргументами составных типов мы должны указывать не только, какой аргумент, но и какой атрибут (поле) этого аргумента нам нужен. Например, предположим, что `emp` — таблица, содержащая данные работников, и это же имя составного типа, представляющего каждую строку таблицы. Следующая функция `double_salary` вычисляет, каким было бы чьё-либо жалование в случае увеличения вдвое:

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Bill |  8400
```

Обратите внимание на запись `$1.salary` позволяющую выбрать одно поле из значения строки аргумента. Также заметьте, что в вызывающей команде `SELECT` указание *`имя_таблицы`*`.*` выбирает всю текущую строку таблицы как составное значение. На строку таблицы можно сослаться и просто по имени таблицы, например так:

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

Однако это использование считается устаревшим, так как провоцирует путаницу. (Подробнее эти две записи составных значений строки таблицы описаны в Подразделе 8.16.5.)

Иногда бывает удобно образовать составное значение аргумента на лету. Это позволяет сделать конструкция `ROW`. Например, так можно изменить данные, передаваемые функции:

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

Также возможно создать функцию, возвращающую составной тип. Например, эта функция возвращает одну строку `emp`:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
```

```
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

В этом примере мы задали для каждого атрибута постоянное значение, но вместо этих констант можно подставить любые вычисления.

Учтите два важных требования относительно определения функции:

- Порядок в списке выборки внутреннего запроса должен в точности совпадать с порядком следования столбцов в составном типе. (Имена столбцов, как показывает пример выше, для системы значения не имеют.)
- Необходимо сделать так, чтобы каждое выражение могло приводиться к типу соответствующего столбца составного типа. В противном случае вы получите такие ошибки:

  ```
  ERROR:  return type mismatch in function declared to return emp
  DETAIL:  Final statement returns text instead of point at column 4.
  ```

  (ОШИБКА: несовпадение типа возврата в функции (в объявлении указан тип emp); ПОДРОБНОСТИ: Последний оператор возвращает text вместо point для столбца 4.) Как и в случае с базовыми типами, никакие явные приведения автоматически не добавляются, возможны только приведения присваивания или неявные.

Ту же функцию можно определить другим способом:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Здесь мы записали `SELECT`, который возвращает один столбец нужного составного типа. В данной ситуации этот вариант на самом деле не лучше, но в некоторых случаях он может быть удобной альтернативой — например, если нам нужно вычислить результат, вызывая другую функцию, которая возвращает нужное составное значение. Этот вариант полезен и в случае, когда мы хотим написать функцию, которая возвращает не обычный составной тип, а домен, определённый поверх составного типа; тогда она в любом случае должна определяться как возвращающая единственный столбец, так как никакого способа осуществить нужное приведение для всей строки результата нет.

Мы можем вызывать эту функцию напрямую, либо указав её в выражении значения:

```
SELECT new_emp();

         new_emp
--------------------------
 (None,1000.0,25,"(2,2)")
```

либо обратившись к ней, как к табличной функции:

```
SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

Второй способ более подробно описан в Подразделе 36.5.8.

Когда используется функция, возвращающая составной тип, может возникнуть желание получить из её результата только одно поле (атрибут). Это можно сделать, применяя такую запись:

```
SELECT (new_emp()).name;

 name
------
```

```
 None
```

Дополнительные скобки необходимы во избежание неоднозначности при разборе запроса. Если вы попытаетесь выполнить запрос без них, вы получите ошибку:

```
SELECT new_emp().name;
ERROR:  syntax error at or near "."
LINE 1: SELECT new_emp().name;
                        ^
```

(ОШИБКА: синтаксическая ошибка (примерное положение: "."))

Функциональную запись также можно использовать и для извлечения атрибутов:

```
SELECT name(new_emp());

 name
------
 None
```

Как рассказывалось в Подразделе 8.16.5, запись с указанием поля и функциональная запись являются равнозначными.

Ещё один вариант использования функции, возвращающей составной тип, заключается в передаче её результата другой функции, которая принимает этот тип строки на вход:

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

36.5.4. Функции SQL с выходными параметрами

Альтернативный способ описать результаты функции — определить её с *выходными параметрами*, как в этом примере:

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

Это по сути не отличается от версии `add_em`, показанной в Подразделе 36.5.2. Действительная ценность выходных параметров в том, что они позволяют удобным способом определить функции, возвращающие несколько столбцов. Например:

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

Фактически здесь мы определили анонимный составной тип для результата функции. Показанный выше пример даёт тот же конечный результат, что и команды:

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

Но предыдущий вариант зачастую удобнее, так как он не требует отдельно заниматься определением составного типа. Заметьте, что имена, назначаемые выходным параметрам, не просто декоративные, а определяют имена столбцов анонимного составного типа. (Если вы опустите имя выходного параметра, система выберет имя сама.)

Заметьте, что выходные параметры не включаются в список аргументов при вызове такой функции из SQL. Это объясняется тем, что Postgres Pro определяет сигнатуру вызова функции, рассматривая только входные параметры. Это также значит, что при таких операциях, как удаление функции, в ссылках на функцию учитываются только типы входных параметров. Таким образом, удалить эту конкретную функцию можно любой из этих команд:

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

Параметры функции могут быть объявлены как `IN` (по умолчанию), `OUT`, `INOUT` или `VARIADIC`. Параметр `INOUT` действует как входной (является частью списка аргументов при вызове) и как выходной (часть типа записи результата). Параметры `VARIADIC` являются входными, но обрабатывается специальным образом, как описано далее.

36.5.5. Процедуры SQL с выходными параметрами

Процедуры также поддерживают выходные параметры, но несколько иначе, чем функции. В командах `CALL` выходные параметры должны быть включены в список аргументов. Например, описанную ранее операцию списания средств с банковского счета можно записать следующим образом:

```
CREATE PROCEDURE tp1 (accountno integer, debit numeric, OUT new_balance numeric) AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tp1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

Для вызова этой процедуры необходимо задать аргумент, соответствующий параметру `OUT`. Обычно принято писать `NULL`:

```
CALL tp1(17, 100.0, NULL);
```

Если вы пишете что-то ещё, то это должно быть выражение, неявно приводимое к объявленному типу параметра, как и для входных параметров. Однако обратите внимание, что такое выражение не вычисляется.

Вызывая процедуру из PL/pgSQL, вместо `NULL` вы должны написать переменную, которая будет получать результат процедуры. За подробностями обратитесь к Подразделу 41.6.3.

36.5.6. Функции SQL с переменным числом аргументов

Функции SQL могут быть объявлены как принимающие переменное число аргументов, с условием, что все «необязательные» аргументы имеют один тип данных. Необязательные аргументы будут переданы такой функции в виде массива. Для этого в объявлении функции последний параметр помечается как `VARIADIC`; при этом он должен иметь тип массива. Например:

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

```
SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
     -1
(1 row)
```

По сути, все фактические аргументы, начиная с позиции `VARIADIC`, собираются в одномерный массив, как если бы вы написали

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- это не будет работать
```

На самом деле так вызвать эту функцию нельзя, или, по крайней мере, это не будет соответствовать определению функции. Параметру `VARIADIC` соответствуют одно или несколько вхождений типа его элемента, но не его собственного типа.

Но иногда бывает полезно передать функции с переменными параметрами уже подготовленный массив; особенно когда одна функция с переменными параметрами хочет передавать свой массив параметров другой. Также это более безопасный способ вызывать такую функцию, существующую в схеме, где могут создавать объекты недоверенные пользователи; см. Раздел 10.3. Это можно сделать, добавив `VARIADIC` в вызов:

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

Это предотвращает разворачивание переменного множества параметров функции в базовый тип, что позволяет сопоставить с ним значение типа массива. `VARIADIC` можно добавить только к последнему фактическому аргументу вызова функции.

Также указание `VARIADIC` даёт единственную возможность передать пустой массив функции с переменными параметрами, например, так:

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

Простой вызов `SELECT mleast()` не будет работать, так как переменным параметрам должен соответствовать минимум один фактический аргумент. (Можно определить вторую функцию с таким же именем `mleast`, но без параметров, если вы хотите выполнять такие вызовы.)

Элементы массива, создаваемые из переменных параметров, считаются не имеющими собственных имён. Это означает, что передать функции с переменными параметрами именованные аргументы нельзя (см. Раздел 4.3), если только при вызове не добавлено `VARIADIC`. Например, этот вариант будет работать:

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

А эти варианты нет:

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

36.5.7. Функции SQL со значениями аргументов по умолчанию

Функции могут быть объявлены со значениями по умолчанию для некоторых или всех входных аргументов. Значения по умолчанию подставляются, когда функция вызывается с недостаточным количеством фактических аргументов. Так как аргументы можно опускать только с конца списка фактических аргументов, все параметры после параметра со значением по умолчанию также получат значения по умолчанию. (Хотя запись с именованными аргументами могла бы ослабить это ограничение, оно всё же остаётся в силе, чтобы позиционные ссылки на аргументы оставались действительными.) Независимо от того, используете вы эту возможность или нет, она требует осторожности при вызове функций в базах данных, где одни пользователи не доверяют другим; см. Раздел 10.3.

Например:

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
```

```
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)

SELECT foo();  -- не работает из-за отсутствия значения по умолчанию для первого
 аргумента
ERROR:  function foo() does not exist
```

(ОШИБКА: функция foo() не существует) Вместо ключевого слова `DEFAULT` можно использовать знак =.

36.5.8. Функции SQL, порождающие таблицы

Все функции SQL можно использовать в предложении `FROM` запросов, но наиболее полезно это для функций, возвращающих составные типы. Если функция объявлена как возвращающая базовый тип, она возвращает таблицу с одним столбцом. Если же функция объявлена как возвращающая составной тип, она возвращает таблицу со столбцами для каждого атрибута составного типа.

Например:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

Как показывает этот пример, мы можем работать со столбцами результата функции так же, как если бы это были столбцы обычной таблицы.

Заметьте, что мы получаем из данной функции только одну строку. Это объясняется тем, что мы не использовали указание `SETOF`. Оно описывается в следующем разделе.

36.5.9. Функции SQL, возвращающие множества

Когда SQL-функция объявляется как возвращающая `SETOF` *`некий_тип`*, конечный запрос функции выполняется до завершения и каждая строка выводится как элемент результирующего множества.

Это обычно используется, когда функция вызывается в предложении `FROM`. В этом случае каждая строка, возвращаемая функцией, становится строкой таблицы, появляющейся в запросе. Например, в предположении, что таблица `foo` имеет то же содержимое, что и раньше, мы выполняем:

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

Тогда в ответ мы получим:

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

Также возможно выдать несколько строк со столбцами, определяемыми выходными параметрами, следующим образом:

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

Здесь ключевая особенность заключается в записи `RETURNS SETOF record`, показывающей, что функция возвращает множество строк вместо одной. Если существует только один выходной параметр, укажите тип этого параметра вместо `record`.

Часто бывает полезно сконструировать результат запроса, вызывая функцию, возвращающую множество, несколько раз, передавая при каждом вызове параметры из очередных строк таблицы или подзапроса. Для этого рекомендуется применить ключевое слово `LATERAL`, описываемое в Подразделе 7.2.1.5. Ниже приведён пример использования функции, возвращающей множество, для перечисления элементов древовидной структуры:

```
SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
```

```
 SubChild2 | Child1
(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name  |   child
--------+-----------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

В этом примере не делается ничего такого, что мы не могли бы сделать, применив простое соединение, но для более сложных вычислений возможность поместить некоторую логику в функцию может быть весьма удобной.

Функции, возвращающие множества, могут также вызываться в списке выборки запроса. Для каждой строки, которая генерируется самим запросом, вызывается функция, возвращающая множество, и для каждого элемента набора её результатов генерируется отдельная строка. Предыдущий пример можно было бы также переписать с применением запросов следующим образом:

```
SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, listchildren(name) FROM nodes;
  name  | listchildren
--------+--------------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

Заметьте, что в последней команде `SELECT` для `Child2`, `Child3` и т. д. строки не выдаются. Это происходит потому, что `listchildren` возвращает пустое множество для этих аргументов, так что строки результата не генерируются. Это же поведение мы получаем при внутреннем соединении с результатом функции с применением `LATERAL`.

Поведение Postgres Pro с функциями, возвращающими множества, в списке выборки запроса практически не отличается от поведения с такими функциями, помещёнными в предложение `LATERAL FROM`. Например, запрос:

```
SELECT x, generate_series(1,5) AS g FROM tab;
```

почти равнозначен

```
SELECT x, g FROM tab, LATERAL generate_series(1,5) AS g;
```

Он мог быть полностью идентичным, но в данном конкретном примере планировщик может решить перенести `g` во внешнюю сторону соединения, так как `g` не имеет фактической зависимости по времени вычисления от `tab`. Такое решение привело бы к изменению порядка строк. Функции, возвращающие множества, в списке выборки всегда вычисляются так, как они вычислялись бы внутри соединения с вложенным циклом с остальным предложением `FROM`, так что эти функции выполняются до завершения прежде чем начинается рассмотрение следующей строки из предложения `FROM`.

Если в списке выборки запроса используются несколько функций, возвращающих запросы, они вычисляются примерно так же, как если бы они были помещены в один элемент `LATERAL ROWS FROM( ... )` предложения `FROM`. Для каждой строки из нижележащего запроса выдаётся строка с первым результатом каждой функции, а затем строка со вторым результатом и так далее. Если какие-либо из функций, возвращающих множества, выдают меньше результатов, чем другие, то вместо недостающих данных подставляются значения NULL, так что общее число строк, выдаваемых для одной нижележащей строки, равно числу строк, которое выдаёт функция с наибольшим количеством строк в возвращаемом множестве. Таким образом, функции, возвращающие множества, выполняются совместно, пока все их множества не будут исчерпаны, а затем выполнение продолжается со следующей нижележащей строкой.

Функции, возвращающие множества, могут быть вложенными в списке выборки, но это не допускается в элементах предложения `FROM`. В таких случаях каждый уровень вложенности обрабатывается отдельно, как если бы это был отдельный элемент `LATERAL ROWS FROM( ... )`. Например, в

```
SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;
```

возвращающие множества функции `srf2`, `srf3` и `srf5` будут выполняться совместно для каждой строки `tab`, а затем `srf1` и `srf4` будут совместно применяться к каждой строке, произведённой нижними функциями.

Функции, возвращающие множества, нельзя использовать в конструкциях, вычисляемых по условию, например, `CASE` или `COALESCE`. Например, рассмотрите запрос

```
SELECT x, CASE WHEN x > 0 THEN generate_series(1, 5) ELSE 0 END FROM tab;
```

Может показаться, что он должен выдать пять экземпляров входных строк, в которых `x > 0`, и по одному экземпляру остальных строк; но на деле, так как `generate_series(1, 5)` будет выполняться в неявном элементе `LATERAL FROM` до того, как выражение `CASE` вообще будет рассматриваться, должно было бы выдаваться пять экземпляров абсолютно всех выходных строк. Во избежание путаницы в таких случаях выдаётся ошибка при разборе запроса.

Примечание

Если последняя команда функции — `INSERT`, `UPDATE`, `DELETE` или `MERGE` с `RETURNING`, эта команда будет всегда выполняться до завершения, даже если функция не объявлена с указанием `SETOF` или вызывающий запрос не выбирает все строки результата. Все дополнительные строки, выданные предложением `RETURNING`, просто игнорируются, но соответствующие изменения в таблице всё равно произойдут (и будут завершены до выхода из функции).

Примечание

В Postgres Pro до версии 10 при помещении нескольких функций, возвращающих множества, в один список выборки поведение было не очень разумным, если они возвращали не одинаковое число строк. В таких случаях число выходных строк равнялось наименьшему общему множителю количеств строк, возвращаемых этими функциями. Также и вложенные функции, возвращающие множества, работали не так, как описано выше; у такой функции мог быть максимум один аргумент, возвращающий множество, и каждая вложенность вычислялась независимо. Кроме того, ранее допускалось и условное выполнение (вычисление таких функций внутри `CASE` и т. п.), что ещё больше всё усложняло. При написании запросов, которые должны работать и со старыми версиями Postgres Pro, рекомендуется использовать синтаксис `LATERAL`, так как это гарантирует одинаковый результат с разными версиями. Если в вашем запросе используется условное вычисление функции, возвращающей множество, его можно исправить, переместив проверку условия в специально созданную функцию, возвращающую множество. Например:

```
SELECT x, CASE WHEN y > 0 THEN generate_series(1, z) ELSE 5 END FROM tab;
```

можно заменить на

```
CREATE FUNCTION case_generate_series(cond bool, start int, fin int, els int)
  RETURNS SETOF int AS $$
BEGIN
  IF cond THEN
    RETURN QUERY SELECT generate_series(start, fin);
  ELSE
    RETURN QUERY SELECT els;
  END IF;
END$$ LANGUAGE plpgsql;

SELECT x, case_generate_series(y > 0, 1, z, 5) FROM tab;
```

Это будет работать одинаково во всех версиях Postgres Pro.

36.5.10. Функции SQL, возвращающие таблицы (`TABLE`)

Есть ещё один способ объявить функцию, возвращающую множества, — использовать синтаксис `RETURNS TABLE(`*столбцы*`)`. Это равнозначно использованию одного или нескольких параметров `OUT` с объявлением функции как возвращающей `SETOF record` (или `SETOF` тип единственного параметра, если это применимо). Этот синтаксис описан в последних версиях стандарта SQL, так что этот вариант может быть более портируемым, чем `SETOF`.

Например, предыдущий пример с суммой и произведением можно также переписать так:

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Запись `RETURNS TABLE` не позволяет явно указывать `OUT` и `INOUT` для параметров — все выходные столбцы необходимо записать в списке `TABLE`.

36.5.11. Полиморфные функции SQL

Функции SQL могут быть объявлены как принимающие и возвращающие полиморфные типы, описанные в Подразделе 36.2.5. В следующем примере полиморфная функция `make_array` создаёт массив из двух элементов произвольных типов:

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

Обратите внимание на приведение типа `'a'::text`, определяющее, что аргумент имеет тип `text`. Оно необходимо, если аргумент задаётся просто строковой константой, так как иначе он будет воспринят как имеющий тип `unknown`, а массив типов `unknown` является недопустимым. Без этого приведения вы получите такую ошибку:

```
ERROR:  could not determine polymorphic type because input has type unknown
```

(ОШИБКА: не удалось определить полиморфный тип, так как входные аргументы имеют тип unknown)

С показанным выше объявлением `make_array` этой функции нужно предоставить два аргумента, имеющих один и тот же тип данных; система не будет пытаться совмещать различные типы. Так, например, следующий вызов не будет работать:

```
SELECT make_array(1, 2.5) AS numericarray;
ERROR:  function make_array(integer, numeric) does not exist
```

(ОШИБКА: функция make_array(integer, numeric) не существует) Альтернативный подход заключается в использовании «общего» семейства полиморфных типов; в этом случае система попытается найти подходящий общий тип:

```
CREATE FUNCTION make_array2(anycompatible, anycompatible)
RETURNS anycompatiblearray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array2(1, 2.5) AS numericarray;
 numericarray
--------------
 {1,2.5}
(1 row)
```

Так как правила нахождения общего типа по умолчанию сводятся к выбору типа `text`, когда все аргументы имеют неизвестные типы, работать будет и следующий вызов:

```
SELECT make_array2('a', 'b') AS textarray;
 textarray
-----------
 {a,b}
(1 row)
```

Функция с полиморфными аргументами может иметь фиксированный тип результата, однако обратное не допускается. Например:

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A result of type anyelement requires at least one input of type anyelement,
 anyarray, anynonarray, anyenum, or anyrange.
```

(ОШИБКА: не удалось определить тип результата; ПОДРОБНОСТИ: Для результата типа anyelement требуется минимум один аргумент типа anyelement, anyarray, anynonarray, anyenum или anyrange.)

Полиморфизм можно применять и с функциями, имеющими выходные аргументы. Например:

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |    f3
----+---------
 22 | {22,22}
(1 row)
```

Полиморфизм также можно применять с функциями с переменными параметрами. Например:

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
(1 row)
```

36.5.12. Функции SQL с правилами сортировки

Когда функция SQL принимает один или несколько параметров сортируемых типов данных, правило сортировки определяется при каждом вызове функции, в зависимости от правил сортировки, связанных с фактическими аргументами, как описано в Разделе 22.2. Если правило сортировки определено успешно (то есть не возникло конфликтов между неявно установленными правилами сортировки аргументов), оно неявно назначается для всех сортируемых параметров. Выбранное правило будет определять поведение операций, связанных с сортировкой, в данной функции. Например, для показанной выше функции `anyleast`, результат

```
SELECT anyleast('abc'::text, 'ABC');
```

будет зависеть от правила сортировки по умолчанию, заданного в базе данных. С локалью `C` результатом будет строка `ABC`, но со многими другими локалями это будет `abc`. Нужное правило сортировки можно установить принудительно, добавив предложение `COLLATE` к одному из аргументов функции, например:

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

С другой стороны, если вы хотите, чтобы функция работала с определённым правилом сортировки, вне зависимости от того, с каким она была вызвана, вставьте предложения `COLLATE` где требуется в определении функции. Эта версия `anyleast` всегда будет сравнивать строки по правилам локали `en_US`:

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

Но заметьте, что при попытке применить правило к несортируемому типу данных, возникнет ошибка.

Если для фактических аргументов не удаётся определить общее правило сортировки, функция SQL считает, что им назначено правило сортировки по умолчанию для их типа данных (обычно это то же правило сортировки, что определено по умолчанию для базы данных, но оно может быть и другим для параметров доменных типов).

Поведение сортируемых параметров можно воспринимать как ограниченную форму полиморфизма, применимую только к текстовым типам данных.

36.6. Перегрузка функций

Вы можете определить несколько функций с одним именем SQL, если эти функции будут принимать разные аргументы. Другими словами, имена функций можно *перегружать*. Независимо от того, используете вы эту возможность или нет, она требует предосторожности при вызове функций в базах данных, где одни пользователи не доверяют другим; см. Раздел 10.3. Когда выполняется запрос, сервер определяет, какую именно функцию вызывать, по количеству и типам представленных аргументов. Перегрузка может быть полезна для имитации функций с переменным количеством аргументов, до какого-то конечного числа.

Создавая семейство перегруженных функций, необходимо не допускать неоднозначности. Например, если созданы функции:

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

не вполне понятно, какая функция будет вызвана с довольно простыми аргументами вроде `test(1, 1.5)`. Реализованные в данный момент правила разрешения типов описаны в Главе 10, но разрабатывать систему, которая будет незаметно полагаться на такие особенности, неразумно.

Функции, принимающей один аргумент составного типа, обычно не следует давать имя, совпадающее с именем какого-либо атрибута (поля) этого типа. Вспомните, что запись `*атрибут*(*таблица*)` считается равнозначной `*таблица*.*атрибут*`. В случае, когда возникает неоднозначность между функцией, принимающей составной тип, и атрибутом составного типа, всегда будет выбираться атрибут. Этот выбор можно переопределить, дополнив имя функции схемой (то есть, записав `*схема*.*функция*(*таблица*)`), но лучше избежать этой проблемы, подобрав разные имена.

Другой тип конфликта возможен между обычными функциями и функциями с переменными параметрами. Например, можно создать функции `foo(numeric)` и `foo(VARIADIC numeric[])`. В этом случае будет непонятно, какая функция должна выбираться при передаче одного числового аргумента, например `foo(10.1)`. При разрешении этого конфликта предпочтение отдаётся функции, найденной первой по пути поиска, либо, если две функции находятся в одной схеме, выбирается функция с постоянными аргументами.

При перегрузке функций на языке C есть дополнительное ограничение: имя уровня C каждой функции в семействе перегруженных функций должно отличаться от имён уровня C всех других функций, как внутренних, так и загружаемых динамически. Если это правило нарушается, поведение зависит от среды. Вы можете получить ошибку компоновщика во время выполнения, либо будет вызвана не та функция (обычно внутренняя). Альтернативная форма предложения `AS` для SQL-команды `CREATE FUNCTION` позволяет отвязать имя SQL-функции от имени, определённого в исходном коде на C. Например:

```
CREATE FUNCTION test(int) RETURNS int
    AS 'имя_файла', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
```

```
    AS 'имя_файла', 'test_2arg'
    LANGUAGE C;
```

Имена функций на C здесь следуют одному из множества возможных соглашений.

36.7. Категории изменчивости функций

Для каждой функции определяется характеристика *изменчивости*, с возможными вариантами: `VOLATILE`, `STABLE` и `IMMUTABLE`. Если эта характеристика не задаётся явно в команде `CREATE FUNCTION`, по умолчанию подразумевается `VOLATILE`. Категория изменчивости представляет собой обещание некоторого поведения функции для оптимизатора:

- Изменчивая функция (`VOLATILE`) может делать всё, что угодно, в том числе, модифицировать базу данных. Она может возвращать различные результаты при нескольких вызовах с одинаковыми аргументами. Оптимизатор не делает никаких предположений о поведении таких функций. В запросе, использующем изменчивую функцию, она будет вычисляться заново для каждой строки, когда потребуется её результат.
- Стабильная функция (`STABLE`) не может модифицировать базу данных и гарантированно возвращает одинаковый результат, получая одинаковые аргументы, для всех строк в одном операторе. Эта характеристика позволяет оптимизатору заменить множество вызовов этой функции одним. В частности, выражение, содержащее такую функцию, можно безопасно использовать в условии поиска по индексу. (Так как при поиске по индексу целевое значение вычисляется только один раз, а не для каждой строки, использовать функцию с характеристикой `VOLATILE` в условии поиска по индексу нельзя.)
- Постоянная функция (`IMMUTABLE`) не может модифицировать базу данных и гарантированно всегда возвращает одинаковые результаты для одних и тех же аргументов. Эта характеристика позволяет оптимизатору предварительно вычислить функцию, когда она вызывается в запросе с постоянными аргументами. Например, запрос вида `SELECT ... WHERE x = 2 + 2` можно упростить до `SELECT ... WHERE x = 4`, так как нижележащая функция оператора сложения помечена как `IMMUTABLE`.

Для наилучших результатов оптимизации, функции следует назначать самую строгую характеристику изменчивости, которой она соответствует.

Любая функция с побочными эффектами *должна* быть помечена как `VOLATILE`, чтобы обращения к ней не исключались при оптимизации. Даже если функция не имеет побочных эффектов, её нужно пометить как `VOLATILE`, если её значение может меняться при выполнении одного запроса; таковы функции `random()`, `currval()` и `timeofday()`.

Другой важный пример представляет семейство функций `current_timestamp`, которые имеют характеристику `STABLE`, потому что их значения не меняются в рамках одной транзакции.

Характеристики `STABLE` и `IMMUTABLE` мало различаются, когда речь идёт о простых интерактивных запросах, которые планируются и сразу же выполняются; не имеет большого значения, будет ли функция выполнена однократно на этапе планирования или в начале выполнения. Существенное различие проявляется, когда план сохраняется и многократно используется позже. Если функция помечена как `IMMUTABLE`, тогда как на самом деле она не является постоянной, она может быть сведена к константе во время планирования, так что при последующих выполнениях плана вместо неё будет использоваться неактуальное значение. Это опасно при использовании подготовленных операторов или языков функций, кеширующих планы (например, PL/pgSQL).

У функций, написанных на SQL или на любом другом стандартном процедурном языке, есть ещё одно важное свойство, определяемое характеристикой изменчивости, а именно видимость изменений, произведённых командой SQL, которая вызывает эту функцию. Функция `VOLATILE` будет видеть такие изменения, тогда как `STABLE` и `IMMUTABLE` — нет. Это поведение реализуется посредством снимков в MVCC (см. Главу 13): `STABLE` и `IMMUTABLE` используют снимок, полученный в начале вызывающего запроса, тогда как функции `VOLATILE` получают свежий снимок в начале каждого запроса, который они выполняют.

Примечание

Функции, написанные на C, могут работать со снимками как угодно, но обычно лучше сделать так, чтобы они действовали аналогично.

Вследствие такой организации работы со снимками, функцию, содержащую только команды `SELECT`, можно безопасно пометить как `STABLE`, даже если она выбирает данные из таблиц, которые могут быть изменены параллельными запросами. Postgres Pro выполнит все команды в функции `STABLE` со снимком, полученным для вызывающего запроса, так что они будут видеть одно представление базы данных на протяжении всего запроса.

То же самое поведение со снимками распространяется на команды `SELECT` в функциях `IMMUTABLE`. Вообще в функциях `IMMUTABLE` обычно неразумно выбирать данные из таблиц, так как «постоянство» функции будет нарушено, если содержимое таблиц изменится. Однако Postgres Pro не принуждает вас явно отказаться от этого.

Одна из распространённых ошибок — помечать функцию как `IMMUTABLE`, при том, что её результаты зависят от параметра конфигурации. Например, функция, работающая с временем, может выдавать результаты, зависящие от параметра TimeZone. Для надёжности такие функции следует помечать как `STABLE`.

Примечание

Postgres Pro требует, чтобы функции `STABLE` и `IMMUTABLE` не содержали SQL-команд, кроме `SELECT`, для предотвращения модификации данных. (Это не совсем непробиваемое ограничение, так как эти функции всё же могут вызывать функции `VOLATILE`, способные модифицировать базу данных. Если вы реализуете такую схему, вы увидите, что функция `STABLE` и `IMMUTABLE` не замечает изменений в базе данных, произведённых вызванной функцией, так как они не проявляются в её снимке данных.)

36.8. Функции на процедурных языках

Postgres Pro позволяет разрабатывать собственные функции и на языках, отличных от SQL и C. Эти другие языки в целом обычно называются *процедурными языками* (PL, Procedural Languages). Процедурные языки не встроены в сервер Postgres Pro; они предлагаются загружаемыми модулями. За дополнительной информацией обратитесь к Главе 40 и следующим главам.

36.9. Внутренние функции

Внутренние функции — это функции, написанные на языке C, и статически скомпонованные в исполняемый код сервера Postgres Pro. В «теле» определения функции задаётся имя функции на уровне C, которое не обязательно должно совпадать с именем, объявленным для использования в SQL. (Обратной совместимости ради, тело функции может быть пустым, что будет означать, что имя функции на уровне C совпадает с именем в SQL.)

Обычно все внутренние функции, представленные на сервере, объявляются в ходе инициализации кластера баз данных (см. Раздел 17.2), но пользователь может воспользоваться командой `CREATE FUNCTION` и добавить дополнительные псевдонимы для внутренней функции. Внутренние функции объявляются в `CREATE FUNCTION` с именем языка `internal`. Например, так можно создать псевдоним для функции `sqrt`:

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

(Большинство внутренних функций должны объявляться как «strict».)

Примечание

Не все «предопределённые» функции являются «внутренними» в вышеописанном смысле. Некоторые предопределённые функции написаны на SQL.

36.10. Функции на языке C

Пользовательские функции могут быть написаны на C (или на языке, который может быть совместим с C, например C++). Такие функции компилируются в динамически загружаемые объекты (также называемые разделяемыми библиотеками) и загружаются сервером по требованию. Именно метод динамической загрузки отличает функции «на языке C» от «внутренних» функций — правила написания кода по сути одни и те же. (Собственно, поэтому стандартная библиотека внутренних функций может быть богатым источником примеров для написания собственных функций на языке C.)

В настоящее время для функций на C применяется только одно соглашение о вызовах («версии 1»). Поддержка этого соглашения обозначается объявлением функции с макросом `(PG_FUNCTION_INFO_V1)`, как показано ниже.

36.10.1. Динамическая загрузка

В первый раз, когда в сеансе вызывается пользовательская функция в определённом внешнем объектном файле, загрузчик динамических модулей загружает этот файл в память, чтобы можно было вызвать эту функцию. Таким образом, в команде `CREATE FUNCTION`, объявляющей пользовательскую функцию на языке C, необходимо определить две сущности для функции: имя загружаемого объектного файла и имя уровня C (символ для компоновки) заданной функции в этом объектном файле. Если имя уровня C не указано явно, предполагается, что оно совпадает с именем функции в SQL.

Для нахождения разделяемого объектного файла по имени, заданному в команде `CREATE FUNCTION`, применяется следующий алгоритм:

1. Если имя задаётся абсолютным путём, загружается заданный файл.
2. Если имя начинается со строки `$libdir`, эта часть пути заменяется путём к каталогу библиотек Postgres Pro, который определяется во время сборки.
3. Если в имени не указывается каталог, поиск файла производится по пути, заданному конфигурационной переменной dynamic_library_path.
4. В противном случае (файл не был найден в пути поиска, или в его имени указывается не абсолютный путь к каталогу), загрузчик попытается принять имя как есть, что, скорее всего, не увенчается успехом. (Полагаться на текущий рабочий каталог ненадёжно.)

Если эта последовательность не даёт положительный результат, к данному имени добавляется принятое на данной платформе расширение файлов библиотек (часто `.so`) и последовательность повторяется снова. Если и это не приводит к успеху, происходит сбой загрузки.

Для поиска разделяемых библиотек рекомендуется задавать либо путь относительно `$libdir`, либо путь динамических библиотек. Это упрощает обновление версии при перемещении новой инсталляции в другое место. Какой именно каталог подразумевается под `$libdir`, можно узнать с помощью команды `pg_config --pkglibdir`.

Пользователь, от имени которого работает сервер Postgres Pro, должен иметь возможность пройти путь к файлу, который требуется загрузить. Очень распространённая ошибка — когда сам файл или каталог верхнего уровня оказывается недоступным для чтения и/или исполнения для пользователя postgres.

В любом случае имя файла, заданное в команде `CREATE FUNCTION`, записывается в системные каталоги буквально, так что если этот файл потребуется загрузить ещё раз, та же процедура будет проделана снова.

Примечание

Postgres Pro не будет компилировать функцию на C автоматически, поэтому прежде чем ссылаться на объектный файл в команде `CREATE FUNCTION`, его нужно скомпилировать. За дополнительными сведениями обратитесь к Подразделу 36.10.5.

Чтобы гарантировать, что динамически загружаемый объектный файл не будет загружен несовместимым сервером, Postgres Pro проверяет, содержит ли этот файл «отличительный блок» с требуемым содержимым. Благодаря этому сервер может выявить очевидную несовместимость, например, когда код скомпилирован для другой старшей версии Postgres Pro. Чтобы включить его в свой модуль, напишите это в одном (и только одном) из исходных файлов модуля, после включения заголовочного файла `fmgr.h`:

```
PG_MODULE_MAGIC;
```

После того как он был использован первый раз, динамически загружаемый объектный файл сохраняется в памяти. Следующие обращения в том же сеансе к функциям в этом файле повлекут только небольшие издержки, связанные с поиском в таблице символов. Если вам нужно принудительно перезагрузить объектный файл, например, после перекомпиляции, начните новый сеанс.

Динамически загружаемый файл может дополнительно содержать функции инициализации и завершения работы библиотеки. Если в файле находится функция с именем `_PG_init`, эта функция будет вызвана сразу после загрузки файла. Эта функция не принимает параметры и не должна ничего возвращать. В настоящее время возможности выгрузить динамически загруженный файл нет.

36.10.2. Базовые типы в функциях на языке C

Чтобы понимать, как написать функцию на языке C, вы должны знать, как внутри Postgres Pro представляются базовые типы данных и как их могут принимать и передавать функции. Postgres Pro внутри воспринимает базовые типы как «блоки памяти». Пользовательские функции, устанавливаемые для типов, в свою очередь, определяют, как Postgres Pro может работать с этими типами. То есть, Postgres Pro только сохраняет и загружает данные с диска, а для ввода, обработки и вывода данных он использует определяемые вами функции.

Базовые типы могут иметь один из трёх внутренних форматов:

- передаётся по значению, фиксированной длины
- передаётся по ссылке, фиксированной длины
- передаётся по ссылке, переменной длины

Типы, передаваемые по значению, могут иметь размер только 1, 2 или 4 байта (и 8 байт, если `sizeof(Datum)` равен 8 на вашей машине). Определяя собственные типы, следует позаботиться о том, чтобы они имели одинаковый размер (в байтах) во всех архитектурах. Например, тип `long` опасен, так как он имеет размер 4 байта на одних машинах, и 8 байт на других, тогда как тип `int` состоит из 4 байт в большинстве систем Unix. Поэтому разумной реализацией типа `int4` на платформе Unix может быть такая:

```
/* 4-байтное целое, передаётся по значению */
typedef int int4;
```

(В коде собственно Postgres Pro этот тип называется `int32`, так как в C принято соглашение, что `int`*`XX`* подразумевает *`XX` бит*. Заметьте, что вследствие этого тип `int8` в C имеет размер 1 байт. Тип `int8`, принятый в SQL, в C называется `int64`. См. также Таблицу 36.2.)

С другой стороны, типы фиксированной длины любого размера можно передавать по ссылке. Например, взгляните на пример реализации типа Postgres Pro:

```
/* 16-байтная структура, передаётся по ссылке */
typedef struct
{
    double  x, y;
```

```
} Point;
```

В функции Postgres Pro и из них могут передаваться только указатели на такие типы. Чтобы вернуть значение такого типа, выделите для него нужное количество памяти функцией `palloc`, заполните выделенную память и верните указатель на неё. (Если вы захотите просто вернуть то же значение, что было получено во входном аргументе этого же типа данных, вы можете пропустить дополнительный вызов `palloc` и просто вернуть указатель на это поступившее значение.)

Наконец, все типы переменной длины также должны передаваться по ссылке. Все типы переменной длины должны начинаться с обязательного поля длины размером ровно 4 байта, которая будет задаваться макросом `SET_VARSIZE`; никогда не устанавливайте это поле вручную! Все данные, которые будут храниться в этом типе, должны размещаться в памяти непосредственно за этим полем длины. Поле длины содержит полную длину структуры, то есть включает размер самого поля длины.

Ещё один важный момент — старайтесь не оставлять неинициализированных байт в значениях данных; например, обнуляйте все возможные байты выравнивания, которые могут присутствовать в структурах. Если этого не делать, логически равные значения ваших данных могут представляться неравными планировщику, что приведёт к построению неэффективных (хотя и корректных) планов.

Предупреждение

Никогда не изменяйте содержимое, передаваемое на вход по ссылке. Если вы сделаете это, вы скорее всего испортите данные на диске, так как полученный вами указатель указывает непосредственно на место в дисковом буфере. Единственное исключение из этого правила освещается в Разделе 36.12.

В качестве примера мы можем определить тип `text` так:

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

Запись `[FLEXIBLE_ARRAY_MEMBER]` означает, что действительная длина массива данных в этом объявлении не указывается.

Работая с типами переменной длины, мы должны аккуратно выделить нужный объём памяти и записать его размер в поле длины. Например, если нужно сохранить 40 байт в структуре `text`, можно применить такой код:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

`VARHDRSZ` совпадает с `sizeof(int32)`, но для получения размера заголовка типа переменной длины хорошим стилем считается применять макрос `VARHDRSZ`. Кроме того, поле длины *должно* устанавливаться макросом `SET_VARSIZE`, а не простым присваиванием.

В Таблице 36.2 указано, какие типы языка C соответствуют типам SQL при написании функций на C с использованием встроенных типов Postgres Pro. В столбце «Определён в» указывается, какой заголовочный файл необходимо подключить, чтобы получить определение типа. (Фактическое определение может быть в другом файле, который подключается из указанного, однако рекомендуется придерживаться обозначенного интерфейса.) Заметьте, что в любом исходном файле первым всегда необходимо включать `postgres.h`, потому что в нём объявляется ряд вещей, которые нужны в любом случае, и потому что включение первым другого заголовочного файла может сделать код непереносимым.

Таблица 36.2. Типы C, эквивалентные встроенным типам SQL

Тип SQL	Тип C	Определён в
`boolean`	`bool`	`postgres.h` (может быть встроен в компиляторе)
`box`	`BOX*`	`utils/geo_decls.h`
`bytea`	`bytea*`	`postgres.h`
`"char"`	`char`	(встроен в компиляторе)
`character`	`BpChar*`	`postgres.h`
`cid`	`CommandId`	`postgres.h`
`date`	`DateADT`	`utils/date.h`
`float4` (`real`)	`float4`	`postgres.h`
`float8` (`double precision`)	`float8`	`postgres.h`
`int2` (`smallint`)	`int16`	`postgres.h`
`int4` (`integer`)	`int32`	`postgres.h`
`int8` (`bigint`)	`int64`	`postgres.h`
`interval`	`Interval*`	`datatype/timestamp.h`
`lseg`	`LSEG*`	`utils/geo_decls.h`
`name`	`Name`	`postgres.h`
`numeric`	`Numeric`	`utils/numeric.h`
`oid`	`Oid`	`postgres.h`
`oidvector`	`oidvector*`	`postgres.h`
`path`	`PATH*`	`utils/geo_decls.h`
`point`	`POINT*`	`utils/geo_decls.h`
`regproc`	`RegProcedure`	`postgres.h`
`text`	`text*`	`postgres.h`
`tid`	`ItemPointer`	`storage/itemptr.h`
`time`	`TimeADT`	`utils/date.h`
`time with time zone`	`TimeTzADT`	`utils/date.h`
`timestamp`	`Timestamp`	`datatype/timestamp.h`
`timestamp with time zone`	`TimestampTz`	`datatype/timestamp.h`
`varchar`	`VarChar*`	`postgres.h`
`xid`	`TransactionId`	`postgres.h`

Теперь, когда мы рассмотрели все возможные структуры для базовых типов, мы можем перейти к примерам реальных функций.

36.10.3. Соглашение о вызовах версии 1

Соглашение о вызовах версии 1 полагается на макросы, скрывающие основную долю сложностей, связанных с передачей аргументов и результатов. По соглашению версии 1 функция на C должна всегда определяться так:

```
Datum funcname(PG_FUNCTION_ARGS)
```

В дополнение к этому, в том же исходном файле должен присутствовать вызов макроса:

```
PG_FUNCTION_INFO_V1(funcname);
```

(Обычно его принято записывать непосредственно перед функцией.) Этот вызов макроса не нужен для функций `internal`, так как Postgres Pro предполагает, что все внутренние функции используют соглашении версии 1. Однако для функций, загружаемых динамически, этот макрос необходим.

В функции версии 1 каждый аргумент выбирается макросом `PG_GETARG_xxx()`, который соответствует типу данных аргумента. В нестрогих функциях этому вызову должна предшествовать проверка на NULL в аргументе с использованием `PG_ARGISNULL()` (см. ниже). Результат возвращается макросом `PG_RETURN_xxx()` для возвращаемого типа. `PG_GETARG_xxx()` принимает в качестве параметра номер выбираемого аргумента функции (нумерация начинается с 0). `PG_RETURN_xxx()` принимает фактическое значение, которое нужно возвратить.

Несколько примеров использования соглашения о вызовах версии 1:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"
#include "varatt.h"

PG_MODULE_MAGIC;

/* by value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
```

```
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_PP(0);

    /*
     * VARSIZE_ANY_EXHDR is the size of the struct in bytes, minus the
     * VARHDRSZ or VARHDRSZ_SHORT of its header.  Construct the copy with a
     * full-length header.
     */
    text     *new_t = (text *) palloc(VARSIZE_ANY_EXHDR(t) + VARHDRSZ);
    SET_VARSIZE(new_t, VARSIZE_ANY_EXHDR(t) + VARHDRSZ);

    /*
     * VARDATA is a pointer to the data region of the new struct.  The source
     * could be a short datum, so retrieve its data through VARDATA_ANY.
     */
    memcpy(VARDATA(new_t),          /* destination */
           VARDATA_ANY(t),          /* source */
           VARSIZE_ANY_EXHDR(t));   /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_PP(0);
    text  *arg2 = PG_GETARG_TEXT_PP(1);
    int32 arg1_size = VARSIZE_ANY_EXHDR(arg1);
    int32 arg2_size = VARSIZE_ANY_EXHDR(arg2);
    int32 new_text_size = arg1_size + arg2_size + VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA_ANY(arg1), arg1_size);
    memcpy(VARDATA(new_text) + arg1_size, VARDATA_ANY(arg2), arg2_size);
    PG_RETURN_TEXT_P(new_text);
}
```

В предположении, что приведённый выше код был подготовлен в файле `funcs.c` и скомпилирован в разделяемый объект, мы можем объявить эти функции в Postgres Pro следующими командами:

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'КАТАЛОГ/funcs', 'add_one'
```

```
     LANGUAGE C STRICT;

-- обратите внимание — это перегрузка SQL-функции "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'КАТАЛОГ/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'КАТАЛОГ/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'КАТАЛОГ/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'КАТАЛОГ/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

Здесь *`КАТАЛОГ`* — это путь к каталогу, в который помещён разделяемый библиотечный файл (например, каталог учебных материалов (tutorial) в исходном коде Postgres Pro, содержащий код примеров, использованных в этом разделе). (Лучше было бы просто написать `'funcs'` в предложении `AS`, предварительно добавив *`КАТАЛОГ`* в путь поиска. В любом случае мы можем опустить принятое в системе расширение файлов разделяемых библиотек, обычно `.so`.)

Заметьте, что мы объявили эти функции как «strict» (строгие) — это означает, что система будет автоматически подразумевать результат NULL, если в одном из входных значений передаётся NULL. Благодаря этому, мы избегаем необходимости проверять входные значения на NULL в коде функции. Без такого объявления нам пришлось бы явно проверять параметры на NULL, используя `PG_ARGISNULL()`.

Макрос `PG_ARGISNULL(n)` позволяет функции проверить на NULL каждый из её аргументов. (Разумеется, это нужно делать только в функциях, объявленных без характеристики «strict».) Как и с макросом `PG_GETARG_xxx()`, входные аргументы нумеруются, начиная с нуля. Заметьте, что не следует обращаться к макросу `PG_GETARG_xxx()`, не убедившись, что соответствующий аргумент не NULL. Чтобы возвратить NULL в качестве результата, воспользуйтесь макросом `PG_RETURN_NULL()`; это работает и со строгими, и с нестрогими функциями.

На первый взгляд соглашения о вызовах версии 1 могут показаться всего лишь бессмысленным мракобесием, по сравнению с соглашениями простого `C`. Однако они позволяют работать с аргументами и возвращаемыми значениями, в которых может передаваться `NULL`, а также со значениями в формате TOAST (сжатыми или хранимыми отдельно).

Кроме того, в интерфейсе версии 1 появились две вариации макроса `PG_GETARG_xxx()`. Первая вариация, `PG_GETARG_xxx_COPY()`, гарантированно возвращает копию указанного аргумента, которую можно безопасно модифицировать. (Обычный макрос иногда возвращает указатель на значение, которое физически хранится в таблице, в которую нельзя писать. С макросом `PG_GETARG_xxx_COPY()` гарантированно получается результат, доступный для записи.) Вторая вариация представлена макросом `PG_GETARG_xxx_SLICE()`, принимающим три параметра. В первом передаётся номер аргумента функции (как и раньше). Во втором и третьем передаётся смещение и длина сегмента, который должен быть возвращён. Смещение отсчитывается с нуля, а отрицательная длина указывает, что запрашивается оставшаяся часть значения. Эти макросы дают более эффективный доступ к частям больших значений, имеющим тип хранения «external». (Тип хранения столбца может задаваться командой `ALTER TABLE имя_таблицы ALTER COLUMN имя_столбца SET STORAGE тип_хранения`, где *`тип_хранения`*: `plain`, `external`, `extended` или `main`.)

Наконец, соглашения о вызовах версии 1 позволяют возвращать множества (Подраздел 36.10.8) и реализовывать триггерные функции (Глава 37) и обработчики вызовов процедурных языков (Глава 54).

36.10.4. Написание кода

Прежде чем перейти к более сложным темам, мы должны обсудить некоторые правила написания кода функций на языке C для Postgres Pro. Хотя принципиально можно загружать в Postgres Pro функции, написанные на языках, отличных от C, обычно это довольно сложно (когда вообще возможно), так как другие языки, например C++, FORTRAN или Pascal часто не следуют соглашениям, принятым в C. То есть другие языки могут передавать аргументы и возвращаемые значения между функциями разными способами. Поэтому далее предполагается, что ваши функции на языке C действительно написаны на C.

Основные правила написания и компиляции функций на C таковы:

- Чтобы выяснить, где находятся заголовочные файлы сервера Postgres Pro, установленные в вашей системе (или в системе, с которой будут работать ваши пользователи), воспользуйтесь командой `pg_config --includedir-server`.
- Для компиляции и компоновки кода, который можно будет динамически загрузить в Postgres Pro, требуется указать специальные флаги. Чтобы конкретнее узнать, как это сделать в вашей конкретной операционной системе, обратитесь к Подразделу 36.10.5.
- Не забудьте определить «отличительный блок» для вашей разделяемой библиотеки, как описано в Подразделе 36.10.1.
- Для выделения памяти используйте функцию Postgres Pro `palloc`, а для освобождения `pfree`, вместо соответствующих функций библиотеки C `malloc` и `free`. Память, выделяемая функцией `palloc`, будет автоматически освобождаться в конце каждой транзакции, во избежание утечек памяти.
- Всегда обнуляйте байты ваших структур, применяя `memset` (или сразу выделяйте память функцией `palloc0`). Даже если вы присвоите значение каждому полю структуры, в ней могут оставаться байты выравнивания (пустоты в структуре), содержащие случайные значения. Если исключить это требование, будет сложно поддерживать индексы или соединение по хешу, так как для вычисления хеша придётся выбирать только значащие биты из вашей структуры данных. Планировщик также иногда полагается на побитовое сравнение констант, так что результаты планирования могут оказаться неожиданными, если логически равные значения окажутся неравными на битовом уровне.
- Большинство внутренних типов Postgres Pro объявлены в `postgres.h`, тогда как интерфейс менеджера функций (`PG_FUNCTION_ARGS` и т. д.) определён в `fmgr.h`, так что потребуется подключить как минимум два этих файла. По соображениям портируемости, лучше включить `postgres.h` *первым*, до каких-либо других системных или пользовательских файлов заголовков. При подключении `postgres.h` автоматически также будут подключены `elog.h` и `palloc.h`.
- Имена символов, определённые в объектных файлах, не должны конфликтовать друг с другом или с именами других символов, определённых в исполняемых файлах сервера Postgres Pro. Если вы столкнётесь с ошибками, вызванными таким конфликтом, вам придётся переименовать ваши функции или переменные.

36.10.5. Компиляция и компоновка динамически загружаемых функций

Прежде чем вы сможете использовать ваши написанные на C функции, расширяющие возможности Postgres Pro, их необходимо скомпилировать и скомпоновать особым образом, чтобы сервер мог динамически загрузить полученный файл. Точнее говоря, вам необходимо создать *разделяемую библиотеку*.

За подробной информацией, дополняющей и поясняющей то, что описано в этом разделе, вам следует обратиться к документации вашей операционной системы, в частности к страницам руководства компилятора C (`cc`) и компоновщика (`ld`).

Создание разделяемых библиотек в принципе не отличается от сборки исполняемых файлов: сначала исходные файлы компилируются в объектные, а затем объектные связываются вместе. Объектные файлы должны создаваться так, чтобы они содержали *позиционно-независимый* код (PIC, position-independent code), что означает, что при загрузке для выполнения этот код может быть помещён в любое место в памяти. (Объектные файлы, предназначенные для сборки непосредственно исполняемых файлов, обычно собираются не так.) Команда для компоновки разделяемой библиотеки принимает специальные флаги, что отличают её от компоновки исполняемого файла (по крайней мере в теории — в некоторых системах реальность не так прекрасна).

В следующих примерах предполагается, что исходный код находится в файле `foo.c` и мы будем создавать разделяемую библиотеку `foo.so`. Промежуточный объектный файл будет называться `foo.o`, если не отмечено другое. Разделяемая библиотека может включать больше одного объектного файла, но здесь мы ограничимся одним.

FreeBSD

Для создания кода PIC компилятору передаётся флаг `-fPIC`. Чтобы создать разделяемую библиотеку, используется флаг компилятора `-shared`.

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

Это применимо к FreeBSD версии 13.0, в более старых версиях используется компилятор `gcc`.

Linux

Для создания кода PIC компилятору передаётся флаг `-fPIC`. Для создания разделяемой библиотеки компилятору передаётся флаг `-shared`. Полный пример будет выглядеть так:

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

macOS

Следующий пример показывает нужные команды, в предположении, что установлены инструменты разработчика.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

Для создания кода PIC компилятору передаётся флаг `-fPIC`. Для компоновки разделяемых библиотек в системах ELF компилятору передаётся флаг `-shared`, а в старых системах, не поддерживающих ELF, применяется команда `ld -Bshareable`.

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

Для создания кода PIC компилятору передаётся флаг `-fPIC`, а для компоновки разделяемых библиотек применяется команда `ld -Bshareable`.

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

Для создания кода PIC компилятору Sun передаётся флаг `-KPIC`, а компилятору GCC — флаг `-fPIC`. Для компоновки разделяемой библиотеки можно передать обоим компиляторам флаг `-G` либо передать флаг `-shared` компилятору GCC.

```
cc -KPIC -c foo.c
```

```
cc -G -o foo.so foo.o
```

или

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

Подсказка

Если это слишком сложно для вас, попробуйте использовать средство *GNU Libtool*, которое скрывает различия платформ за единым интерфейсом.

Полученную разделяемую библиотеку можно будет затем загрузить в Postgres Pro. Когда команде `CREATE FUNCTION` передаётся имя файла, это должно быть имя файла разделяемой библиотеки, а не промежуточного объектного файла. Заметьте, что принятое в системе расширение файлов библиотек (как правило, `.so` или `.sl`) в команде `CREATE FUNCTION` можно опустить, и так обычно следует делать для лучшей портируемости.

Чтобы уточнить, где сервер будет искать файлы разделяемых библиотек, вернитесь к Подразделу 36.10.1.

36.10.6. Аргументы составного типа

Составные типы не имеют фиксированного макета данных, как структуры C. В частности, экземпляры составного типа могут содержать поля NULL. Кроме того, в контексте наследования составные типы могут иметь разные поля для разных членов в одной иерархии наследования. Поэтому Postgres Pro предоставляет функциям специальный интерфейс для обращения к полям составных типов из C.

Предположим, что мы хотим написать функцию, отвечающую на запрос:

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

С соглашениями о вызовах версии 1 мы можем определить функцию `c_overpaid` так:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* Alternatively, we might prefer to do PG_RETURN_NULL() for null salary. */

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

`GetAttributeByName` — это системная функция Postgres Pro, которая возвращает атрибуты указанной строки. Она принимает три параметра: аргумент типа `HeapTupleHeader`, имя нужного атрибута и выходной параметр, устанавливаемый, если значение атрибута — NULL. `GetAttributeByName` возвращает значение `Datum`, которое вы можете привести к подходящему типу данных, используя соответствующую функцию `DatumGet`*`XXX`*`()`. Заметьте, что возвращаемое значение недействительно, если установлен флаг null; всегда проверяйте этот флаг, прежде чем что-либо делать с результатом.

Есть также функция `GetAttributeByNum`, которая выбирает целевой атрибут не по имени, а по номеру столбца.

Следующая команда объявляет функцию `c_overpaid` в SQL:

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'КАТАЛОГ/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

Заметьте, что мы использовали `STRICT`, чтобы нам не пришлось проверять входные аргументы на равенство NULL.

36.10.7. Возврат строк (составных типов)

Чтобы вернуть строку или значение составного типа из функции на языке C, можно использовать специальный API, предоставляющий макросы и функции, скрывающие основную сложность формирования составных типов данных. Для использования этого API необходимо включить в исходный файл:

```
#include "funcapi.h"
```

Сформировать значение составного типа (далее «кортеж») можно двумя способами: его можно построить из массива значений Datum, или из массива строк C, которые будут переданы функциям преобразования ввода для типов столбцов кортежа. В любом случае сначала нужно получить или сконструировать дескриптор `TupleDesc` для структуры кортежа. Работая со значениями Datum, вы передаёте `TupleDesc` функции `BlessTupleDesc`, а затем вызываете `heap_form_tuple` для каждой строки. Работая со строками C, вы передаёте `TupleDesc` функции `TupleDescGetAttInMetadata`, а затем для каждой строки вызываете `BuildTupleFromCStrings`. В случае функции, возвращающей множество кортежей, все подготовительные действия можно выполнить один раз при первом вызове функции.

Для получения требуемого дескриптора `TupleDesc` предлагается несколько дополнительных функций. Рекомендованный способ возврата составных значений заключается в вызове функции:

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

При этом в `fcinfo` должна передаваться та же структура, что была передана самой вызывающей функции. (Для этого, конечно, необходимо использовать соглашения о вызовах версии 1.) В `resultTypeId` можно передать `NULL` или адрес локальной переменной, в которую будет записан OID типа результата функции. В `resultTupleDesc` должен передаваться адрес локальной переменной `TupleDesc`. Убедить, что функция возвратила результат `TYPEFUNC_COMPOSITE`; в этом случае в `resultTupleDesc` оказывается требуемая структура `TupleDesc`. (Если получен другой результат, вы можете выдать ошибку с сообщением «функция, возвращающая запись, вызвана в контексте, не допускающем этот тип».)

Подсказка

`get_call_result_type` позволяет получить фактический тип результата полиморфной функции, так что она полезна и в функциях, возвращающих скалярные полиморфные результаты, не только в функциях, возвращающих составные типы. Выходной параметр `resultTypeId` полезен в первую очередь для полиморфных скалярных функций.

Примечание

В дополнение к `get_call_result_type` есть схожая функция `get_expr_result_type`, позволяющая получить ожидаемый тип результата для вызова функции, представленного деревом выражения. Её можно использовать, когда тип результата нужно определить извне самой функции. Есть также функция `get_func_result_type`, которую можно применять, когда известен только OID функции. Однако эти две функции неспособны выдать тип результата функций, возвращающих `record`, а `get_func_result_type` неспособна разрешать полиморфные типы, так что вместо них лучше использовать `get_call_result_type`.

Ранее для получения `TupleDesc` использовались теперь уже устаревшие функции:

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

(возвращает `TupleDesc` для типа строк указанного отношения) и:

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

(возвращает `TupleDesc` для типа, задаваемого по OID). Применяя её, можно получить `TupleDesc` для базового или составного типа. Однако она не подойдёт для функции, возвращающей тип `record`, и не сможет разрешить полиморфные типы.

Получив `TupleDesc`, вызовите:

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

если вы планируете работать со структурами Datum, либо:

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

если планируете работать со строками C. Если вы разрабатываете функцию, возвращающую набор данных, вы можете сохранить результаты этих функций в структуре `FuncCallContext`, в поле `tuple_desc` или `attinmeta`, соответственно.

Если вы работаете со структурами Datum, воспользуйтесь функцией:

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

Она формирует `HeapTuple` из переданных ей данных в форме Datum.

Если вы работаете со строками C, воспользуйтесь функцией:

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

Она формирует `HeapTuple` из переданных ей данных в виде строк C. В параметре *`values`* ей передаётся массив строк C, по одной для каждого атрибута выходной строки. Каждая из этих строк должна иметь формат, принимаемый функцией ввода типа данных атрибута. Чтобы задать значение NULL для одного из этих атрибутов, вместо соответствующего указателя в массиве *`values`* нужно передать `NULL`. Эту функцию нужно вызывать для каждой строки, которую вы будете возвращать.

Получив кортеж, который вы будете возвращать из вашей функции, вы должны преобразовать его в тип `Datum`. Чтобы преобразовать `HeapTuple` в Datum, воспользуйтесь функцией:

```
HeapTupleGetDatum(HeapTuple tuple)
```

Полученный тип `Datum` можно вернуть непосредственно, если должна возвращаться только одна строка, либо использовать как текущее выдаваемое значение в функции, возвращающей набор строк.

Пример приведён в следующем разделе.

36.10.8. Возврат множеств

Функции на языке C могут возвращать наборы данных (множества строк) двумя способами. Первый способ, который называется *ValuePerCall* (значение за вызов), заключается в многократном вызове функции (при этом ей каждый раз передаются одни и те же аргументы). Эта функция при очередном вызове должна возвращать следующую строку, пока не выдаст все строки, о чём она сообщает, возвращая NULL. Таким образом, возвращающая множество функция (Set-Returning Function, SRF) должна сохранять между вызовами своё состояние в достаточном объёме, чтобы помнить, какие данные уже были выданы, и возвращать следующие при очередном вызове. Второй вариант, *Materialize* (Материализация), заключается в формировании в SRF объекта tuplestore, содержащего сразу весь результирующий набор; единственный вызов производится для получения сразу всего результата, и никакое состояние между вызовами сохранять не нужно.

Реализуя режим ValuePerCall, важно не забывать, что выполнение запроса до полного завершения не гарантируется. Так, например, получив указание `LIMIT`, исполнитель запроса может перестать вызывать возвращающую множество функцию до получения всех строк. Это означает, что выполнять действия, связанные с очисткой, в последнем вызове небезопасно, так как он может вовсе не состояться. Поэтому для функций, которым нужно обращаться к внешним ресурсам, например, открывая файловые дескрипторы, рекомендуется использовать режим материализации результатов.

В продолжении этого раздела описываются несколько вспомогательных макросов, которые часто используются (хотя не являются обязательными) в SRF, реализующих метод ValuePerCall.

Для использования описанных здесь макросов поддержки ValuePerCall подключите `funcapi.h`. Эти макросы работают со структурой `FuncCallContext`, содержащей состояние, которое требуется сохранять между вызовами. Указатель на `FuncCallContext` внутри вызываемой SRF сохраняется между вызовами в поле `fcinfo->flinfo->fn_extra`, которое макросы автоматически заполняют при первом использовании, рассчитывая прочесть из него тот же указатель при последующих вызовах.

```
typedef struct FuncCallContext
{
    /*
     * Счётчик числа ранее выполненных вызовов
     *
     * call_cntr сбрасывается в 0 макросом SRF_FIRSTCALL_INIT() и
     * увеличивается на 1 каждый раз, когда вызывается SRF_RETURN_NEXT().
     */
    uint64 call_cntr;

    /*
     * Максимальное число вызовов (может не использоваться)
     *
     * max_calls не является обязательным и присутствует здесь только для удобства.
     * Если это значение не задано, вы должны предоставить другую возможность
 определить,
     * когда функция завершила свою работу.
     */
    uint64 max_calls;

    /*
     * Указатель на разнообразную контекстную информацию,
     * представленную пользователем; (может не использоваться)
     *
     * user_fctx используется как указатель на ваши собственные данные,
     * позволяющий сохранить контекстную информацию между вызовами функции.
     */
```

```
    void *user_fctx;

    /*
     * Указатель на структуру, содержащую метаданные ввода типа атрибута
     * (может не использоваться)
     *
     * attinmeta задействуется, когда возвращаются кортежи (т. е. составные типы
 данных),
     * и не применяется для возврата базовых типов. Он нужен, только если
     * вы планируете использовать BuildTupleFromCStrings() для формирования
 возвращаемого
     * кортежа.
     */
    AttInMetadata *attinmeta;

    /*
     * Контекст памяти, нужный для структур, которые должны сохраняться при нескольких
 вызовах
     *
     * Поле multi_call_memory_ctx заполняется в SRF_FIRSTCALL_INIT() и используется
     * в SRF_RETURN_DONE() для очистки. Это наиболее подходящий контекст
     * для любых блоков памяти, которые должны многократно использоваться при
     * повторных вызовах SRF.
     */
    MemoryContext multi_call_memory_ctx;

    /*
     * Указатель на структуру, содержащую описание кортежа (может не использоваться)
     *
     * tuple_desc задействуется, когда возвращаются кортежи (т. е. составные типы),
     * и нужен только, если вы планируете формировать кортежи с помощью функции
     * heap_form_tuple(), а не BuildTupleFromCStrings().  Заметьте, что сохраняемый
     * здесь указатель TupleDesc обычно должен сначала пройти через вызов
     * BlessTupleDesc().
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

Для SRF предоставляется ряд макросов, использующих эту инфраструктуру:

```
SRF_IS_FIRSTCALL()
```

Используйте этот макрос, чтобы определить, вызывается ли ваша функция в первый раз. При первом вызове (но не при последующих) выполните:

```
SRF_FIRSTCALL_INIT()
```

для того, чтобы инициализировать `FuncCallContext`. При каждом вызове функции, включая первый, выполняйте:

```
SRF_PERCALL_SETUP()
```

для того, чтобы подготовиться к использованию `FuncCallContext`.

Если у вашей функции есть данные, которые она должна выдать в текущем вызове, выполните:

```
SRF_RETURN_NEXT(funcctx, result)
```

для того, чтобы передать их вызывающему. (Переменная `result` должна быть типа `Datum`, либо одним значением, либо кортежем, подготовленным как описано выше.) Наконец, когда ваша функция закончила выдавать данные, выполните:

```
SRF_RETURN_DONE(funcctx)
```

для того, чтобы провести очистку и завершить SRF.

Контекст памяти, в котором вызывается SRF, временный, он будет очищаться между вызовами. Это значит, что вам не нужно вызывать `pfree` для всех блоков памяти, которые вы получили через `palloc`; они всё равно будут освобождены. Однако если вы хотите выделить структуры данных, сохраняющиеся между вызовами, вам нужно разместить их где-то в другом месте. Для размещения данных, которые не должны уничтожаться, пока SRF не закончит работу, подходит контекст памяти, на который указывает `multi_call_memory_ctx`. В большинстве случаев это означает, что вы должны переключиться в контекст `multi_call_memory_ctx` в коде подготовки при первом вызове. Для сохранения указателя на такие долгоживущие структуры воспользуйтесь полем `funcctx->user_fctx`. (Память, которую вы получаете в контексте `multi_call_memory_ctx`, будет освобождена автоматически при завершении запроса, так что и её освобождать вручную нет необходимости.)

Предупреждение

Тогда как фактические аргументы такой функции не меняются от вызова к вызову, если вы распаковываете значения аргументов (что обычно прозрачно делают макросы `PG_GETARG_xxx`) во временном контексте, распакованные копии будут освобождаться при каждом вызове. Соответственно, если вы сохраните ссылки на такие значения в своём контексте `user_fctx`, вы должны либо скопировать эти значения в `multi_call_memory_ctx` после распаковки, либо распаковывать значения только в этом контексте.

Полный пример с псевдокодом будет выглядеть так:

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    другие необходимые объявления

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* Код подготовки при первом вызове: */
        некоторый код
        если возвращается составной тип
            получить TupleDesc и, возможно, AttInMetadata
        конец ветвления для составного типа
        некоторый код
        MemoryContextSwitchTo(oldcontext);
    }

    /* Код подготовки для каждого вызова: */
    некоторый код
    funcctx = SRF_PERCALL_SETUP();
    некоторый код

    /* Только так мы можем определить, не последний ли это вызов: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Здесь мы возвращаем ещё один результат: */
        некоторый код
```

```
        получение результирующих данных
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* Мы заканчиваем выдавать результаты, и это надо отразить. */
        /* (Воздержитесь от соблазна написать здесь код, освобождающий ресурсы.) */
        SRF_RETURN_DONE(funcctx);
    }
}
```

Полный пример простой SRF-функции, возвращающей составной тип, выглядит так:

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;

    /* stuff done only on the first call of the function */
    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext   oldcontext;

        /* create a function context for cross-call persistence */
        funcctx = SRF_FIRSTCALL_INIT();

        /* switch to memory context appropriate for multiple function calls */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* total number of tuples to be returned */
        funcctx->max_calls = PG_GETARG_INT32(0);

        /* Build a tuple descriptor for our result type */
        if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
            ereport(ERROR,
                    (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                     errmsg("function returning record called in context "
                            "that cannot accept type record")));

        /*
         * generate attribute metadata needed later to produce tuples from raw
         * C strings
         */
        attinmeta = TupleDescGetAttInMetadata(tupdesc);
        funcctx->attinmeta = attinmeta;

        MemoryContextSwitchTo(oldcontext);
    }

    /* stuff done on every call of the function */
    funcctx = SRF_PERCALL_SETUP();

    call_cntr = funcctx->call_cntr;
```

```
    max_calls = funcctx->max_calls;
    attinmeta = funcctx->attinmeta;

    if (call_cntr < max_calls)    /* do when there is more left to send */
    {
        char       **values;
        HeapTuple    tuple;
        Datum        result;

        /*
         * Prepare a values array for building the returned tuple.
         * This should be an array of C strings which will
         * be processed later by the type input functions.
         */
        values = (char **) palloc(3 * sizeof(char *));
        values[0] = (char *) palloc(16 * sizeof(char));
        values[1] = (char *) palloc(16 * sizeof(char));
        values[2] = (char *) palloc(16 * sizeof(char));

        snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
        snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
        snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

        /* build a tuple */
        tuple = BuildTupleFromCStrings(attinmeta, values);

        /* make the tuple into a datum */
        result = HeapTupleGetDatum(tuple);

        /* clean up (this is not really necessary) */
        pfree(values[0]);
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* do when there is no more left */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

В SQL её можно объявить следующим образом:

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'имя_файла', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Также её можно объявить с параметрами OUT:

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'имя_файла', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Заметьте, что при таком подходе выходным типом функции формально является анонимный тип `record`.

36.10.9. Полиморфные типы аргументов и результата

Функции на языке C могут быть объявлены как принимающие и возвращающие полиморфные типы, которые описаны в Подразделе 36.2.5. Когда типы аргументов или результата определены как полиморфные, автор функции не может заранее знать, с какими типами данных она будет вызываться и какой возвращать. Чтобы функция на C в стиле версии 1 могла определить фактические типы данных своих аргументов и тип, который она должна вернуть, в `fmgr.h` предлагаются две функции. Они называются `get_fn_expr_rettype(FmgrInfo *flinfo)` и `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)` и возвращают соответственно OID типа результата и аргумента, либо `InvalidOid`, если информация о типе отсутствует. Структуру `flinfo` обычно можно получить по ссылке `fcinfo->flinfo`. Номер аргумента `argnum` задаётся, начиная с нуля. В качестве альтернативы `get_fn_expr_rettype` также можно использовать функции `get_call_result_type`. Кроме того, есть функция `get_fn_expr_variadic`, позволяющая определить, были ли переменные аргументы объединены в массив. Это полезно в основном для функций `VARIADIC "any"`, так как такое объединение всегда имеет место для функций с переменными аргументами, принимающих обычные типы.

Например, предположим, что нам нужно написать функцию, принимающую один элемент любого типа и возвращающую одномерный массив этого типа:

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* получить переданный элемент, учитывая, что это может быть NULL */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* мы имеем дело с одной размерностью */
    ndims = 1;
    /* и одним элементом */
    dims[0] = 1;
    /* с нижней границей, равной 1 */
    lbs[0] = 1;

    /* получить требуемую информацию о типе элемента */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* теперь создать массив */
```

```
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

Следующая команда объявляет функцию `make_array` в SQL:

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'КАТАЛОГ/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

Существует один вариант полиморфизма, которым могут пользоваться только функции на языке C: их можно объявить с параметрами типа `"any"`. (Заметьте, что имя этого типа нужно заключать в двойные кавычки, так как это также зарезервированное слово в SQL.) Он работает так же, как `anyelement`, за исключением того, что он не требует, чтобы аргументы `"any"` имели одинаковый тип, и не помогает определить тип результата функции. Функцию на языке C можно также объявить с последним параметром `VARIADIC "any"`. Ему будут соответствовать один или более фактических аргументов любого типа (не обязательно одинакового). Эти аргументы *не* будут собираться в массив, как это происходит с обычными функциями с переменными аргументами; они просто будут переданы функции по отдельности. Если применяется этот вариант, то чтобы определить число фактических аргументов и их типы, нужно использовать макрос `PG_NARGS()` и функции, описанные выше. Пользователи такой функции также могут пожелать использовать ключевое слово `VARIADIC` в вызове функции, ожидая, что функция обработает элементы массива как отдельные аргументы. При необходимости соответствующее поведение должна реализовывать сама функция, определив с помощью `get_fn_expr_variadic`, был ли фактический аргумент передан с указанием `VARIADIC`.

36.10.10. Разделяемая память

36.10.10.1. Запрос разделяемой памяти при запуске сервера

Модули расширений могут резервировать разделяемую память при запуске сервера. Для этого нужно указать разделяемую библиотеку модуля в shared_preload_libraries, чтобы она предварительно загружалась на этапе запуска сервера. Библиотека должна также зарегистрировать в своей функции `_PG_init` обработчик `shmem_request_hook`, который сможет зарезервировать разделяемую память, если из него вызвать:

```
void RequestAddinShmemSpace(Size size)
```

Каждый обслуживающий процесс должен получить указатель на зарезервированную разделяемую память, для этого нужно вызвать:

```
void *ShmemInitStruct(const char *name, Size size, bool *foundPtr)
```

Если указатель `foundPtr` установлен в `false`, вызывающему следует начать инициализацию содержимого зарезервированной разделяемой памяти. Если `foundPtr` установлен в `true`, это означает, что разделяемая память уже инициализирована другим обслуживающим процессом и инициализация от вызывающего не требуется.

Во избежание возможных условий гонки каждый обслуживающий процесс должен вызывать `AddinShmemInitLock` при инициализации разделяемой памяти, как показано здесь:

```
static mystruct *ptr = NULL;
bool        found;

LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
ptr = ShmemInitStruct("my struct name", size, &found);
if (!found)
{
    ... инициализировать содержимое разделяемой памяти ...
    ptr->locks = GetNamedLWLockTranche("my tranche name");
```

```
}
LWLockRelease(AddinShmemInitLock);
```

В `shmem_startup_hook` удобно обрабатывать код инициализации, однако необязательно размещать в нём весь подобный код. Каждый обслуживающий процесс вызывает зарегистрированный обработчик `shmem_startup_hook` вскоре после подключения к разделяемой памяти. Обратите внимание, что модули расширений также должны получить `AddinShmemInitLock` в рамках этого обработчика, как показано в примере выше.

36.10.10.2. Запрос разделяемой памяти после запуска сервера

Существует более гибкий способ резервирования разделяемой памяти, который можно применить после запуска сервера и вне обработчика `shmem_request_hook`. Для этого каждый использующий разделяемую память обслуживающий процесс должен получить указатель на неё, вызвав:

```
void *GetNamedDSMSegment(const char *name, size_t size,
                         void (*init_callback) (void *ptr),
                         bool *found)
```

Если сегмент динамической разделяемой памяти с указанным именем ещё не существует, эта функция выделит и инициализирует его при помощи функции-обработчика `init_callback`. Если сегмент уже выделен и инициализирован другим обслуживающим процессом, функция назначает существующий сегмент динамической разделяемой памяти текущему обслуживающему процессу.

В отличие от резервирования разделяемой памяти при запуске сервера, при резервировании при помощи функции `GetNamedDSMSegment` нет необходимости получать `AddinShmemInitLock` или выполнять другие действия, чтобы избежать условий гонки. Эта функция гарантирует, что только один обслуживающий процесс выделяет и инициализирует сегмент памяти и что другие обслуживающие процессы получают указатель на полностью выделенный и инициализированный сегмент.

36.10.11. Лёгкие блокировки

36.10.11.1. Запрос лёгких блокировок при запуске

Модули расширений могут резервировать лёгкие блокировки при запуске сервера. Как и в случае с резервированием разделяемой памяти при запуске, нужно указать библиотеку модуля в shared_preload_libraries, чтобы она предварительно загружалась на этапе запуска сервера, и библиотека должна зарегистрировать в своей функции `_PG_init` обработчик `shmem_request_hook`. Чтобы зарезервировать лёгкие блокировки, из этого обработчика нужно вызвать:

```
void RequestNamedLWLockTranche(const char *tranche_name, int num_lwlocks)
```

. В результате будет сформирован массив из `num_lwlocks` лёгких блокировок под именем `tranche_name`. Указатель на этот массив можно получить, вызвав:

```
LWLockPadded *GetNamedLWLockTranche(const char *tranche_name)
```

36.10.11.2. Запрос лёгких блокировок после запуска сервера

Существует более гибкий способ получения лёгких блокировок, который можно применить после запуска сервера и вне обработчика `shmem_request_hook`. Для этого сначала нужно выделить `tranche_id`, вызвав:

```
int LWLockNewTrancheId(void)
```

Затем инициализировать каждую лёгкую блокировку, передав новый `tranche_id` в качестве аргумента:

```
void LWLockInitialize(LWLock *lock, int tranche_id)
```

Как в случае с разделяемой памятью, каждый обслуживающий процесс должен удостовериться, что только один процесс выделяет новый `tranche_id` и инициализирует каждую новую лёгкую блокировку. Это можно сделать, например, вызывая приведённые выше функции только в коде инициализации разделяемой памяти с исключительной блокировкой `AddinShmemInitLock`. При использовании функции `GetNamedDSMSegment`, чтобы избежать условий гонки, достаточно вызвать эти функции в функции-обработчике `init_callback`.

Наконец, каждый обслуживающий процесс, который использует `tranche_id`, должен связать его с `tranche_name`, вызвав:

```
void LWLockRegisterTranche(int tranche_id, const char *tranche_name)
```

36.10.12. Пользовательские события ожидания

Модули расширений могут задавать пользовательские события ожидания в типе событий ожидания `Extension` с помощью следующего вызова:

```
uint32 WaitEventExtensionNew(const char *wait_event_name)
```

Событие ожидания связывается с задаваемой пользователем строкой.

Пользовательские события ожидания можно просматривать в представлении `pg_stat_activity`:

```
=# SELECT wait_event_type, wait_event FROM pg_stat_activity
     WHERE backend_type ~ 'worker_spi';
 wait_event_type |  wait_event
-----------------+---------------
 Extension       | WorkerSpiMain
(1 row)
```

36.10.13. Точки внедрения

Точка внедрения кода с заданным именем `name` объявляется при помощи следующего макроса:

```
INJECTION_POINT(name);
```

Несколько таких точек уже объявлены в стратегически важных местах кода сервера. После добавления новой точки внедрения необходимо выполнить компиляцию, чтобы эта точка стала доступна в двоичных файлах. В коде модулей расширений, написанных на языке C, можно объявлять точки внедрения кода при помощи этого же макроса. Имена точек должны содержать символы в нижнем регистре, а слова в именах должны быть разделены минусами.

Модули расширений могут подключить обработчики к уже объявленной точке внедрения кода, вызвав:

```
extern void InjectionPointAttach(const char *name,
                                 const char *library,
                                 const char *function,
                                 const void *private_data,
                                 int private_data_size);
```

`name` — имя точки внедрения кода, при достижении которой во время выполнения кода вызывается функция `function`, загруженная из библиотеки `library`. `private_data` — область данных с размером `private_data_size`, которая передаётся в качестве аргумента обработчику при выполнении.

Ниже представлен пример обработчика для `InjectionPointCallback`:

```
static void
custom_injection_callback(const char *name, const void *private_data)
{
    uint32 wait_event_info = WaitEventInjectionPointNew(name);

    pgstat_report_wait_start(wait_event_info);
    elog(NOTICE, "%s: executed custom callback", name);
    pgstat_report_wait_end();
}
```

Этот обработчик добавляет сообщение с уровнем важности `NOTICE` в журнал ошибок сервера, однако обработчики могут выполнять и более сложную логику.

При необходимости точку внедрения можно отключить, вызвав:

```
extern bool InjectionPointDetach(const char *name);
```

При успешном отключении выводится `true`, в противном случае — `false`.

Присоединённый к точке обработчик доступен для всех обслуживающих процессов, в том числе запущенных после вызова `InjectionPointAttach`. Обработчик остаётся присоединённым на время работы сервера и отсоединяется после завершения его работы или при отсоединении точки при помощи `InjectionPointDetach`.

Точки внедрения доступны только в том случае, если Postgres Pro скомпилирована с поддержкой этой возможности.

36.10.14. Использование C++ для расширяемости

Хотя код сервера Postgres Pro написан на C, расширения для него можно писать и на C++, если соблюдать эти правила:

- Все функции, к которым будет обращаться сервер, должны предоставлять ему интерфейс C; эти функции на C затем могут вызывать функции на языке C++. В частности, для функций, доступных серверу, необходимо указать `extern C`. Это также необходимо для всех функций, указатели на которые передаются между кодом сервера и подключаемым кодом на C++.
- Освобождайте память, применяя для этого подходящий метод. Например, память сервера в основном выделяется функцией `palloc()`, так что освобождать её нужно, вызывая `pfree()`. Попытка использовать в таких случаях принятую в C++ операцию `delete` приведёт к ошибке.
- Не допускайте распространения исключений в код C (добавляйте блок, перехватывающий все исключения, на верхнем уровне функций `extern C`). Это необходимо, даже если код на C++ не генерирует исключения явно, потому что исключения могут возникать, например, и при нехватке памяти. Все исключения должны перехватываться, и в интерфейс C должны передаваться соответствующие ошибки. Если возможно, скомпилируйте код C++ с указанием `-fno-exceptions`, чтобы полностью отключить исключения; в таких случаях вы должны будете выявлять исключительные ситуации в коде C++, например, проверять на NULL адрес, возвращённый `new()`.
- Вызывая серверные функции из кода C++, убедитесь, что в стеке вызова C++ содержатся только простые структуры данных. Это необходимо, потому что в случае ошибки сервера выполняется функция `longjmp()`, а она не отматывает стек вызовов C++ должным образом для объектов, отличных от простых структур.

Резюмируя, лучше всего поместить код C++ за ограду из функций `extern C`, которые будут доступны серверу и смогут защитить от исключений, а также потери стека вызовов и утечки памяти.

36.11. Информация для оптимизации функций

Сама по себе функция для СУБД является просто «чёрным ящиком», о поведении которого известно очень мало. Это означает, что запросы, вызывающие функции, могут выполняться гораздо менее эффективно, чем могли бы в теории. Поэтому имеется возможность сообщить планировщику дополнительные сведения о функции, которые помогут ему оптимизировать вызовы функций.

Некоторые основные факты передаются декларативным образом в команде `CREATE FUNCTION`; в их числе один из самых значимых — характеристика изменчивости (`IMMUTABLE`, `STABLE` или `VOLATILE`). Создавая любую функцию, очень важно правильно определить эту характеристику. Также может определяться характеристика распараллеливания (`PARALLEL UNSAFE`, `PARALLEL RESTRICTED` или `PARALLEL SAFE`), если вы рассчитываете, что эта функция будет использоваться в распараллеливаемых запросах. Кроме того, для функции может задаваться примерная стоимость выполнения и/или оценка количества строк, выдаваемого функцией, возвращающей множество. Однако декларативный способ описания двух последних фактов позволяет задать только некоторое постоянное значение, а это полезно далеко не всегда.

Но имеется также возможность связать *вспомогательную функцию для планировщика* с вызываемой из SQL функцией (она будет *целевой функцией* для первой), и передать через неё ту информацию о целевой функции, которая слишком сложна для представления в декларативном виде. Вспомогательные функции для планировщика должным быть написаны на C (хотя язык целевых функций может быть любым), что переводит их в категорию расширенных возможностей, и разрабатывать их будут относительно немногие пользователи.

Вспомогательная функция должна иметь в SQL такую сигнатуру:

```
supportfn(internal) returns internal
```

Она связывается с целевой функцией с помощью указания `SUPPORT` в команде, создающей целевую функцию.

Здесь даётся общее представление о том, что могут делать вспомогательные функции для планировщика. Множество запросов к вспомогательным функциям расширяемое, так что в будущих версиях у них могут появиться и другие возможности.

Некоторые вызовы функций можно упростить во время планирования, в зависимости от особенностей функции. Например, вызов `int4mul(n, 1)` можно свести просто к `n`. Преобразование такого рода может выполняться вспомогательной функцией, если она обрабатывает запросы типа `SupportRequestSimplify`. Эта вспомогательная функция будет вызываться для каждого экземпляра вызова целевой функции, найденного в дереве разобранного запроса. Если она обнаруживает, что этот конкретный вызов можно упростить и привести к другому виду, она может построить и возвратить другое дерево с изменённым выражением. Это будет автоматически работать и для операторов, основанных на функциях, — в данном примере `n * 1` будет также упрощено до `n`. (Но заметьте, что это просто иллюстрация; конкретно эту оптимизацию стандартный Postgres Pro не производит). При этом не гарантируется, что Postgres Pro никогда не вызовет целевую функцию в случаях, которые может упростить вспомогательная функция. С учётом этого важно обеспечить строгую идентичность упрощённого выражения фактическому выполнению целевой функции.

Для целевой функции, возвращающей значение `boolean`, часто бывает полезно оценить, какой процент строк будет выбран предложением `WHERE`, в котором вызывается эта функция. Эту оценку позволяет получить вспомогательная функция, обрабатывающая запросы типа `SupportRequestSelectivity`.

Если алгоритм работы целевой функции значительно меняется в зависимости от её аргументов, для неё может иметь смысл вычислять переменную оценку стоимости. Это можно сделать, реализовав вспомогательную функцию, которая будет обрабатывать запросы типа `SupportRequestCost`.

Для целевой функции, возвращающей множество, часто полезно иметь переменную оценку числа выдаваемых строк. Реализовать это позволяет вспомогательная функция, обрабатывающая запросы типа `SupportRequestRows`.

Для целевой функции, возвращающей значение `boolean`, может существовать возможность преобразовать вызов функции в условии `WHERE` в предложение(я) с индексируемыми операторами. Преобразованные предложения должны быть в точности идентичны условию функции либо могут быть несколько менее строгими (то есть они могут принимать некоторые значения, не удовлетворяющие условию с функцией). В последнем случае условие с индексом считается *неточным*; оно может использоваться для поиска по индексу, но для каждой строки, полученной при таком поиске, должна вызываться функция, чтобы точно определить, удовлетворяет ли строка условию `WHERE`. Для создания таких условий вспомогательная функция должна обрабатывать запросы типа `SupportRequestIndexCondition`.

36.12. Пользовательские агрегатные функции

Агрегатные функции в Postgres Pro определяются в терминах *значений состояния* и *функций перехода состояния*. То есть агрегатная функция работает со значением состояния, которое меняется при обработке каждой последующей строки. Чтобы определить агрегатную функцию, нужно выбрать тип данных для значения состояния, начальное значение состояния и функцию перехода состояния. Функция перехода состояния принимает предыдущее значение состояния и входное агрегируемое значение для текущей строки и возвращает новое значение состояния. Также можно указать *функцию завершения*, на случай, если ожидаемый результат агрегатной функции отличается от данных, которые сохраняются в изменяющемся значении состояния. Функция завершения принимает конечное значение состояния и возвращает то, что она хочет вернуть в виде результата агрегирования. В принципе, функции перехода и завершения представляют собой просто обычные функции, которые также могут применяться вне контекста агрегирования. (На практике для большей производительности часто создаются специализированные функции перехода, которые работают, только когда вызываются при агрегировании.)

Таким образом, помимо типов данных аргументов и результата, с которыми имеет дело пользователь агрегатной функции, есть также тип данных внутреннего состояния, который может отличаться от этих типов.

Если мы определяем агрегат, не использующий функцию завершения, наш агрегат будет вычислять бегущее значение функции по столбцам каждой строки. Примером такой агрегатной функции является `sum`. Вычисление `sum` начинается с нуля, а затем к накапливаемой сумме всегда прибавляется значение из текущей строки. Например, если мы хотим сделать агрегатную функцию `sum` для комплексных чисел, нам потребуется только функция сложения для такого типа данных. Такая агрегатная функция может быть определена так:

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

Использовать её можно будет так:

```
SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

(Заметьте, что мы задействуем перегрузку функций: в системе есть несколько агрегатных функций с именем `sum`, но Postgres Pro может определить, какая именно из них применима к столбцу типа `complex`.)

Определённая выше функция `sum` вернёт ноль (начальное значение состояния), если в наборе данных не окажется значений, отличных от NULL. У нас может возникнуть желание вернуть NULL в этом случае — стандарт SQL требует, чтобы `sum` работала так. Мы можем добиться этого, просто опустив фразу `initcond`, так что начальным значением состояния будет NULL. Обычно это будет означать, что в `sfunc` придётся проверять входное значение состояния на NULL. Но для `sum` и некоторых других простых агрегатных функций, как `max` и `min`, достаточно вставить в переменную состояния первое входное значение не NULL, а затем начать применять функцию перехода со следующего значения не NULL. Postgres Pro сделает это автоматически, если начальное значение состояние равно NULL и функция перехода помечена как «strict» (то есть не должна вызываться для аргументов NULL).

Ещё одна особенность поведения по умолчанию «строгой» функции перехода — предыдущее значение состояния остаётся без изменений, когда встречается значение NULL. Другими словами, значения NULL игнорируются. Если вам нужно другое поведение для входных значений NULL, не объявляйте свою функцию перехода строгой (strict); вместо этого, проверьте в ней поступающие значения на NULL и обработайте их, как требуется.

Функция `avg` (вычисляющая среднее арифметическое) представляет собой более сложный пример агрегатной функции. Ей необходимы два компонента текущего состояния: сумма входных значений и их количество. Окончательный результат получается как частное этих величин. При реализации этой функции для значения состояния обычно используется массив. Например, встроенная реализация `avg(float8)` выглядит так:

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

Примечание

Функция `float8_accum` принимает массив из трёх, а не двух элементов, так как в дополнение к количеству и сумме значений она подсчитывает ещё сумму их квадратов. Это сделано для того, чтобы её можно было применять для `avg` и для некоторых других агрегатных функций.

Вызовы агрегатных функций SQL допускают указания `DISTINCT` и `ORDER BY`, которые определяют, какие строки и в каком порядке будут поступать в функцию перехода агрегата. Это реализовано на заднем плане и непосредственно не затрагивает функции, поддерживающие работу агрегата.

За дополнительными подробностями обратитесь к описанию CREATE AGGREGATE.

36.12.1. Режим движущегося агрегата

Агрегатные функции могут дополнительно поддерживать *режим движущегося агрегата*, который позволяет значительно быстрее выполнять агрегатные функции в окнах со сдвигающимся началом рамки. (За информацией об использовании агрегатных функций в качестве оконных обратитесь к Разделу 3.5 и Подразделу 4.2.8.) Основная идея состоит в том, что помимо добавления обычной функции перехода «вперёд», для агрегатной функции задаётся *функция обратного перехода*, которая позволяет убирать строки из накапливаемого значения состояния, когда они покидают рамку окна. Например, для `sum` в качестве функции прямого перехода выполняется сложение, а в качестве функции обратного перехода выполняется вычитание. Без функции обратного перехода механизм оконных функций вынужден вычислять агрегат заново при каждом перемещении начала рамки, в результате чего время обработки оказывается пропорциональным количеству входных строк, помноженному на средний размер рамки. С функцией обратного перехода это время пропорционально только количеству входных строк.

Функции обратного перехода передаётся текущее значение состояния и агрегируемое входное значение(я) для строки, ранее учтённой в текущем состоянии. Она должна восстановить то значение состояния, которое было бы получено, если бы эта строка не агрегировалась, но агрегировались все последующие. Иногда для этого нужно, чтобы функция обратного перехода сохраняла больше информации о состоянии, чем это требуется для простого режима агрегирования. Таким образом, для режима движущегося агрегата используется реализация, отличная от простого режима: для него определяется отдельный тип данных, отдельная функция прямого перехода и отдельная функция завершения, при необходимости. Они могут совпадать с типом данных и аналогичными функциями обычного режима, если в дополнительном состоянии необходимости нет.

В качестве примера мы можем доработать показанную выше агрегатную функцию `sum`, чтобы она поддерживала режим движущегося агрегата так:

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)',
    msfunc = complex_add,
    minvfunc = complex_sub,
    mstype = complex,
```

```
    minitcond = '(0,0)'
);
```

Параметры, имена которых начинаются с `m`, определяют реализацию для движущегося агрегата. За исключением функции обратного перехода, `minvfunc`, они соответствуют параметрам обычного агрегата без `m`.

Функции прямого перехода в режиме движущегося агрегата не разрешено возвращать NULL в качестве нового значения состояния. Если функция обратного перехода возвращает NULL, это воспринимается как признак того, что она не может восстановить предыдущее состояние для полученных данных, и значит, агрегатное вычисление нужно производить заново с текущей позиции начала рамки. Это соглашение позволяет применять режим движущегося агрегата и в ситуациях, когда прокручивать назад значение состояния непрактично. Функция обратного перехода может «спасовать» в таких случаях, но включаться в работу, насколько это возможно в большинстве случаев. Например, агрегатная функция, работающая с числами с плавающей точкой, может спасовать, когда от неё потребуется убрать значение `NaN` (не число, not a number) из текущего значения состояния.

Разрабатывая функции, реализующие режим движущегося агрегата, важно, чтобы функция обратного перехода могла восстановить в точности требуемое значение состояния. Иначе в результатах могут проявляться различия в зависимости от того, использовался ли режим движущегося агрегата. Например, на первый взгляд может показаться, что легко добавить функцию обратного перехода для сложения, но заявленное требование не будет выполняться для `sum` с типом `float4` или `float8`. Наивное объявление `sum(float8)` может быть таким:

```
CREATE AGGREGATE unsafe_sum (float8)
(
    stype = float8,
    sfunc = float8pl,
    mstype = float8,
    msfunc = float8pl,
    minvfunc = float8mi
);
```

Однако такой агрегат может выдавать результаты, радикально отличающиеся от тех, что он выдавал бы без функции обратного перехода. Например, рассмотрите запрос

```
SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
FROM (VALUES (1, 1.0e20::float8),
             (2, 1.0::float8)) AS v (n,x);
```

Он возвращает `0` в качестве второго результата, а не ожидаемое значение `1`. Это связано с ограниченной точностью значений с плавающей точкой: при добавлении `1` к `1e20` снова получается `1e20`, а при вычитании `1e20` из результата получается `0`, а не `1`. Заметьте, что это принципиальное ограничение арифметики чисел с плавающей точкой, а не недостаток Postgres Pro.

36.12.2. Агрегатные функции с полиморфными и переменными аргументами

Агрегатная функция может использовать полиморфные функции перехода состояния или функции завершения, так что эти функции могут применяться для реализации нескольких агрегатов. За объяснением полиморфных функций обратитесь к Подразделу 36.2.5. Более того, сама агрегатная функция может описываться с полиморфными типами входных данных и состояния, так что одно определение агрегатной функции может служить для использования с разными типами данных. Пример полиморфного агрегата:

```
CREATE AGGREGATE array_accum (anycompatible)
(
    sfunc = array_append,
    stype = anycompatiblearray,
```

```
    initcond = '{}'
);
```

Здесь фактическим типом состояния для любого конкретного агрегатного вызова будет массив, элементы которого будут иметь тип входных данных. Действие данного агрегата заключается в накоплении всех входных значений в массиве этого типа. (К вашему сведению: встроенная агрегатная функция `array_agg` обеспечивает подобную функциональность, но работает быстрее, чем могла бы функция с приведённым определением.)

Так будут выглядеть результаты с аргументами двух различных типов:

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

Обычно агрегатная функция с полиморфным типом результата имеет и полиморфный тип состояния, как в предыдущем примере. Это необходимо, так как иначе нельзя будет объявить функцию завершения: она должна будет иметь полиморфный тип результата, но не будет иметь полиморфного аргумента, что команда `CREATE FUNCTION` не примет на основании того, что тип результата нельзя будет определить при вызове. Но иметь полиморфный тип состояния не всегда удобно. Чаще всего эта проблема возникает, когда функции реализации агрегата пишутся на C и тип состояния должен объявляться как `internal`, так как для него нет соответствующего типа на уровне SQL. Чтобы решить эту проблему, можно объявить функцию завершения как принимающую дополнительные фиктивные аргументы, соответствующие входным аргументам агрегата. В этих фиктивных аргументах всегда передаются значения NULL, так как при вызове функции завершения какое-либо определённое значение отсутствует. Единственное их предназначение — позволить связать тип результата полиморфной функции завершения с типом входных данных агрегата. Например, определение встроенного агрегата `array_agg` выглядит так:

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn,
    finalfunc_extra
);
```

Здесь параметр `finalfunc_extra` указывает, что функция завершения помимо значения состояния получит дополнительные фиктивные аргументы, соответствующие входным аргументам агрегата.

Дополнительный аргумент `anynonarray` позволяет сделать объявление `array_agg_finalfn` допустимым.

Агрегатную функцию можно сделать принимающей переменное число аргументов, объявив её последний аргумент как массив `VARIADIC`, в том же ключе, как и обычную функцию; см. Подраздел 36.5.6. При этом у функций перехода агрегата их последний аргумент должен иметь тот же тип массива. Такие функции обычно также объявляются как `VARIADIC`, но строго это не требуется.

> **Примечание**
>
> Агрегатные функции с переменными аргументами легко допускают ошибочное использование в сочетании с указанием `ORDER BY` (см. Подраздел 4.2.7), так как анализатор запроса не может определить, было ли передано нужное количество фактических параметров в такой комбинации. Помните, что всё, находящееся справа от `ORDER BY`, является ключом сортировки, а не аргументом агрегатной функции. Например, в
>
> ```
> SELECT myaggregate(a ORDER BY a, b, c) FROM ...
> ```
>
> анализатор запроса увидит один агрегатный аргумент функции и три ключа сортировки. Однако пользователь мог подразумевать и следующее:
>
> ```
> SELECT myaggregate(a, b, c ORDER BY a) FROM ...
> ```
>
> Если функция `myaggregate` принимает переменные аргументы, оба эти вызова будут вполне допустимы.
>
> По этой же причине, стоит подумать дважды, прежде чем создавать агрегатные функции с одинаковыми именами, но разным числом обычных аргументов.

36.12.3. Сортирующие агрегатные функции

Описанные выше агрегатные функции были «обычными» агрегатами. Но Postgres Pro также поддерживает *сортирующие агрегатные функции*, которые имеют два отличия от обычных. Во-первых, в дополнение к обычным агрегируемым аргументам, вычисляемых для каждой входной строки, сортирующий агрегат может иметь «непосредственные» аргументы, которые должны вычисляться в операции агрегирования только один раз. Во-вторых, для обычных агрегируемых аргументов порядок их сортировки задаётся явно, а сортирующий агрегат обычно выполняет вычисления, зависящие от конкретного порядка строк, например, вычисляет ранг или процентиль, так что порядок сортировки критичен для каждого вызова. Например, встроенное определение функции `percentile_disc` равнозначно следующему:

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
    sfunc = ordered_set_transition,
    stype = internal,
    finalfunc = percentile_disc_final,
    finalfunc_extra
);
```

Этот агрегат принимает непосредственный аргумент `float8` (дробь процентиля) и агрегируемые данные, которые могут быть любого упорядочиваемого типа. Используя его, можно рассчитать средний семейный доход следующим образом:

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_disc
-----------------
```

```
 50489
```

В данном случае `0.5` — это непосредственный аргумент; если бы дробь процентиля менялась от строки к строке, это не имело бы смысла.

В отличие от случая с обычными агрегатами, сортировка входных строк для сортирующего агрегата *не* выполняется на заднем плане, а является задачей функций, реализующих агрегат. Типичная реализация такого агрегата заключается в сохранении ссылки на объект «tuplesort» в значении состояния агрегата, загрузке поступающих строк в этот объект, и собственно окончании сортировки и выдачи данных в функции завершения. При такой организации обработки функция завершения может выполнять специальные операции, в частности, вставлять дополнительные «гипотетические» строки в сортируемые данные. Тогда как обычные агрегаты часто реализуются функциями, написанными на PL/pgSQL или другом процедурном языке, сортирующие агрегатные функции, как правило, должны быть написаны на C, так как их значение состояния нельзя выразить каким-либо типом данных SQL. (Обратите внимание, что в приведённом выше примере значение состояния объявлено как имеющее тип `internal` — это типичный вариант.) И вследствие того, что сортировку выполняет функция завершения, нельзя возобновить добавление входных строк, продолжая вызывать функцию перехода. Это означает, что функция завершения не может иметь характеристику `READ_ONLY`; она должна объявляться командой `CREATE AGGREGATE` с характеристикой `READ_WRITE` или `SHAREABLE` (если она позволяет при последующих вызовах функции завершения использовать уже отсортированное состояние).

Функция перехода состояния для сортирующего агрегата получает значение текущего состояния плюс агрегируемые входные данные для каждой строки и возвращает изменённое значение состояния. Это определение распространяется и на обычные агрегаты, но заметьте, что непосредственные аргументы (если они есть) при этом не передаются. Функция завершения же получает последнее значение состояния и значения непосредственных аргументов (если они есть), а также (если присутствует указание `finalfunc_extra`) значения NULL, соответствующие агрегируемым данным. С обычными агрегатами указание `finalfunc_extra` действительно полезно, только если агрегат полиморфный; тогда дополнительные фиктивные аргументы необходимы, чтобы связать тип результата функции завершения с типом(ами) входных данных агрегата.

В настоящее время сортирующие агрегаты не могут использоваться в качестве оконных функций, поэтому от них поддержка режима движущегося агрегата не требуется.

36.12.4. Частичное агрегирование

Дополнительно агрегатная функция может поддерживать *частичное агрегирование*. Идея такого агрегирования в том, чтобы вызывать функции перехода состояния для различных подмножеств входных данных независимо, а затем комбинировать значения состояния, вычисленные по этим подмножествам, и получать тот же результат, что был бы получен при сканировании сразу всех входных данных. Этот режим может применяться для параллельного агрегирования, когда разные рабочие процессы сканируют различные части таблицы. При этом каждый рабочий процесс выдаёт частичное значение состояния, а в конце эти значения комбинируются вместе и получается окончательное значение состояния. (В будущем этот режим может также применяться, например для комбинированного агрегирования локальных и удалённых таблиц, но пока это не реализовано.)

Для поддержки частичного агрегирования в определении агрегатной функции должна задаваться *комбинирующая функция*, принимающая два значения типа состояния агрегата (представляющие результаты агрегирования по двум подмножествам входных строк) и выдающая новое значение типа состояния, представляющее то состояние, которое было бы получено при агрегировании совокупности этих подмножеств строк. При этом относительный порядок входных строк в этих двух множествах не оговаривается. Это значит, что для агрегатных функций, зависящих от порядка входных строк, обычно невозможно определить осмысленную комбинирующую функцию.

В качестве простого примера, частичное агрегирование могут поддерживать функции `MAX` и `MIN`, если задать в качестве комбинирующей соответственно функцию сравнения значений большее-из-двух или меньшее-из-двух, ту же, что они используют и как функцию перехода. Для `SUM` комбинирующей функцией будет просто функция сложения. (И это опять же функция перехода, если только значение состояния не выходит за рамки типа входных данных.)

Комбинирующая функция задействуется практически так же, как функция перехода, но принимает в качестве второго аргумента значение типа состояния, а не нижележащего входного типа. В частности, на неё распространяются те же правила строгости функции и передачи значений NULL. Также учтите, что если в определении агрегатной функции задаётся отличное от NULL значение `initcond`, оно будет задавать начальное состояние не только для каждого прохода частичного агрегирования, но также и начальное состояние для комбинирующей функции, которая будет вызываться для комбинирования каждого частичного результата в этом состоянии.

Если типом состояния агрегатной функции выбран `internal`, комбинирующая функция отвечает за то, чтобы её результат был помещён в контекст памяти, подходящий для значений агрегатного состояния. В частности это значит, что, получив в первом аргументе `NULL`, нельзя просто возвратить второй аргумент, так как это значение окажется в неверном контексте и не просуществует достаточное время.

Когда типом состояния агрегатной функции выбран `internal`, обычно в определении агрегатной функции также уместно задать *функцию сериализации* и *функцию десериализации*, которые позволяют копировать значение состояния из одного процесса в другой. Без этих функций параллельное агрегирование невозможно, а также вероятно не будут работать такие будущие приложения, как локальное/удалённое агрегирование.

Функция сериализации должна принимать один аргумент типа `internal` и возвращать результат типа `bytea`, представляющий значение состояния, упакованное в плоский набор байтов. Функция десериализации, напротив, обращает это преобразование. Она должна принимать два аргумента типов `bytea` и `internal` и возвращать результат типа `internal`. (Второй её аргумент не используется и всегда равен нулю, но он требуется из соображений типобезопасности.) Результат функции десериализации следует просто разместить в текущем контексте памяти, так как в отличие от результата комбинирующей функции он недолговечен.

Также стоит заметить, что для выполнения агрегатной функции в параллельном режиме она должна иметь характеристику `PARALLEL SAFE` (безопасная для распараллеливания). Характеристики допустимости распараллеливания её опорных функций значения не имеют.

36.12.5. Вспомогательные функции для агрегатов

Функция, написанная на C, может определить, была ли она вызвана как вспомогательная функция агрегирования, вызвав `AggCheckCallContext`, например, так:

```
if (AggCheckCallContext(fcinfo, NULL))
```

Смысл такой проверки в том, что в случае положительного её результата первым входным аргументом является временное значение состояния, которое можно безопасно модифицировать на месте, не создавая новую копию. Пример вы можете увидеть в функции `int8inc()`. (Хотя агрегатные функции перехода всегда могут изменить непосредственно переходное значение, агрегатные функции завершения должны избегать этого; если они это делают, такое поведение должно отмечаться при создании агрегата. За дополнительными подробностями обратитесь к CREATE AGGREGATE.)

Второй аргумент `AggCheckCallContext` можно использовать, чтобы получить контекст памяти, в котором содержатся значения агрегатного состояния. Это полезно для функций перехода, которые желают использовать «развёрнутые» объекты (см. Подраздел 36.13.1) в качестве значений состояния. При первом вызове такая функция перехода должна возвратить развёрнутый объект в контексте памяти, относящемся к контексту состояния агрегата, а затем продолжать возвращать тот же объект при последующих вызовах. Например, эту логику можно увидеть в функции `array_append()`. (Функция `array_append()` не используется в качестве перехода никаким встроенным агрегатом, но она написана так, чтобы работать эффективно в таком качестве в дополнительном агрегате.)

Ещё одна вспомогательная подпрограмма, предназначенная для агрегатных функций, написанных на C, называется `AggGetAggref`. Эта функция возвращает узел разбора `Aggref`, описывающий вызов агрегата. Это в основном полезно для сортирующих агрегатов, которые могут исследовать структуру узла `Aggref` и выяснить, какой порядок сортировки они должны реализовать.

36.13. Пользовательские типы

Как описывалось в Разделе 36.2, Postgres Pro может расширяться и поддерживать новые типы данных. В этом разделе описывается, как определить новые базовые типы, то есть типы данных, описанные ниже уровня языка SQL. Для создания нового базового типа необходимо реализовать функции, работающие с этим типом, на языке низкого уровня, обычно C.

Пользовательский тип должен всегда иметь функции ввода и вывода. Эти функции определяют, как тип будет выглядеть в строковом виде (при вводе и выводе для пользователя) и как этот тип размещается в памяти. Функция ввода принимает в качестве аргумента строку символов, заканчивающуюся нулём, и возвращает внутреннее представление типа (в памяти). Функция вывода принимает в качестве аргумента внутреннее представление типа и возвращает строку символов, заканчивающуюся нулём. Если мы хотим не просто сохранить тип, но делать с ним нечто большее, мы должны предоставить дополнительные функции, реализующие все операции, которые мы хотели бы иметь для этого типа.

Предположим, что нам нужен тип `complex`, представляющий комплексные числа. Естественным образом комплексное число можно представить в памяти в виде следующей структуры C:

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

Нам нужно будет передавать этот тип по ссылке, так как он слишком велик, чтобы уместиться в одном значении `Datum`.

В качестве внешнего строкового представления типа мы выберем строку вида `(x,y)`.

Функции ввода и вывода обычно несложно написать, особенно функцию вывода. Но определяя внешнее строковое представление типа, помните, что в конце концов вам придётся реализовать законченный и надёжный метод разбора этого представления в функции ввода. Например, так:

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for type %s: \"%s\"",
                        "complex", str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

Функция вывода может быть простой:

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
```

```
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

Вам нужно позаботиться о том, чтобы функции ввода и вывода были обратными друг к другу. В противном случае вы столкнётесь с проблемами, когда вам потребуется выгрузить данные в файл, а затем прочитать их. Это особенно распространённая проблема, когда дело касается чисел с плавающей точкой.

Дополнительно пользовательский тип может предоставлять функции для ввода и вывода в двоичном виде. Двоичный ввод-вывод обычно работает быстрее, но хуже портируется, чем текстовый. Как и с текстовым представлением, выбор, каким будет двоичное представление, остаётся за вами. Многие встроенные типы данных стараются обеспечить двоичное представление, независимое от машинной архитектуры. Для типа `complex` мы воспользуемся функциями двоичного ввода-вывода типа `float8`:

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Написав функции ввода-вывода и скомпилировав их в разделяемую библиотеку, мы можем определить тип `complex` в SQL. Сначала мы объявим его как тип-пустышку:

```
CREATE TYPE complex;
```

Это позволит нам ссылаться на этот тип, определяя для него функции ввода-вывода. Теперь мы определим функции ввода-вывода:

```
CREATE FUNCTION complex_in(cstring)
```

```
    RETURNS complex
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'имя_файла'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'имя_файла'
   LANGUAGE C IMMUTABLE STRICT;
```

Наконец, мы можем предоставить полное определение типа данных:

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

Когда определяется новый базовый тип, Postgres Pro автоматически обеспечивает поддержку массивов с элементами такого типа. Тип массива обычно получает имя по имени базового типа с добавленным спереди символом подчёркивания (_).

Когда тип данных определён, мы можем объявить дополнительные функции для выполнения полезных операций с этим типом. Затем поверх этих функций могут быть определены операторы, а если потребуется, и классы операторов, для поддержки индексации этого типа. Эти дополнительные уровни обсуждаются в следующих разделах.

Если внутреннее представление типа данных имеет переменную длину, оно должно соответствовать стандартной схеме данных переменной длины: первые четыре байта должно занимать поле `char[4]`, к которому никогда не следует обращаться напрямую (по обыкновению названное `vl_len_`). Чтобы сохранить в этом поле размер элемента (включая длину самого поля), вы должны использовать макрос `SET_VARSIZE()`, а чтобы получить его — макрос `VARSIZE()`. (Эти макросы нужны, потому что поле длины может кодироваться по-разному на разных платформах.)

За дополнительными подробностями обратитесь к команде CREATE TYPE.

36.13.1. Особенности TOAST

Если значения вашего типа данных могут быть разного размера (во внутренней форме), обычно для такого типа желательно реализовать поддержку TOAST (см. Раздел 63.2). Это следует делать, даже если значения слишком малы для сжатия или внешнего хранения, так как TOAST позволяет сэкономить пространство и с данными маленького размера, сокращая издержки в заголовке.

Для поддержки хранения TOAST функции на C, работающие с таким типом данных, должны позаботиться о распаковке поступивших им данных, используя макрос `PG_DETOAST_DATUM`. (Эту внутреннюю особенность обычно скрывает дополнительный, определяемый для типа макрос `GETARG_DATATYPE_P`.) Затем, выполняя команду `CREATE TYPE`, укажите в качестве внутренней длины `variable` и выберите подходящий вариант хранения (не `plain`).

Если выравнивание данных не имеет значения (либо только для некоторой функции, либо потому что для типа данных в любом случае применяется выравнивание по байтам), некоторых издержек, связанных с макросом `PG_DETOAST_DATUM`, можно избежать. Вместо него можно использовать `PG_DETOAST_DATUM_PACKED` (его обычно скрывает определяемый для типа макрос `GETARG_DATATYPE_PP`) и воспользоваться макросами `VARSIZE_ANY_EXHDR` и `VARDATA_ANY` для обращения к потенциально сжатым данным. Стоит ещё раз отметить, что данные, возвращаемые этими макросами, не выравниваются, даже если выравнивание задано в определении типа. Если выравнивание важно, вы должны задействовать обычный интерфейс `PG_DETOAST_DATUM`.

Примечание

В старом коде поле `vl_len_` часто объявлялось как `int32`, а не `char[4]`. Это ничем не чревато до той поры, пока в определении структуры имеются другие поля с выравниванием как минимум `int32`. Но с потенциально невыровненными данными такое определение структуры опасно; компилятор может воспринять его как право полагать, что данные выровнены, что может привести к аварийным сбоям в архитектурах, строгих к выравниванию.

Поддержка TOAST даёт также возможность иметь *развёрнутое* представление данных в памяти, работать с которым будет удобнее, чем с форматом хранения на диске. Обычный или «плоский» формат хранения varlena в конце концов представляет собой просто набор байт; он не может содержать указатели, так как эти байты могут быть скопированы в другие адреса. Для сложных типов данных работать с плоским форматом данных может быть довольно дорого, так что Postgres Pro даёт возможность «развернуть» плоский формат в представление, более подходящее для вычислений, и затем передавать эту структуру в памяти функциям, работающим с этим типом.

Для использования развёрнутого хранения тип данных должен определять развёрнутый формат по правилам, описанным в *src/include/utils/expandeddatum.h*, и предоставлять функции для «разворачивания» плоского значения в этот формат, а также для «заворачивания» этого формата опять в обычное представление varlena. Затем надо добиться, чтобы все функции на C могли принимать любое представление, возможно выполняя преобразование одного в другое непосредственно при получении. Для этого не требуется исправлять сразу все существующие функции для этого типа данных, так как имеющийся стандартный макрос `PG_DETOAST_DATUM` способен преобразовывать развёрнутые входные данные в обычный плоский формат. Таким образом, все существующие функции, работающие с плоским форматом varlena продолжат работать, хотя и не очень эффективно, с развёрнутыми входными данными; их необязательно переделывать, пока не потребуется оптимизировать производительность.

Функции на C, умеющие работать с развёрнутым представлением, обычно делятся на две категории: те, что могут работать с развёрнутым форматом, и те, что могут принимать и развёрнутые, и плоские данные varlena. Первые проще написать, но они могут быть менее эффективными в целом, так как преобразование плоского значения в развёрнутую форму для использования только одной функцией может стоить больше, чем сэкономится при обработке данных в развёрнутом формате. Когда нужно работать только с развёрнутым форматом, преобразование плоских значений в развёрнутую форму можно скрыть в макросе, извлекающем аргументы, чтобы функция была не сложнее, чем работающая с традиционными входными данными varlena. Чтобы принимать оба варианта входных значений, напишите функцию извлечения аргументов, которая будет распаковывать значения с сокращённым заголовком, а также внешние и сжатые, но не развёрнутые данные. Такую функцию можно определить как возвращающую указатель на объединение плоского формата varlena и развёрнутого формата. Какой формат получен фактически, вызывающий код может определить, вызвав макрос `VARATT_IS_EXPANDED_HEADER()`.

Инфраструктура TOAST позволяет не только отличить обычные значения varlena от развёрнутых значений, но и различить указатели «для чтения/записи» и «только для чтения» на развёрнутые значения. Функции на C, которым нужно читать развёрнутое значение, или которые будут менять его безопасным и невидимым извне образом, могут не обращать внимания на тип полученного указателя. Если же функции на C выдают изменённую версию входного значения, они могут изменять развёрнутые входные данные на месте, только когда получают указатель для чтения/записи, но не когда получен указатель только для чтения. В последнем случае они должны сначала скопировать значение и получить новое значение, допускающее изменение. Функция на C, создающая новое развёрнутое значение, должна всегда возвращать указатель на него для чтения/записи. Кроме того, функция, изменяющая развёрнутое значение непосредственно по указателю для чтения/записи должна позаботиться о том, чтобы это значение осталось в приемлемом состоянии, если она отработает не полностью.

36.14. Пользовательские операторы

Любой оператор представляет собой «синтаксический сахар» для вызова нижележащей функции, выполняющей реальную работу; поэтому прежде чем вы сможете создать оператор, необходимо создать нижележащую функцию. Однако оператор — *не исключительно* синтаксический сахар, так как он несёт и дополнительную информацию, помогающую планировщику запросов оптимизировать запросы с этим оператором. Рассмотрению этой дополнительной информации будет посвящён следующий раздел.

Postgres Pro поддерживает префиксные и инфиксные операторы. Операторы могут быть перегружены; то есть одно имя оператора могут иметь различные операторы с разным количеством и типами операндов. Когда выполняется запрос, система определяет, какой оператор вызвать, по количеству и типам предоставленных операндов.

В следующем примере создаётся оператор сложения двух комплексных чисел. Предполагается, что мы уже создали определение типа `complex` (см. Раздел 36.13). Сначала нам нужна функция, собственно выполняющая операцию, а затем мы сможем определить оператор:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'имя_файла', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    function = complex_add,
    commutator = +
);
```

Теперь мы можем выполнить такой запрос:

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

Мы продемонстрировали создание бинарного оператора. Чтобы создать префиксный оператор, просто опустите `leftarg`. Обязательными в `CREATE OPERATOR` являются только предложение `function` и объявления аргументов. Предложение `commutator`, добавленное в данном примере, представляет необязательную подсказку для оптимизатора запросов. Подробнее о `commutator` и других подсказках для оптимизатора рассказывается в следующем разделе.

36.15. Информация для оптимизации операторов

Определение оператора в Postgres Pro может включать различные дополнительные предложения, которые сообщают системе полезные сведения о поведении оператора. Старайтесь задавать эти предложения при возможности, так как они могут значительно ускорить выполнение запросов, использующих данный оператор. Но если вы задаёте их, убедитесь, что они корректны! Неправильное применение предложений оптимизации может привести к замедлению запросов, неочевидным ошибочным результатам и другим неприятностям. Если вы не уверены в правильности предложения оптимизации, лучше вовсе не использовать его; единственным последствием будет то, что запросы будут работать медленнее, чем могли бы.

В будущих версиях Postgres Pro могут быть добавлены и другие предложения. Здесь описываются те, что поддерживаются версией 17.6.1.

Также для функции, реализующей оператор, имеется возможность добавить вспомогательную функцию для планировщика, которая будет передавать другую информацию о поведении оператора. За дополнительными сведениями обратитесь к Разделу 36.11.

36.15.1. `COMMUTATOR`

Предложение `COMMUTATOR`, если представлено, задаёт оператор, коммутирующий для определяемого. Оператор A является коммутирующим для оператора B, если (x A y) равняется (y B x) для всех возможных значений x, y. Заметьте, что B также будет коммутирующим для A. Например, операторы `<` и `>` для конкретного типа данных обычно являются коммутирующими друг для друга, а оператор `+` — коммутирующий для себя. Но традиционный оператор `-` коммутирующего не имеет.

Тип левого операнда оператора должен совпадать с типом правого операнда коммутирующего для него оператора, и наоборот. Поэтому имя коммутирующего оператора — это всё, что Postgres Pro должен знать, чтобы найти коммутатор, и всё, что нужно указать в предложении `COMMUTATOR`.

Информация о коммутирующих операторах крайне важна для операторов, которые будут применяться в индексах и условиях соединения, так как, используя её, оптимизатор запросов может «переворачивать» такие выражения и получать формы, необходимые для разных типов планов. Например, рассмотрим запрос с предложением WHERE `tab1.x = tab2.y`, где `tab1.x` и `tab2.y` имеют пользовательский тип, и предположим, что у нас есть индекс по столбцу `tab2.y`. Оптимизатор сможет задействовать сканирование по индексу, только если ему удастся перевернуть выражение `tab2.y = tab1.x`, так как механизм сканирования по индексу ожидает, что индексируемый столбец находится слева от оператора. Postgres Pro сам по себе *не* будет полагать, что такое преобразование возможно — это должен определить создатель оператора `=`, добавив информацию о коммутирующем операторе.

36.15.2. `NEGATOR`

Предложение `NEGATOR`, если присутствует, задаёт оператор, обратный к определяемому. Оператор A является обратным к оператору B, если они оба возвращают логический результат и (x A y) равняется NOT (x B y) для всех возможных x, y. Заметьте, что B так же является обратным к A. Например, операторы `<` и `>=` составляют пару обратных друг к другу для большинства типов данных. Никакой оператор не может быть обратным к себе же.

В отличие от коммутирующих операторов, два унарных оператора вполне могут быть обратными к друг другу; это будет означать, что (A x) равняется NOT (B x) для всех x.

У оператора, обратного данному, типы левого и/или правого операнда должны соответствовать типам данного оператора, так же как и с предложением `COMMUTATOR`; отличие только в том, что имя оператора задаётся в предложении `NEGATOR`.

Указание обратного оператора очень полезно для оптимизатора запросов, так как это позволяет упростить выражение вида `NOT (x = y)` до `x <> y`. Такие выражения не так редки, как может показаться, так как операции `NOT` могут добавляться автоматически в результате реорганизаций выражений.

36.15.3. `RESTRICT`

Предложение `RESTRICT`, если представлено, определяет функцию оценки избирательности ограничения для оператора. (Заметьте, что в нём задаётся имя функции, а не оператора.) Предложения `RESTRICT` имеют смысл только для бинарных операторов, возвращающих `boolean`. Идея оценки избирательности ограничения заключается в том, чтобы определить, какой процент строк в таблице будет удовлетворять условию `WHERE` вида:

```
column OP constant
```

для текущего оператора и определённого значения константы. Это помогает оптимизатору примерно определить, сколько строк будет исключено предложениями `WHERE` такого вида. (ВЫ спросите, а что если константа находится слева? Ну, собственно для таких случаев и задаётся `COMMUTATOR`...)

Рамки данной главы не позволяют описать разработку новых функций оценки избирательности ограничения, но обычно можно использовать один из стандартных системных оценщиков для большинства дополнительных операторов. Стандартные оценщики ограничений следующие:

`eqsel` для =
`neqsel` для <>
`scalarltsel` для <
`scalarlesel` для <=
`scalargtsel` для >
`scalargesel` для >=

Часто вы можете обойтись функциями `eqsel` и `neqsel` для операторов с очень высокой или низкой избирательностью, даже если это не операторы собственно равенства или неравенства. Например, геометрические операторы приблизительного равенства используют `eqsel` в предположении, что соответствующие (равные) элементы будут составлять только небольшой процент от всех записей таблицы.

Функции `scalarltsel`, `scalarlesel`, `scalargtsel` и `scalargesel` можно использовать для сравнений с типами данных, которые могут быть каким-либо осмысленным образом преобразованы в числовые скалярные значения для сравнения диапазонов.

Есть ещё одна полезная функция оценки избирательности, `matchingsel`, которая будет работать практически с любым бинарным оператором, если для его входных типов данных собирается статистика MCV и/или строится гистограмма. По умолчанию эта оценка в два раза больше той, что выдаёт `eqsel`, таким образом, данная функция наиболее полезна для операторов сравнения, более избирательных, чем оператор равенства. (Также можно вызвать нижележащую функцию `generic_restriction_selectivity`, передав ей другую оценку по умолчанию.)

36.15.4. JOIN

Предложение `JOIN`, если представлено, определяет функцию оценки избирательности соединения для оператора. (Заметьте, что в нём задаётся имя функции, а не оператора.) Предложения `JOIN` имеют смысл только для бинарных операторов, возвращающих `boolean`. Идея оценки избирательности соединения заключается в том, чтобы угадать, какой процент строк в паре таблиц будет удовлетворять условию `WHERE` следующего вида:

```
table1.column1 OP table2.column2
```

для текущего оператора. Как и `RESTRICT`, это предложение очень помогает оптимизатору, позволяя ему выяснить, какой из возможных вариантов соединения скорее всего окажется выгоднее.

Как и ранее, в этой главе мы не будем пытаться рассказать, как написать функцию оценивания избирательности соединения, а просто отметим, что вы можете использовать один из подходящих стандартных оценщиков:

`eqjoinsel` для =
`neqjoinsel` для <>
`scalarltjoinsel` для <
`scalarlejoinsel` для <=
`scalargtjoinsel` для >
`scalargejoinsel` для >=
`matchingjoinsel` для типовых операторов сопоставления
`areajoinsel` для сравнений областей в плоскости
`positionjoinsel` для сравнения положений в плоскости
`contjoinsel` для проверки на включение в плоскости

36.15.5. `HASHES`

Предложение `HASHES`, если присутствует, говорит системе, что для соединений с применением этого оператора допустимо использовать метод соединения по хешу. `HASHES` имеет смысл только для бинарного оператора, который возвращает `boolean`, и на практике этот оператор должен выражать равенство значений некоторого типа данных или пары типов данных.

Соединение по хешу базируется на том предположении, что оператор соединения возвращает истину только для таких пар значений слева и справа, для которых получается одинаковый хеш. Если два значения оказываются в разных ячейках хеша, операция соединения никогда не будет сравнивать их, неявно подразумевая, что результат оператора соединения в этом случае должен быть ложным. Поэтому не имеет никакого смысла указывать `HASHES` для операторов, которые не представляют какую-либо форму равенства. В большинстве случаев практический смысл в поддержке хеширования есть только для операторов, принимающих один тип данных с обеих сторон. Однако иногда возможно разработать хеш-функции, совместимые сразу с несколькими типами данных; то есть, функции, которые будут выдавать одинаковые хеш-коды для «равных» значений, несмотря на то, что эти значения будут представлены по-разному. Например, довольно легко функции с такой особенностью реализуются для хеширования целых чисел различного размера.

Чтобы оператор соединения имел характеристику `HASHES`, он должен входить в семейство операторов индексирования по хешу. Это требование откладывается, когда оператор только создаётся, ведь нужное семейство операторов, разумеется, ещё не может существовать. Но при попытке использовать такой оператор для соединения по хешу, возникнет ошибка во время выполнения, если такого семейства не окажется. Системе необходимо знать семейство операторов, чтобы найти функции для хеширования типа(ов) входных данных оператора. Конечно, вы должны также определить подходящие функции хеширования, прежде чем сможете создать семейство операторов.

При подготовке функции хеширования обязательно позаботьтесь о том, чтобы она всегда выдавала нужный результат, вне зависимости от особенностей машинной архитектуры. Например, если ваш тип данных представлен в структуре, в которой есть незначащие дополняющие биты, нельзя просто передать всю структуру функции `hash_any`. (Это возможно, только если все ваши операторы и функции гарантированно очищают незначащие биты, что является рекомендуемой стратегией.) В качестве другого примера можно привести типы с плавающей точкой в стандарте IEEE, в которых отрицательный ноль и положительный ноль — различные значения (отличаются на уровне битов), но при сравнении они считаются равными. Если значение с плавающей точкой может содержать отрицательный ноль, требуются дополнительные действия, чтобы для него выдавался тот же хеш, что и для положительного нуля.

Оператор соединения по хешу должен иметь коммутирующий (это может быть тот же оператор, если у него два операнда одного типа, либо связанный оператор равенства, в противном случае), относящийся к тому же семейству операторов. В случае его отсутствия, при попытке использования оператора возможны ошибки планировщика. Также желательно (хотя это строго не требуется), чтобы в семействе операторов хеширования, поддерживающем несколько типов данных, определялись операторы равенства для всех комбинаций этих типов данных; это способствует лучшей оптимизации.

Примечание

Функция, реализующая оператор соединения по хешу, должна быть постоянной (IMMUTABLE) или стабильной (STABLE). Если эта функция изменчивая (VOLATILE), система никогда не будет применять этот оператор для соединения по хешу.

Примечание

Если оператор соединения по хешу реализуется строгой функцией (STRICT), эта функция также должна быть полной: то есть она должна возвращать true или false, но не NULL, для любых двух аргументов, отличных от NULL. Если это правило не соблюдается, оптимизация операций `IN` с хешем может приводить к неверным результатам. (В частности, выражение `IN` может вернуть false, когда правильным ответом, согласно стандарту, должен быть NULL, либо выдать ошибку с сообщением о том, что оно не готов к результату NULL.)

36.15.6. `MERGES`

Предложение `MERGES`, если присутствует, говорит системе, что для соединений с применением этого оператора допустимо использовать метод соединения слиянием. `MERGES` имеет смысл только для бинарного оператора, который возвращает `boolean`, и на практике этот оператор должен выражать равенство значений некоторого типа данных или пары типов данных.

Идея объединения слиянием заключается в упорядочивании таблиц слева и справа и затем параллельном сканировании их. Поэтому оба типа данных должны поддерживать сортировку в полном объёме, а оператор соединения должен давать положительный результат только для пар значений, оказавшихся в «одном месте» при определённом порядке сортировки. На практике это означает, что оператор соединения должен работать как проверка на равенство. Но при этом возможно объединить слиянием два различных типа данных, если они совместимы логически. Например, оператор проверки равенства `smallint` и `integer` может применяться для соединений слиянием; понадобятся только операторы сортировки, приводящие оба типа данных в логически совместимые последовательности.

Чтобы оператор соединения имел характеристику `MERGES`, он должен являться членом семейства операторов индекса `btree`, реализующим равенство. Это требование откладывается, когда оператор только создаётся, ведь нужное семейство операторов, разумеется, ещё не может существовать. Но этот оператор не будет фактически применяться для соединений слиянием, пока не будет найдено соответствующее семейство операторов. Таким образом, флаг `MERGES` только подсказывает планировщику, что стоит обратиться к соответствующему семейству.

Оператор соединения слиянием должен иметь коммутирующий (это может быть тот же оператор, если у него два операнда одного типа, либо связанный оператор равенства, в противном случае), относящийся к тому же семейству операторов. В случае его отсутствия, при попытке использования оператора возможны ошибки планировщика. Также желательно (хотя это строго не требуется), чтобы в семействе операторов `btree`, поддерживающем несколько типов данных, определялись операторы равенства для всех комбинаций этих типов данных; это способствует лучшей оптимизации.

Примечание

Функция, реализующая оператор соединения слиянием, должна быть постоянной (IMMUTABLE) или стабильной (STABLE). Если эта функция изменчивая (VOLATILE), система никогда не будет применять этот оператор для соединения слиянием.

36.16. Интерфейсы расширений для индексов

Описанные до этого процедуры позволяли определять новые типы, функции и операторы. Однако мы ещё не можем определить индекс по столбцу нового типа данных. Для этого нам потребуется создать *класс операторов* для нового типа данных. Далее в этом разделе мы продемонстрируем эту концепцию на примере: мы создадим новый класс операторов для метода индекса-B-дерева, в котором будут храниться комплексные числа и сортироваться по возрастанию абсолютного значения.

Классы операторов могут объединяться в *семейства операторов*, выражающие зависимости между семантически совместимыми классами. Когда вводится один тип данных, достаточно класса операторов, так что мы начнём с него, а к семействам операторов вернёмся позже.

36.16.1. Методы индексов и классы операторов

Классы операторов связываются с индексными методами доступа, например B-дерево или GIN. Пользовательский индексный метод доступа можно определить с помощью команды CREATE ACCESS METHOD. За подробностями обратитесь к Главе 60.

Процедуры метода индекса непосредственно ничего не знают о типах данных, с которыми будет применяться этот метод. Вместо этого, набор операций, которые нужны методу индекса для работы с конкретным типом данных, определяется *классом операторов*. Классы операторов называются так потому, что они определяют множество операторов в предложении `WHERE`, которые могут использоваться с индексом (т. е. могут быть сведены к сканированию индекса). В классе операторов могут также определяться некоторые *опорные функции*, необходимые для внутренних операций метода индекса, но они не соответствуют напрямую каким-либо операторам предложения `WHERE`, которые могут обрабатываться с индексом.

Для одного типа данных и метода индекса можно определить несколько классов операторов. Благодаря этому, для одного типа данных можно использовать несколько семантически разных вариантов индексирования. Например, индекс-B-дерево требует, чтобы для каждого типа данных, с которым он работает, определялся порядок сортировки. Для типа комплексных чисел может быть полезен класс операторов B-дерева, сортирующий данные по модулю комплексного числа, и ещё один, сортирующий по вещественной части, и т. п. Обычно предполагается, что один из классов операторов будет применяться чаще других, и тогда он помечается как класс по умолчанию для данного типа и метода индекса.

Одно и то же имя класса операторов может использоваться для разных методов индекса (например, для методов индекса-B-дерева или хеш-индекса применяются классы операторов `int4_ops`), но все такие классы являются независимыми и должны определяться отдельно.

36.16.2. Стратегии методов индексов

Операторам, которые связываются с классом операторов, назначаются «номера стратегий», определяющие роль каждого оператора в контексте его класса. Например, в B-дереве должен быть строгий порядок ключей с отношениями меньше/больше, так что в данном контексте представляют интерес операторы «меньше» и «больше или равно». Так как Postgres Pro позволяет пользователям определять операторы произвольным образом, Postgres Pro не может просто посмотреть на имя оператора (< или >=) и сказать, какое сравнение он выполняет. Вместо этого для метода индекса определяется набор «стратегий», которые можно считать обобщёнными операторами. Каждый класс операторов устанавливает, какие фактические операторы соответствуют стратегиям для определённого типа данных и интерпретации семантики индекса.

Для метода индекса-B-дерева определены пять стратегий, описанных в Таблице 36.3.

Таблица 36.3. Стратегии B-дерева

Операция	Номер стратегии
меньше	1
меньше или равно	2
равно	3
больше или равно	4
больше	5

Индексы по хешу поддерживают только сравнение на равенство, так что они используют только одну стратегию, показанную в Таблице 36.4.

Таблица 36.4. Стратегии хеша

Операция	Номер стратегии
равно	1

Индексы GiST более гибкие: для них вообще нет фиксированного набора стратегий. Вместо этого опорная процедура «согласованности» каждого конкретного класса операторов GiST интерпретирует номера стратегий как ей угодно. Например, некоторые из встроенных классов операторов для индексов GiST индексируют двумерные геометрические объекты, и реализуют стратегии «R-дерева», показанные в Таблице 36.5. Четыре из них являются истинно двумерными проверками (overlaps, same, contains, contained by); другие четыре учитывают только ординаты, а ещё четыре проводят же проверки только с абсциссами.

Таблица 36.5. Стратегии двумерного «R-дерева» индекса GiST

Операция	Номер стратегии
строго слева от	1
не простирается правее	2
пересекается с	3
не простирается левее	4
строго справа от	5
одинаковы	6
содержит	7
содержится в	8
не простирается выше	9
строго ниже	10
строго выше	11
не простирается ниже	12

Индексы SP-GiST такие же гибкие, как и индексы GiST: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 36.6 приведены номера стратегий, установленные для встроенных классов операторов для точек.

Таблица 36.6. Стратегии SP-GiST для точек

Операция	Номер стратегии
строго слева от	1
строго справа от	5
одинаковы	6
содержится в	8
строго ниже	10
строго выше	11

Индексы GIN такие же гибкие, как и индексы GiST и SP-GiST: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 36.7 приведены номера стратегий, установленные для встроенного класса операторов для массивов.

Таблица 36.7. Стратегии GIN для массивов

Операция	Номер стратегии
пересекается с	1
содержит	2
содержится в	3

Операция	**Номер стратегии**
равно	4

Индексы BRIN такие же гибкие, как и индексы GiST, SP-GiST и GIN: для них не задаётся фиксированный набор стратегий. Вместо этого опорные процедуры каждого класса операторов интерпретируют номера стратегий в соответствии с определением класса операторов. В качестве примера, в Таблице 36.8 приведены номера стратегий, используемые встроенными классами операторов `Minmax`.

Таблица 36.8. Стратегии BRIN Minmax

Операция	**Номер стратегии**
меньше	1
меньше или равно	2
равно	3
больше или равно	4
больше	5

Заметьте, что все вышеперечисленные операторы возвращают булевы значения. На практике все операторы, определённые как операторы поиска для метода индекса, должны возвращать тип `boolean`, так как они должны находиться на верхнем уровне предложения `WHERE`, чтобы для них применялся индекс. (Некоторые методы доступа по индексу также поддерживают *операторы упорядочивания*, которые обычно не возвращают булевы значения; это обсуждается в Подразделе 36.16.7.)

36.16.3. Опорные процедуры метода индекса

Стратегии обычно не дают системе достаточно информации, чтобы понять, как использовать индекс. На практике, чтобы методы индекса работали, необходимы дополнительные опорные процедуры. Например, метод индекса-B-дерева должен уметь сравнивать два ключа и определять, больше, равен или меньше ли первый второго. Аналогично, метод индекса по хешу должен уметь сравнивать хеш-коды значений ключа. Эти операции не соответствуют операторам, которые применяются в условиях в командах SQL; это внутрисистемные подпрограммы, используемые методами индекса.

Так же, как и со стратегиями, класс операторов определяет, какие конкретные функции должны играть каждую из ролей для определённого типа данных и интерпретации семантики индекса. Для метода индекса определяется набор нужных ему функций, а класс оператора выбирает нужные функции для применения, назначая им «номера опорных функций», определяемые методом индекса.

Некоторые классы операторов дополнительно позволяют задать параметры, управляющие их поведением. У всех встроенных индексных методов доступа имеется необязательная опорная функция `options`, которая определяет набор параметров, поддерживаемых данным классом.

Для B-деревьев требуется опорная функция сравнения и могут предоставляться четыре дополнительные опорные функции по выбору разработчика класса операторов, описанные в Таблице 36.9. Требования к этим опорным функциям подробно рассматриваются в Подразделе 62.1.3.

Таблица 36.9. Опорные функции B-деревьев

Функция	**Номер опорной функции**
Сравнивает два ключа и возвращает целое меньше нуля, ноль или целое больше нуля, показывающее, что первый ключ меньше, равен или больше второго	1
Возвращает адреса вызываемых из C опорных функций (или функции) сортировки (необязательная)	2

Функция	Номер опорной функции
Сравнивает проверяемое значение с базовым плюс/минус смещение и возвращает true или false в зависимости от результата сравнения (необязательная)	3
Определяет, может ли в индексах, использующих данный класс операторов, безопасно применяться реализованное в btree исключение дубликатов (необязательная)	4
Определяет параметры, относящиеся к данному классу операторов (необязательная)	5

Для хеш-индексов требуется одна опорная функция, и ещё две могут задаваться по выбору разработчика класса операторов, как показано в Таблице 36.10.

Таблица 36.10. Опорные функции хеша

Функция	Номер опорной функции
Вычисляет 32-битное значение хеша для ключа	1
Вычисляет 64-битное значение хеша для ключа с заданной 64-битной солью; если значение соли равно 0, младшие 32 бита результата должны соответствовать значению, которое было бы вычислено функцией 1 (необязательная)	2
Определяет параметры, относящиеся к данному классу операторов (необязательная)	3

Для индексов GiST предусмотрены одиннадцать опорных функций, шесть из которых необязательные; они описаны в Таблице 36.11. (За дополнительными сведениями обратитесь к Разделу 62.2.)

Таблица 36.11. Опорные функции GiST

Функция	Описание	Номер опорной функции
`consistent`	определяет, удовлетворяет ли ключ условию запроса	1
`union`	вычисляет объединение набора ключей	2
`compress`	вычисляет сжатое представление ключа или индексируемого значения (необязательная)	3
`decompress`	вычисляет развёрнутое представление сжатого ключа (необязательная)	4
`penalty`	вычисляет стоимость добавления нового ключа в поддерево с заданным ключом	5
`picksplit`	определяет, какие записи страницы должны быть перемещены в новую страницу, и вычисляет ключи объединения для результирующих страниц	6
`same`	сравнивает два ключа и возвращает true, если они равны	7
`distance`	определяет дистанцию от ключа до искомого значения (необязательная)	8
`fetch`	вычисляет исходное представление сжатого ключа для сканирования только по индексу (необязательная)	9

Функция	Описание	Номер опорной функции
`options`	определяет параметры, относящиеся к данному классу операторов (необязательная)	10
`sortsupport`	предоставляет компаратор для сортировки, который будет использоваться для быстрого построения индекса (необязательная)	11

Для индексов SP-GiST предусмотрены шесть опорных функций, одна из которых необязательная; они описаны в Таблице 36.12. (За дополнительными сведениями обратитесь к Разделу 62.3.)

Таблица 36.12. Опорные функции SP-GiST

Функция	Описание	Номер опорной функции
`config`	предоставляет основную информацию о классе операторов	1
`choose`	определяет, как вставить новое значение во внутренний элемент	2
`picksplit`	определяет, как разделить множество значений	3
`inner_consistent`	определяет, в каких внутренних ветвях нужно искать заданное значение	4
`leaf_consistent`	определяет, удовлетворяет ли ключ условию запроса	5
`options`	определяет параметры, относящиеся к данному классу операторов (необязательная)	6

Для индексов GIN предусмотрены семь опорных функций, четыре из которых необязательные; они описаны в Таблице 36.13. (За дополнительными сведениями обратитесь к Разделу 62.4.)

Таблица 36.13. Опорные функции GIN

Функция	Описание	Номер опорной функции
`compare`	сравнивает два ключа и возвращает целое меньше нуля, ноль или целое больше нуля, показывающее, что первый ключ меньше, равен или больше второго	1
`extractValue`	извлекает ключи из индексируемого значения	2
`extractQuery`	извлекает ключи из условия запроса	3
`consistent`	определяет, соответствует ли значение условию запроса (логическая вариация) (не требуется, если присутствует опорная функция 6)	4
`comparePartial`	сравнивает частичный ключ из запроса с ключом из индекса и возвращает целое число меньше нуля, ноль или больше нуля, показывающее, что GIN должен игнорировать эту запись индекса, принять её как соответствующую или прекратить сканирование индекса (необязательная)	5
`triConsistent`	определяет, соответствует ли значение условию запроса (троичная вариация) (не требуется, если присутствует опорная функция 4)	6

Функция	**Описание**	**Номер опорной функции**
`options`	определяет параметры, относящиеся к данному классу операторов (необязательная)	7

Для индексов BRIN предусмотрены пять базовых опорных функций, перечисленных в Таблице 36.14. Для некоторых видов базовых функций может потребоваться предоставить дополнительные опорные функции. (За дополнительными сведениями обратитесь к Подразделу 62.5.3.)

Таблица 36.14. Опорные функции BRIN

Функция	**Описание**	**Номер опорной функции**
`opcInfo`	возвращает внутреннюю информацию, описывающую сводные данные по индексированным столбцам	1
`add_value`	добавляет новое значение в существующий сводный кортеж индекса	2
`consistent`	определяет, удовлетворяет ли значение условию запроса	3
`union`	вычисляет объединение двух обобщающих кортежей	4
`options`	определяет параметры, относящиеся к данному классу операторов (необязательная)	5

В отличие от операторов поиска, опорные функции возвращают тот тип данных, который ожидает конкретный метод индекса; например, функция сравнения для B-деревьев возвращает знаковое целое. Количество и типы аргументов для каждой опорной функции так же зависят от метода индекса. Для методов B-дерева и хеша функции сравнения и хеширования принимают те же типы данных, что и операторы, включённые в класс операторов, но для большинства опорных функций GiST, SP-GiST, GIN и BRIN это не так.

36.16.4. Пример

Теперь, когда мы познакомились с основными идеями, мы можем перейти к обещанному примеру создания нового класса операторов. Класс операторов включает операторы, сортирующие комплексные числа по порядку абсолютных значений, поэтому мы выбрали для него имя `complex_abs_ops`. Во-первых, нам понадобится набор операторов. Процедура определения операторов была рассмотрена в Разделе 36.14. Для класса операторов B-деревьев нам понадобятся операторы:

- абсолютное-значение меньше (стратегия 1)
- абсолютное-значение меньше-или-равно (стратегия 2)
- абсолютное-значение равно (стратегия 3)
- абсолютное-значение больше-или-равно (стратегия 4)
- абсолютное-значение больше (стратегия 5)

Чтобы не провоцировать ошибки при определении связанного набора операторов сравнения, лучше всего сначала написать вспомогательную функцию сравнения для B-дерева, а затем написать другие функции как однострочные оболочки этой вспомогательной функции. Это уменьшит вероятность получения несогласованных результатов в исключительных случаях. Следуя этому подходу, мы сначала напишем:

```
#define Mag(c)   ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
```

```
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

Теперь функция «меньше» будет выглядеть так:

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

Остальные четыре функции отличаются от неё только тем, как сравнивают результат внутренней функции с нулём.

Затем мы объявим в SQL функции и операторы на основе этих функций:

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'имя_файла', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

Важно правильно определить обратные и коммутирующие операторы, а также подходящие функции избирательности ограничения и соединения; иначе оптимизатор не сможет использовать наш индекс эффективно.

Здесь также стоит обратить внимание на следующее:

- Учтите, что может быть только один оператор с именем, например, `=`, который будет принимать тип `complex` с двух сторон. В этом случае у нас не будет другого оператора `=` для `complex`, но если мы создаём практически полезный тип данных, вероятно, мы захотим, чтобы оператор `=` проверял обычное равенство двух комплексных чисел (а не равенство их абсолютных значений). В этом случае для `complex_abs_eq` нужно выбрать какое-то другое имя оператора.
- Хотя в Postgres Pro разные функции могут иметь одинаковые имена SQL, если у них различные типы аргументов, в C только одна глобальная функция может иметь заданное имя. Поэтому не следует давать функции на C имя вроде `abs_eq`. Во избежание конфликтов с функциями для других типов данных, в имя функции на C обычно включается имя конкретного типа данных.
- Мы могли быть дать нашей функции имя `abs_eq` в SQL, рассчитывая на то, что Postgres Pro отличит её от любых других одноимённых функций SQL по типам аргументов. Но в данном случае для упрощения примера мы дали ей одинаковые имена на уровне C и уровне SQL.

На следующем этапе регистрируется опорная процедура, необходимая для B-деревьев. В нашем примере код C, реализующий её, находится в том же файле, что и функции операторов. Мы объявляем эту процедуру так:

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'имя_файла'
    LANGUAGE C IMMUTABLE STRICT;
```

Теперь, когда мы объявили требуемые операторы и опорную функцию, мы наконец можем создать класс операторов:

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

Вот и всё! Теперь должно быть возможно создавать и использовать индексы-B-деревья по столбцам `complex`.

Операторы можно было записать более многословно, например, так:

```
        OPERATOR        1       < (complex, complex) ,
```

но в этом необходимости, так как эти операторы принимают тот же тип данных, для которого определяется класс операторов.

В приведённом примере предполагается, что этот класс операторов будет классом операторов B-дерева по умолчанию для типа `complex`. Если вам это не нужно, просто опустите слово `DEFAULT`.

36.16.5. Семейства и классы операторов

До этого мы неявно полагали, что класс операторов работает только с одним типом данных. Хотя в конкретном индексируемом столбце, определённо, может быть только один тип данных, часто бывает полезно индексировать операции, сравнивающие значение столбца со значением другого типа. Также, если в сочетании с классом операторов возможно применение оператора, работающего с двумя типами, для другого типа данных обычно тоже создаётся собственный класс. В таких случаях полезно установить явную связь между связанными классами, так как это поможет планировщику оптимизировать SQL-запросы (особенно для классов операторов B-дерева, потому что планировщик хорошо знает, как работать с ними).

Для удовлетворения этих потребностей в Postgres Pro введена концепция *семейства операторов*. Семейство операторов содержит один или несколько классов операторов и может также содержать индексируемые операторы и соответствующие опорные функции, принадлежащие к семейству в целом, но не к какому-то одному классу в нём. Мы называем такую связь операторов и функций с семейством «слабой», в отличие от обычной связи с определённым классом. Как правило, классы содержат операторы с операндами одного типа, тогда как межтиповые операторы слабо связываются с семейством.

Все операторы и функции в семействе операторов должны иметь совместимую семантику; требования к совместимости устанавливаются методом индекса. Вы можете спросить, зачем в таком случае вообще выделять конкретные подмножества семейства в виде классов операторов; и на самом деле во многих случаях деление на классы не имеет значения, важно только связывание с семейством. Смысл классов операторов в том, что они определяют, какая часть семейства необходима для поддержки некоторого индекса. Если существует индекс, использующий класс операторов, этот класс нельзя будет удалить, не удалив индекс — но другие части семейства, а именно, другие классы операторов и слабосвязанные операторы, удалить можно. Таким образом, класс операторов должен определяться так, чтобы он содержал минимальный набор операторов и функций, обоснованно требующихся для работы с индексом по определённому типу данных, а несущественные операторы могут добавляться в качестве слабосвязанных членов в семейство операторов.

В качестве примера, в Postgres Pro есть встроенное семейство операторов B-дерева `integer_ops`, включающее классы операторов `int8_ops`, `int4_ops` и `int2_ops` для индексов по столбцам `bigint` (`int8`), `integer` (`int4`) и `smallint` (`int2`), соответственно. В этом семействе также содержатся операторы межтипового сравнения, позволяющие сравнивать значения любых двух этих типов, так что индексом по любому из этих типов можно пользоваться, выполняя сравнение с другим типом. Это семейство можно представить такими определениями:

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- standard int8 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ,
  FUNCTION 3 in_range(int8, int8, int8, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  -- standard int4 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ,
  FUNCTION 3 in_range(int4, int4, int4, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  -- standard int2 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ,
  FUNCTION 2 btint2sortsupport(internal) ,
  FUNCTION 3 in_range(int2, int2, int2, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
  -- cross-type comparisons int8 vs int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,
```

```
-- cross-type comparisons int8 vs int4
OPERATOR 1 < (int8, int4) ,
OPERATOR 2 <= (int8, int4) ,
OPERATOR 3 = (int8, int4) ,
OPERATOR 4 >= (int8, int4) ,
OPERATOR 5 > (int8, int4) ,
FUNCTION 1 btint84cmp(int8, int4) ,

-- cross-type comparisons int4 vs int2
OPERATOR 1 < (int4, int2) ,
OPERATOR 2 <= (int4, int2) ,
OPERATOR 3 = (int4, int2) ,
OPERATOR 4 >= (int4, int2) ,
OPERATOR 5 > (int4, int2) ,
FUNCTION 1 btint42cmp(int4, int2) ,

-- cross-type comparisons int4 vs int8
OPERATOR 1 < (int4, int8) ,
OPERATOR 2 <= (int4, int8) ,
OPERATOR 3 = (int4, int8) ,
OPERATOR 4 >= (int4, int8) ,
OPERATOR 5 > (int4, int8) ,
FUNCTION 1 btint48cmp(int4, int8) ,

-- cross-type comparisons int2 vs int8
OPERATOR 1 < (int2, int8) ,
OPERATOR 2 <= (int2, int8) ,
OPERATOR 3 = (int2, int8) ,
OPERATOR 4 >= (int2, int8) ,
OPERATOR 5 > (int2, int8) ,
FUNCTION 1 btint28cmp(int2, int8) ,

-- cross-type comparisons int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ,

-- cross-type in_range functions
FUNCTION 3 in_range(int4, int4, int8, boolean, boolean) ,
FUNCTION 3 in_range(int4, int4, int2, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int8, boolean, boolean) ,
FUNCTION 3 in_range(int2, int2, int4, boolean, boolean) ;
```

Заметьте, что в определении семейства «перегружаются» номера стратегий операторов и опорных функций: каждый номер фигурирует в семействе неоднократно. Это допускается, если для каждого экземпляра определённого номера задаются свои типы данных. Экземпляры, у которых оба входных типа совпадают с входным типом класса операторов, являются первичными операторами и опорными функциями для этого класса, и в большинстве случаев они должны объявляться в составе класса операторов, а не быть слабосвязанными членами семейства.

В семействе операторов B-дерева все операторы должны быть совместимыми в контексте сортировки; это подробно описано в Подразделе 62.1.2. Для каждого оператора в семействе должна существовать опорная функция, принимающая на вход те же два типа, что и оператор. Семейство рекомендуется делать полным, то есть включать в него все операторы для каждого сочетания типов данных. В классы операторов следует включать только однотиповые операторы и опорные функции для определённого типа данных.

Чтобы создать семейство операторов хеширования для нескольких типов данных, необходимо создать совместимые функции поддержки хеша для каждого типа данных, который будет поддерживать семейство. Здесь под совместимостью понимается гарантия получения одного хеш-кода для любых двух значений, которые операторы сравнения в этом семействе считают равными, даже если они имеют разные типы. Обычно это сложно осуществить, когда типы имеют разное физическое представление, но в некоторых случаях всё же возможно. Более того, преобразование значения одного типа данных, представленного в семействе операторов, к другому типу, также представленному в этом семействе, путём неявного или двоичного сведения не должно менять значение вычисляемого хеша. Заметьте, что единственная опорная функция задаётся для типа данных, а не для оператора равенства. Семейство рекомендуется делать полным, то есть включить в него оператор равенства для всех сочетаний типов данных. В классы операторов следует включать только однотиповый оператор равенства и опорную функция для определённого типа данных.

В индексах GiST, SP-GiST и GIN межтиповые операции явно не выражены. Множество поддерживаемых операторов определяется только теми операциями, которые могут выполнять основные опорные функции заданного класса операторов.

В BRIN требования зависят от инфраструктуры, предоставляющей классы операторов. Для классов операторов, построенных на инфраструктуре `minmax`, требуется то же поведение, что и для семейств операторов B-дерева: все операторы в семействе должны поддерживать совместимый порядок, а приведения не должны влиять на установленный порядок сортировки.

Примечание

До версии 8.3 в PostgreSQL не было понятия семейства операторов, поэтому любые межтиповые операторы, предназначенные для применения с индексом, должны были привязываться непосредственно к классу оператора индекса. Хотя этот подход по-прежнему работает, он считается устаревшим, потому что он создаёт слишком много зависимостей для индекса, а также потому, что планировщик может выполнять межтиповые сравнения более эффективно, когда для обоих типов данных определены операторы в одном семействе.

36.16.6. Системные зависимости от классов операторов

Postgres Pro использует классы операторов для наделения операторов такими свойствами, которые могут быть полезны не только для индексов. Поэтому классы операторов могут быть полезны, даже если вы не намерены индексировать столбцы со значениями определённого вами типа.

В частности, это касается SQL-конструкций `ORDER BY` и `DISTINCT`, для которых требуется сравнивать и упорядочивать значения. Чтобы эти конструкции работали с определённым пользователем типом данных, Postgres Pro задействует класс операторов B-дерева по умолчанию для этого типа. Член «равно» этого класса определяет, как система будет понимать равенство значений для `GROUP BY` и `DISTINCT`, а порядок сортировки, задаваемый классом операторов, определяет порядок `ORDER BY` по умолчанию.

Если класс операторов B-дерева по умолчанию для типа данных не определён, система будет искать класс операторов хеширования по умолчанию. Но так как подобный класс поддерживает только равенство, с ним будет возможна только группировка, но не сортировка.

Если для типа не определён класс операторов по умолчанию, попытавшись использовать эти конструкции SQL с данным типом, вы получите ошибку вида «не удалось найти оператор сортировки».

Примечание

До версии PostgreSQL 7.4, в операциях сортировки и группировки неявно использовались операторы с именами =, < и >. С новым подходом, опирающимся на классы операторов по умолчанию, система не делает никаких предположений о поведении операторов по их именам.

Сортировка с нестандартным классом операторов B-дерева возможна, если указать в предложении `USING` оператор «меньше или равно» в данном классе:

```
SELECT * FROM mytable ORDER BY somecol USING ~<~;
```

Также возможно выполнить сортировку в порядке по убыванию, если указать в `USING` оператор «больше или равно».

Сравнение массивов пользовательских типов также производится в зависимости от семантики, определённой классом операторов B-дерева. Если класс операторов B-дерева по умолчанию для данного типа не определён, но имеется класс операторов хеширования, то будет поддерживаться сравнение массивов, но не упорядочивание.

Ещё одна возможность языка SQL, которая требует дополнительных знаний о типе данных — это указание `RANGE` *`смещение`* `PRECEDING`/`FOLLOWING` для оконных функций (см. Подраздел 4.2.8). Для запроса вида

```
SELECT sum(x) OVER (ORDER BY x RANGE BETWEEN 5 PRECEDING AND 10 FOLLOWING)
  FROM mytable;
```

недостаточно знать, как упорядочить `x`; база данных должна также понимать, как «вычесть 5» или «прибавить 10» к значению `x` в текущей строке для определения рамок текущего окна. Сравнивать результирующие границы со значениями `x` в других строках можно используя операторы сравнения, предоставленные классом операторов B-дерева, которые задают упорядочивание `ORDER BY`, но операторы сложения и вычитания не входят в этот класс операторов. Так какие же использовать в этом случае? Жёстко зафиксировать операторы в коде было бы нежелательно, так как при различных порядках сортировки (с различными классами операторов B-дерева) поведение может меняться. Поэтому класс операторов B-дерева позволяет задать опорную функцию *in_range*, осуществляющую сложение и вычитание в соответствии с порядком сортировки. Опорных функций in_range может быть даже несколько, если в качестве смещения в предложениях `RANGE` имеет смысл передавать данные разных типов. Если в классе операторов B-дерева, связанном с указанным для окна предложением `ORDER BY`, нет подходящей опорной функции in_range, то указание `RANGE` *`смещение`* `PRECEDING`/`FOLLOWING` не поддерживается.

Также важно отметить, что оператор равенства, указанный в семействе операторов хеширования, является кандидатом для применения при слиянии и агрегации по хешу, а также при связанной оптимизации. Семейство операторов хеширования играет в данном случае определяющую роль, так как именно в нём задаётся функция хеширования.

36.16.7. Операторы упорядочивания

Некоторые методы доступа индексов (в настоящее время только GiST и SP-GiST) поддерживают концепцию *операторов упорядочивания*. Операторы, которые мы обсуждали до этого, были *операторами поиска*. Оператором поиска называется такой оператор, для которого можно выполнить поиск по индексу и найти все строки, удовлетворяющие условию `WHERE` *`индексированный_столбец оператор константа`*. Заметьте, что при этом ничего не говорится о порядке, в котором будут возвращены подходящие строки. Оператор упорядочивания, напротив, не ограничивает набор возвращаемых строк, но определяет их порядок. С таким оператором, просканировав индекс, можно получить строки в порядке, заданным указанием `ORDER BY` *`индексированный_столбец оператор константа`*. Такое определение объясняется тем, что оно поддерживает поиск ближайшего соседа, если этот оператор вычисляет расстояние. Например, запрос

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

находит десять ближайших к заданной точке мест. Индекс GiST по столбцу location может сделать это эффективно, так как `<->` — это оператор упорядочивания.

Тогда как операторы поиска должны возвращать логические результаты, операторы упорядочивания обычно возвращают другой тип, например, float или numeric для расстояний. Этот тип, как правило, отличается от типа индексируемых данных. Чтобы избежать жёстко запрограммированных предположений о поведении различных типов данных, при объявлении оператора упорядочивания должно указываться семейство операторов B-дерева, определяющее порядок сортировки результирующего типа данных. Как было отмечено в предыдущем разделе, семейства операторов B-дерева определяют понятие упорядочивания для Postgres Pro, так что такое объявление оказывается естественным. Так как оператор `<->` для точек возвращает `float8`, его можно включить в команду создания класса операторов так:

```
OPERATOR 15    <-> (point, point) FOR ORDER BY float_ops
```

где `float_ops` — встроенное семейство операторов, включающее операции с `float8`. Это объявление означает, что индекс может возвращать строки в порядке увеличения значений оператора `<->`.

36.16.8. Особенности классов операторов

Есть ещё две особенности классов операторов, которые мы до этого не обсуждали, в первую очередь потому, что они не востребованы для наиболее часто применяемых методов индексов.

Обычно объявление оператора в качестве члена класса операторов (или семейства) означает, что метод индекса может получить точно набор строк, который удовлетворяет условию `WHERE` с этим оператором. Например, запрос:

```
SELECT * FROM table WHERE integer_column < 4;
```

может быть удовлетворён в точности индексом-B-деревом по целочисленному столбцу. Но бывают случаи, когда индекс полезен как приблизительный указатель на соответствующие строки. Например, если индекс GiST хранит только прямоугольники, описанные вокруг геометрических объектов, он не может точно удовлетворить условие `WHERE`, которое проверяет пересечение не прямоугольных объектов, а например, многоугольников. Однако этот индекс можно применить, чтобы найти объекты, для которых описанные вокруг прямоугольники пересекаются с прямоугольником, описанным вокруг целевого объекта, а затем провести точную проверку пересечения только для найденных по индексу объектов. Если это имеет место, такой индекс называется «неточным» для оператора. Для реализации поиска по неточному индексу метод индекса возвращает флаг *recheck* (перепроверить), когда строка может действительно удовлетворять, а может не удовлетворять условию запроса. Затем исполнитель запроса перепроверяет полученную строку по исходному условию запроса и определяет, должна ли она выдаваться как действительно соответствующая ему. Этот подход работает, если индекс гарантированно выдаёт все требуемые строки плюс, возможно, дополнительные строки, которые можно исключить, вызвав первоначальный оператор. Методы индексов, поддерживающие неточный поиск (в настоящее время, GiST, SP-GiST и GIN), позволяют устанавливать флаг recheck опорным функциям отдельных классов операторов, так что по сути это особенность класса операторов.

Вернёмся к ситуации, когда мы храним в индексе только прямоугольник, описанный вокруг сложного объекта, такого как многоугольник. В этом случае нет большого смысла хранить в элементе индекса весь многоугольник — мы можем с тем же успехом хранить более простой объект типа `box`. Это отклонение выражается указанием `STORAGE` в команде `CREATE OPERATOR CLASS`, которое записывается примерно так:

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

В настоящее время только методы индексов GiST, SP-GiST, GIN и BRIN позволяют задать в `STORAGE` тип, отличный от типа данных столбца. В GiST преобразованием данных, связанным с использованием `STORAGE`, должны заниматься опорные процедуры `compress` и `decompress`. Аналогичным образом в SP-GiST опорная процедура `compress` занимается преобразованием данных к типу хранения, если они имеют другой тип; если класс операторов SP-GiST поддерживает и получение данных, обратное преобразование выполняет функция `consistent`. В GIN тип `STORAGE` определяет тип значений «ключа», который обычно отличается от типа индексируемого столбца — например, в классе операторов для столбцов с целочисленным массивом ключами могут быть просто целые числа. За извлечение ключей из индексированных значений в GIN отвечают опорные функции `extractValue` и `extractQuery`. BRIN похож на GIN: в нём тип `STORAGE` определяет тип хранимых обобщённых значений, а опорные процедуры классов операторов отвечают за правильное прочтение этих значений.

36.17. Упаковка связанных объектов в расширение

Полезное расширение Postgres Pro обычно включает несколько объектов SQL; например, с появлением нового типа данных могут потребоваться новые функции, новые операторы и новые классы операторов. Все эти объекты удобно собрать в один пакет, с тем чтобы упростить управление базой данных. В Postgres Pro такие пакеты называются *расширениями*. Чтобы определить расширение, вам понадобится как минимум *файл скрипта* с командами SQL, создающими объекты расширения, и *управляющий файл*, в котором определяются несколько базовых свойств самого расширения. Если расширение написано на C, в него обычно также включается файл разделяемой библиотеки, содержащий скомпилированный код. Обеспечив наличие этих файлов, загрузить их в базу данных можно простой командой `CREATE EXTENSION`.

Основное преимущество расширений по сравнению с обычным SQL-скриптом, загружающим множество «разрозненных» объектов в базу данных, состоит в том, что Postgres Pro будет понимать, что объекты расширения связаны вместе. Вы можете удалить все объекты одной командой `DROP EXTENSION` (разрабатывать отдельный скрипт «uninstall» не требуется). Ещё полезнее то, что утилита pg_dump знает, что не нужно выгружать отдельные объекты, составляющие расширение — вместо этого она просто включит в архивный файл команду `CREATE EXTENSION`. Это кардинально упрощает миграцию на новую версию расширения, которая может содержать новые или другие объекты по сравнению с предыдущей версией. Заметьте, однако, что при загрузке такого архива в базу данных обязательно наличие скрипта, управляющего файла и других файлов расширения.

Postgres Pro не позволит удалить отдельный объект, содержащийся в расширении, кроме как при удалении всего расширения. Также имейте в виду, что вы можете изменить определение объекта, относящегося к расширению (например, командой `CREATE OR REPLACE FUNCTION` для функции), но изменённое определение не будет выгружено утилитой pg_dump. Такие изменения обычно разумны, только если они параллельно отражаются в файле скрипта расширения. (Для таблиц, содержащих конфигурационные данные, предусмотрены специальные средства; см. Подраздел 36.17.3.) В производственной среде обычно лучше создавать скрипт обновления расширения, который будет изменять относящиеся к расширению объекты.

Скрипт расширения может устанавливать права доступа для объектов, являющихся частью расширения, выполняя команды `GRANT` и `REVOKE`. Окончательный набор прав для каждого объекта (если они заданы) будет сохранён в системном каталоге `pg_init_privs`. При использовании pg_dump в выгружаемый скрипт будет выведена команда `CREATE EXTENSION` с последующими операторами `GRANT` и `REVOKE`, которые установят права, имевшие место в момент выгрузки.

Postgres Pro в настоящее время не поддерживает скрипты расширений, выполняющие операторы `CREATE POLICY` или `SECURITY LABEL`. Ожидается, что такие команды будут выполняться после того, как расширение будет создано. Выгружая данные, pg_dump будет также включать в вывод все политики RLS и метки безопасности.

Механизм расширений также предоставляет средства для поддержки дополнительных скриптов, призванных изменять определение объектов SQL, содержащихся в расширении. Например, если версия расширения 1.1, по сравнению с версией 1.0, добавляет одну функцию и изменяет тело другой функции, автор расширения может предоставить *скрипт обновления*, который произведёт именно эти два изменения. Затем, воспользовавшись командой `ALTER EXTENSION UPDATE`, можно будет применить эти изменения и отследить, какая версия расширения фактически установлена в заданной базе данных.

Типы SQL-объектов, которые могут быть членами расширения, перечислены в описании `ALTER EXTENSION`. Не могут быть его членами, в частности, объекты уровня кластера, такие как базы данных, роли и табличные пространства, так как расширение существует только в рамках одной базы данных. (Скрипту расширения не запрещается создавать такие объекты, но если он сделает это, они не будут считаться частью расширения.) Также заметьте, что несмотря на то, что таблица может быть членом расширения, её подчинённые объекты, такие как индексы, непосредственными членами расширения считаться не будут. Ещё один важный момент — схемы могут принадлежать расширениям, но не наоборот; поэтому расширение имеет неполное имя и не существует «внутри» какой-либо схемы. Однако объекты-члены расширения, будут относиться к схемам, если это уместно для их типов. Сами расширения могут иметь, а могут не иметь основания владеть схемами, к которым относятся объекты-члены расширения.

Если скрипт расширения создаёт какие-либо временные объекты (например, временные таблицы), эти объекты будут считаться членами расширения до конца текущего сеанса, но удалятся автоматически в конце сеанса, как и должны временные объекты. Это является исключением из правила, запрещающего удаление объектов-членов расширения без удаления всего расширения.

36.17.1. Файлы расширений

Команда `CREATE EXTENSION` задействует управляющий файл расширения, который должен называться по имени расширения, с суффиксом `.control`, и должен быть помещён в каталог сервера `SHAREDIR/extension`. Должен быть также ещё минимум один SQL-скрипт, с именем, соответствующим шаблону *`расширение--версия`*`.sql` (например, `foo--1.0.sql` для версии `1.0` расширения `foo`). По умолчанию скрипт(ы) также помещается в каталог `SHAREDIR/extension`; но в управляющем файле можно задать и другой каталог.

Формат управляющего файла расширения не отличается от формата `postgresql.conf`, а именно представляет собой список присваиваний *`имя_параметра`* `=` *`значение`*, по одному в строке. В нём также допускаются пустые строки и комментарии, начинающиеся с `#`. Все значения, отличные от единственного слова или числа, в нём должны заключаться в кавычки.

В управляющем файле могут устанавливаться следующие параметры:

`directory` (`string`)

Каталог, содержащий SQL-скрипт(ы) расширения. Если только не задан абсолютный путь, это имя рассматривается относительно каталога сервера `SHAREDIR`. По умолчанию подразумевается указание `directory = 'extension'`.

`default_version` (`string`)

Версия расширения по умолчанию (та, которая будет установлена, если в `CREATE EXTENSION` не будет указана никакая версия). Хотя этот параметр можно опустить, это приведёт к ошибке в `CREATE EXTENSION` без явного указания `VERSION`, что вряд ли будет желаемым поведением.

`comment` (`string`)

Комментарий (произвольная строка) к расширению. Комментарий применяется при изначальном создании расширения, но не при обновлениях расширения (так как при этом мог бы заменяться комментарий, заданный пользователем). Комментарий расширения также можно задать посредством команды COMMENT в файле скрипта.

`encoding` (`string`)

Кодировка символов, используемая в файлах скриптов. Её следует указать, если эти файлы содержат символы не из набора ASCII. По умолчанию предполагается, что эти файлы содержат текст в кодировке базы данных.

`module_pathname` (`string`)

Значение этого параметра будет подставляться вместо каждого вхождения `MODULE_PATHNAME` в скриптах. Если этот параметр не задан, подстановка не производится. Обычно для этого параметра устанавливается значение `$libdir/`*`имя_разделяемой_библиотеки`*, а затем в командах `CREATE FUNCTION` для функций на языке C указывается `MODULE_PATHNAME`, чтобы в скриптах не приходилось жёстко задавать имя разделяемой библиотеки.

`requires` (string)

Список имён расширений, от которых зависит данное, например, `requires = 'foo, bar'`. Эти расширения должны быть уже установлены, прежде чем можно будет установить данное.

`no_relocate` (string)

Список имён расширений, от которых зависит данное расширение и которым следует запретить изменение своих схем командой `ALTER EXTENSION ... SET SCHEMA`. Это необходимо, если скрипт данного расширения ссылается на имя требуемой схемы расширения (используя синтаксис `@extschema:`*`имя`*`@`) таким образом, что не может отслеживать переименования.

`superuser` (boolean)

Если этот параметр имеет значение `true` (по умолчанию), только суперпользователи смогут создать это расширение или обновить его до новой версии. (Однако обратите внимание на свойство `trusted`, описанное ниже). Если он имеет значение `false`, для этого будет достаточно прав, необходимых для выполнения команд в установочном скрипте или скрипте обновления. Обычно значение `true` должно устанавливаться, если для выполнения какой-либо из команд в этих скриптах требуются права суперпользователя. Такие команды в любом случае не будут выполнены успешно, но лучше сообщить пользователю об ошибке заранее.

`trusted` (boolean)

Если этот параметр имеет значение `true` (по умолчанию это не так), то расширение, для которого свойство `superuser` равно `true`, смогут устанавливать не только суперпользователи. А именно, установить его смогут любые пользователи, имеющие право `CREATE` в текущей базе данных. Когда пользователь, выполняющий `CREATE EXTENSION`, не является суперпользователем, но ему разрешена установка этого расширения посредством этого параметра, скрипт установки или обновления запускается от имени начального суперпользователя, а не от имени вызывающего пользователя. Этот параметр не играет роли, если свойство `superuser` равно `false`. Вообще говоря, этот параметр не следует устанавливать для расширений, которые могут открыть возможности, иначе доступные только суперпользователям, например, предоставить доступ к файловой системе. Кроме того, если расширение помечается как доверенное, написание безопасных скриптов установки и обновления для него требует дополнительных усилий; см. Подраздел 36.17.6.

`relocatable` (boolean)

Расширение является *перемещаемым*, если относящиеся к нему объекты после создания расширения можно переместить в другую схему. По умолчанию подразумевается `false`, то есть расширение не считается перемещаемым. Подробнее об этом рассказывается в Подразделе 36.17.2.

`schema` (string)

Этот параметр может задаваться только для неперемещаемых расширений. Если он задан, расширение можно будет загрузить только в указанную схему и не в какую другую. Подробнее об этом рассказывается ниже. Параметр `schema` учитывается только при изначальном создании расширения, но не при его обновлении. Подробнее об этом рассказывается в Подразделе 36.17.2.

Помимо главного управляющего файла *`расширение`*`.control`, расширение может включать дополнительные управляющие файлы с именами вида *`расширение`*`--`*`версия`*`.control`. Если они присутствуют, они должны находиться в том же каталоге, что и основной скрипт. Дополнительные управляющие файлы имеют тот же формат, что и основной. Любые параметры, заданные в дополнительном управляющем файле, переопределяют параметры основного файла, когда выполняется установка этой версии расширения или обновление до неё. Однако параметры `directory` и `default_version` в дополнительных управляющих файлах задать нельзя.

SQL-скрипты расширений могут содержать любые команды SQL, за исключением команд управления транзакциями (`BEGIN`, `COMMIT` и т. д.) и команд, которые не могут выполняться внутри блока транзакции (например, `VACUUM`). Это объясняется тем, что эти скрипты неявно выполняются в блоке транзакции.

SQL-скрипты расширений также могут содержать строки, начинающиеся с `\echo`, и они будут игнорироваться (восприниматься как комментарии) механизмом расширений. Это часто используется для вывода ошибки в случае, если этот скрипт выполняется в psql, а не загружается командой `CREATE EXTENSION` (см. пример скрипта в Подразделе 36.17.7). Если такое выполнение не предотвратить, пользователи могут случайно загрузить содержимое расширения как «разрозненные» объекты, а не как собственно расширение, и получить состояние, которое довольно сложно исправить.

Если скрипт расширения содержит строку `@extowner@`, она будет заменена именем (если требуется, заключённым в кавычки) пользователя, выполняющего команду `CREATE EXTENSION` или `ALTER EXTENSION`. Обычно это полезно для доверенных расширений, в которых владельцем внутренних объектов назначается не начальный суперпользователь, а вызывающий пользователь. (Однако это следует делать с осторожностью. Например, если назначить обычного пользователя владельцем функции на языке C, это позволит ему повысить свои права.)

Тогда как файлы скриптов могут содержать любые символы, допустимые в указанной кодировке, управляющие файлы могут содержать только ASCII-символы, так как указать кодировку этих файлов в Postgres Pro нет никакой возможности. На практике это представляет проблему, только если вы хотите использовать символы не из набора ASCII в комментарии расширения. В таких случаях рекомендуется не использовать параметр `comment` в управляющем файле, а вместо этого задать комментарий командой `COMMENT ON EXTENSION` в файле скрипта.

36.17.2. Перемещаемость расширений

У пользователей часто возникает желание загрузить объекты, содержащиеся в расширении, в схему, отличную от той, что выбрал автор расширения. Насколько это поддерживает расширение, описывается одним из трёх уровней:

- Полностью перемещаемое расширение может быть перемещено в другую схему в любое время, даже после того, как оно загружено в базу данных. Это осуществляется командой `ALTER EXTENSION SET SCHEMA`, которая автоматически переименовывает все объекты-члены расширения, перенося их в новую схему. Обычно это возможно, только если в расширении нет никаких внутренних предположений о том, в какой схеме находятся все его объекты. Кроме того, все объекты расширения должны находиться в одной исходной схеме (за исключением объектов, не принадлежащих схемам, как например, процедурные языки). Чтобы пометить расширение как полностью перемещаемое, установите `relocatable = true` в его управляющем файле.
- Расширение может быть перемещаемым в момент установки, но не после. Обычно это имеет место, когда скрипту расширения необходимо явно ссылаться на целевую схему, например, устанавливая свойства `search_path` для функций SQL. Для такого расширения нужно задать `relocatable = false` в его управляющем файле и обращаться к целевой схеме в скрипте по псевдоимени `@extschema@`. Все вхождения этого псевдоимени будут заменены именем выбранной целевой схемы (при необходимости заключённым в двойные кавычки) перед выполнением скрипта. Пользователь может выбрать целевую схему в указании `SCHEMA` команды `CREATE EXTENSION`.
- Если расширение вовсе не поддерживает перемещение, установите в его управляющем файле `relocatable = false`, и также задайте в параметре `schema` имя предполагаемой целевой схемы. Это предотвратит использование указания `SCHEMA` команды `CREATE EXTENSION`, если только оно задаёт не то же имя, что определено в управляющем файле. Этот выбор обычно необходим, если в расширении делаются внутренние предположения об имени схемы расшире-

ния, которое нельзя свести к использованию псевдоимени `@extschema@`. Механизм подстановки `@extschema@` будет работать и в этом случае, хотя польза от него будет ограниченной, так как имя схемы определяется управляющим файлом.

В любом случае при выполнении файла скрипта параметр search_path изначально будет указывать на целевую схему; то есть, `CREATE EXTENSION` делает то же, что и:

```
SET LOCAL search_path TO @extschema@, pg_temp;
```

Это позволяет направить объекты, создаваемые скриптом, в целевую схему. Скрипт может изменить `search_path`, если пожелает, но обычно это нежелательно. Параметр `search_path` восстанавливает предыдущее значение по завершении `CREATE EXTENSION`.

Целевая схема определяется параметром `schema` (если он задан) в управляющем файле, либо указанием `SCHEMA` команды `CREATE EXTENSION` (если оно присутствует), а в противном случае выбирается текущая схема для создания объектов по умолчанию (первая указанная в параметре `search_path` вызывающего). Когда используется параметр управляющего файла `schema`, целевая схема будет создана, если она ещё не существует, но в двух других случаях она должна уже существовать.

Если в параметре `requires` в управляющем файле расширения указаны какие-либо расширения, необходимые для данного, их целевые схемы добавляются к начальному значению `search_path` после целевой схемы нового расширения. Благодаря этому их объекты видны для скрипта нового расширения.

В целях безопасности схема `pg_temp` всегда автоматически добавляется в конец `search_path`.

Хотя неперемещаемое расширение может содержать объекты, распределяемые по нескольким схемам, обычно желательно поместить все объекты, предназначенные для внешнего использования, в одну схему, назначенную целевой схемой расширения. Такой порядок будет хорошо согласовываться со значением `search_path` по умолчанию в процессе создания зависимых расширений.

Если расширение ссылается на объекты, принадлежащие другому расширению, рекомендуется дополнить эти ссылки именами схем. Для этого напишите `@extschema:`*`имя`*`@` в файле скрипта расширения, где *`имя`* — это имя другого расширения (которое должно быть в списке `requires` этого расширения). Эта строка будет заменена именем (при необходимости заключённым в двойные кавычки) целевой схемы этого расширения. Хотя такая запись избавляет от необходимости делать запрограммированные предположения об именах схем в файле скрипта расширения, при её использовании имя схемы другого расширения может встраиваться в установленные объекты. (Обычно это происходит, когда `@extschema:`*`имя`*`@` используется внутри строкового литерала, такого как тело функции или параметр `search_path`. В других случаях ссылка на объект сокращается до OID во время синтаксического анализа и не требует последующего поиска.) Если имя схемы расширения встраивается таким образом, следует запретить перемещение такого расширения, добавив его имя в список `no_relocate` после установки пользовательского расширения.

36.17.3. Конфигурационные таблицы расширений

Некоторые расширения включают конфигурационные таблицы, содержащие данные, которые могут быть добавлены или изменены пользователем после установки расширения. Обычно, если таблица является частью расширения, ни определение таблицы, ни её содержимое не будет выгружаться утилитой pg_dump. Но это поведение нежелательно для конфигурационных таблиц — изменения, внесённые в них пользователем, должны выгружаться; в противном случае расширение будет вести себя по-другому, когда будет загружено вновь.

Чтобы решить эту проблему, скрипт расширения может пометить созданную им таблицу или последовательность как конфигурационное отношение, в результате чего pg_dump включит в выгружаемые данные содержимое (но не определение) этой таблицы или последовательности. Для этого нужно вызвать функцию `pg_extension_config_dump(regclass, text)` после создания таблицы или последовательности, например так:

```
CREATE TABLE my_config (key text, value text);
CREATE SEQUENCE my_config_seq;
```

```
SELECT pg_catalog.pg_extension_config_dump('my_config', '');
SELECT pg_catalog.pg_extension_config_dump('my_config_seq', '');
```

Так можно пометить любое число таблиц или последовательностей, в том числе последовательности, связанные со столбцами `serial` или `bigserial`.

Когда второй аргумент `pg_extension_config_dump` — пустая строка, pg_dump выгружает всё содержимое таблицы. Обычно это правильно, только если после создания скриптом расширения эта таблица изначально пуста. Если же в таблице оказывается смесь начальных данных и данных, добавленных пользователем, во втором аргументе `pg_extension_config_dump` передаётся условие `WHERE`, которое отфильтровывает данные, подлежащие выгрузке. Например, имея таблицу, созданную таким образом:

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

можно сделать так, чтобы поле `standard_entry` содержало true только для строк, создаваемых скриптом расширения.

Для последовательностей второй аргумент функции `pg_extension_config_dump` не имеет значения.

В более сложных ситуациях, когда пользователи могут модифицировать и изначально существовавшие строки, можно создать триггеры для конфигурационной таблицы, которые корректно пометят изменённые строки.

Условие фильтра, связанное с конфигурационной таблицей, можно изменить, повторно вызвав `pg_extension_config_dump`. (Обычно это находит применение в скрипте обновления расширения.) Единственный способ обозначить, что некоторая таблица более не является конфигурационной — разорвать её связь с расширением командой `ALTER EXTENSION ... DROP TABLE`.

Заметьте, что отношения внешних ключей между таблицами определяют порядок, в котором эти таблицы будет выгружать pg_dump. В частности, pg_dump попытается выгрузить сначала основную таблицу, а затем подчинённую. Так как отношения внешних ключей устанавливаются во время выполнения CREATE EXTENSION (до загрузки данных в таблицы), циклические зависимости не поддерживаются. Когда образуются циклические зависимости, данные тем не менее будут выгружены, но полученный архив нельзя будет восстановить обычным образом, потребуется вмешательство пользователя.

Чтобы сохранить состояние последовательностей, связанных со столбцами `serial` или `bigserial`, их нужно помечать непосредственно. Для этой цели недостаточно пометить только их родительское отношение.

36.17.4. Обновление расширений

Один из плюсов механизма расширений заключается в том, что он предоставляет удобные способы управления обновлениями SQL-команд, определяющих объекты расширения. В частности, каждой выпускаемой версии установочного скрипта расширения назначается имя или номер версии. Кроме того, если вы хотите, чтобы пользователи могли динамически обновлять одну версию расширения до другой, вы должны предоставить *скрипты обновления*, которые внесут необходимые изменения для перехода от старой версии к новой. Скриптам обновлений назначаются имена, соответствующие шаблону `*расширение*--*старая_версия*--*новая_версия*.sql` (например, `foo--1.0--1.1.sql` будет содержать команды, меняющие версию `1.0` расширения `foo` на версию `1.1`).

С условием, что имеется подходящий скрипт расширения, команда `ALTER EXTENSION UPDATE` обновит установленное расширение до указанной новой версии. Скрипт обновления запускается в том же окружении, которое организует команда `CREATE EXTENSION` для установочных скриптов: в частности, `search_path` устанавливается таким же образом, а любые новые объекты, созданные скриптом, автоматически добавляются в расширение. И если скрипт решит удалить объекты-члены расширения, они будут автоматически исключены из его состава.

Если у расширения есть дополнительные управляющие файлы, для скрипта обновления применяются те параметры, которые связаны с целевой (новой) версией скрипта.

`ALTER EXTENSION` также может выполнять последовательности скриптов обновления для получения запрошенной версии. Например, если имеются только скрипты `foo--1.0--1.1.sql` и `foo--1.1--2.0.sql`, `ALTER EXTENSION` будет применять их по порядку, если при установленной версии `1.0` запрошено обновлении до версии `2.0`.

Postgres Pro не делает никаких предположений о свойствах имён версий: например, он не знает, следует ли версия `1.1` за `1.0`. Он просто сопоставляет имена имеющихся версий и следует пути, который требует применить как можно меньше скриптов обновлений. (Именем версии на самом деле может быть любая строка, которая не содержит `--` и при этом не начинается и не заканчивается символом `-`.)

Иногда бывают полезны скрипты «понижения версии», например, `foo--1.1--1.0.sql`, которые позволяют откатить изменения, связанные с версией `1.1`. Если вы применяете их, учтите, что есть вероятность неожиданного выполнения такого скрипта, если он окажется в кратчайшем пути. Рискованная ситуация возникает при наличии скрипта обновления по «короткому пути», который перепрыгивает через несколько версий, и скрипта понижения версии до начальной точки первого скрипта. В результате может получиться так, что понижение версии с последующим обновлением по короткому пути окажется на несколько шагов короче, чем последовательное повышение версии. Если скрипт понижения версии удаляет какие-либо незаменимые объекты, это может привести к нежелательным результатам.

Чтобы убедиться, что при обновлении не будет выбран нежелательный путь, воспользуйтесь этой командой:

```
SELECT * FROM pg_extension_update_paths('имя_расширения');
```

Она показывает каждую пару различных известных имён версий для указанного расширения, вместе с последовательностью обновления, которая будет выбрана для перехода от одной версии к другой, либо `NULL`, если путь обновления не находится. Путь выводится в текстовом виде с разделителями `--`. Если вы предпочитаете формат массива, вы можете применить `regexp_split_to_array(path,'--')`.

36.17.5. Установка расширений скриптами обновления

Расширение, существующее некоторое время, вероятно, будет иметь несколько версий, для которых автору надо будет писать скрипты обновления. Например, если вы выпустили расширение `foo` версий `1.0`, `1.1` и `1.2`, у вас должны быть скрипты обновления `foo--1.0--1.1.sql` и `foo--1.1--1.2.sql`. До Postgres Pro версии 10 необходимо было также создавать файлы скриптов `foo--1.1.sql` и `foo--1.2.sql`, которые устанавливают непосредственно новые версии скриптов; в противном случае их можно было установить, только установив `1.0` и произведя обновление. Это было утомительно и неэффективно, но теперь такой необходимости нет, так как команда `CREATE EXTENSION` может сама построить цепочку обновлений. Например, если имеются только файлы скриптов `foo--1.0.sql`, `foo--1.0--1.1.sql` и `foo--1.1--1.2.sql`, то запрос на установку версии `1.2` удовлетворяется запуском этих трёх скриптов по очереди. Это не будет отличаться от установки версии `1.0` с последующим обновлением до `1.2`. (Как и с командой `ALTER EXTENSION UPDATE`, при наличии нескольких путей выбирается самый короткий.) Организация скриптов расширения по такой схеме может упростить сопровождение небольших обновлений.

Если вы используете дополнительные (ориентированные на версию) управляющие файлы для расширения, поддерживаемого по такой схеме, имейте в виду, что управляющий файл нужен для каждой версии, даже если для неё нет отдельного скрипта установки, так как этот файл будет определять, как произвести неявное обновление до этой версии. Например, если в файле `foo--1.0.control` задаётся `requires = 'bar'`, а в других управляющих файлах `foo` — нет, зависимость расширения от `bar` будет удалена при обновлении с версии `1.0` до другой.

36.17.6. Замечания о безопасности расширений

Широко распространяемые расширения не должны строить никаких предположений относительно базы данных, в которой они находятся. Таким образом, функции, предоставляемые расширениями, следует писать в безопасном стиле, так, чтобы их нельзя было скомпрометировать в атаках с использованием пути поиска.

Расширение, у которого свойство `superuser` имеет значение true, должно быть также защищено от угроз безопасности, связанных с действиями, которые выполняются при установке и обновлении расширения. Для злонамеренного пользователя не составит большого труда создать объект типа троянского коня, который впоследствии скомпрометирует выполнение неаккуратно написанного скрипта расширения и позволит этому пользователю стать суперпользователем.

Если расширение имеет характеристику `trusted`, вызывающий пользователь может выбрать схему, в которую оно будет устанавливаться. При этом он может намеренно выбрать небезопасную схему в надежде получить таким образом права суперпользователя. Поэтому доверенное расширение крайне уязвимо с точки зрения безопасности, так что все содержащиеся в его скриптах команды необходимо тщательно проверять, чтобы исключить возможность его компрометации.

Советы по безопасному написанию функций представлены ниже в Подразделе 36.17.6.1, а советы по написанию установочных скриптов — в Подразделе 36.17.6.2.

36.17.6.1. Замечания о безопасности функций в расширениях

Функции, реализованные в расширениях на языках SQL и PL*, подвержены атакам с использованием пути поиска во время выполнения, так как синтаксический разбор этих функций имеет место, когда они выполняются, а не когда создаются.

На странице `CREATE FUNCTION` даётся полезный совет по безопасному написанию функций с характеристикой `SECURITY DEFINER`. Эти приёмы рекомендуется применять и для функций, предоставляемых расширениями, так как подобная функция может вызываться пользователем с расширенными правами.

Если вы не можете оставить в `search_path` только безопасные схемы, считайте, что каждое заданное без схемы имя может быть разрешено в объект, созданный злонамеренным пользователем. Избегайте конструкций, явно зависящих от `search_path`; например, `IN` и `CASE` *выражение* `WHEN` всегда выбирают оператор по пути поиска. Вместо них следует использовать конструкции `OPERATOR(`*схема*`.=) ANY` и `CASE WHEN` *выражение*.

Расширения общего назначения не должны рассчитывать на то, что они устанавливаются в безопасную схему, что означает, что даже ссылаться на собственные объекты с указанием схемы в них небезопасно. Например, если в расширении определена функция `myschema.myfunc(bigint)`, её вызов в виде `myschema.myfunc(42)` можно перехватить, создав специальную функцию `myschema.myfunc(integer)`. Позаботьтесь о том, чтобы типы параметров функций и операторов в точности соответствовали объявленным типам их аргументов, и используйте явные приведения, где это необходимо.

36.17.6.2. Замечания о безопасности скриптов расширений

Скрипт установки или обновления расширения следует защищать от атак, осуществляемых во время выполнения с использованием пути поиска. Если имя объекта, который должен использоваться в скрипте по замыслу автора, может быть разрешено в какой-либо другой объект, компрометация расширения произойдёт сразу либо позже, когда объект расширения будет использоваться.

Команды DDL, например `CREATE FUNCTION` и `CREATE OPERATOR CLASS`, в целом безопасны, но будьте бдительны в отношении команд, в которых фигурируют произвольные запросы и выражения. Например, требуют проверки команды `CREATE VIEW`, а также выражения `DEFAULT` в `CREATE FUNCTION`.

Иногда в скрипте расширения возникает потребность выполнить произвольный SQL, например, чтобы внести в каталог изменения, невозможные через DDL. В этом случае обязательно выполняйте такие команды с безопасным `search_path`; *не* доверяйте пути, установленному при выполнении `CREATE/ALTER EXTENSION`. Для этого рекомендуется временно сменить `search_path` на `pg_catalog, pg_temp` и добавить явные указания схемы, в которую устанавливается расширение, везде, где это требуется. (Этот приём также может быть полезен при создании представлений.)

Ссылки на другие расширения крайне сложно полностью обезопасить, отчасти из-за отсутствия понимания, в какой схеме находится другое расширения. Риски уменьшаются, если оба расширения устанавливаются в одну схему, так как в этом случае зловредный объект не может оказаться перед объектами в схеме целевого расширения при используемом во время установки `search_path`. Однако в настоящее время нет механизма, который бы это требовал. Поэтому на данный момент рекомендуется не помечать расширение как доверенное, если оно зависит от других, не считая тех, что уже установлены в схему `pg_catalog`.

36.17.7. Пример расширения

Здесь представлен полный пример расширения, в котором средствами исключительно SQL реализуется составной тип с двумя элементами, который может сохранить в своих слотах значения любого типа, названные «k» и «v». Для хранения все значения переводятся в текстовый формат (если они имеют другой формат).

Файл скрипта `pair--1.0.sql` выглядит так:

```
-- complain if script is sourced in psql, rather than via CREATE EXTENSION
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, FUNCTION = pair);

-- "SET search_path" is easy to get right, but qualified names perform better.
CREATE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'
SET search_path = pg_temp;

CREATE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(pg_catalog.||) $2.k,
               $1.v OPERATOR(pg_catalog.||) $2.v)::@extschema@.pair;';
```

Управляющий файл `pair.control` выглядит так:

```
# расширение pair
comment = 'Тип данных для пары ключ/значение'
default_version = '1.0'
# расширение не может быть перемещаемым, так как использует @extschema@
relocatable = false
```

Хотя вам вряд ли понадобится сборочный файл, только для того, чтобы установить эти два файла в нужный каталог, вы можете использовать `Makefile` следующего содержания:

```
EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
```

```
include $(PGXS)
```

Этот Makefile опирается на инфраструктуру PGXS, которая описывается в Разделе 36.18. С ним команда `make install` установит управляющий файл и скрипт в правильный каталог, который определит pg_config.

Когда эти файлы будут установлены, выполните команду `CREATE EXTENSION`, чтобы загрузить объекты в определённую базу данных.

36.18. Инфраструктура сборки расширений

Если вы задумываетесь о распространении ваших модулей расширения Postgres Pro, знайте, что организовать для них портируемую систему сборки может быть довольно сложно. Поэтому инсталляция Postgres Pro включает инфраструктуру сборки расширений, названную PGXS, так что несложные модули расширений можно собрать просто в среде установленного сервера. PGXS предназначена в первую очередь для расширений, написанных на C, хотя её можно применять и для расширения на чистом SQL. Заметьте, что PGXS не претендует на роль универсальной инфраструктуры сборки, способной собрать любой программный объект, взаимодействующий с Postgres Pro; она просто автоматизирует общие правила для сборки простых модулей расширения сервера. Для более сложных пакетов вам придётся разработать собственную систему сборки.

Чтобы использовать инфраструктуру PGXS для вашего расширения, вы должны написать простой сборочный файл. В нём вы должны установить нужные переменные и подключить глобальный сборочный файл PGXS. Следующий пример собирает модуль расширения с именем `isbn_issn`, который включает разделяемую библиотеку, написанную на C, управляющий файл расширения, SQL-скрипт, текстовый файл документации и заголовочный файл (он нужен, только если другие модули будут вызывать функции данного расширения напрямую, без использования SQL):

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn
HEADERS_isbn_issn = isbn_issn.h

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Последние три строки всегда должны быть такими. Выше в файле вы определяете переменные или добавляете собственные правила для make.

Установите одну из этих трёх переменных, чтобы указать, что будет собрано:

`MODULES`

> список объектов разделяемых библиотек, которые должны быть собраны из исходных файлов с одной основой (суффиксы библиотек в этом списке не указываются)

`MODULE_big`

> разделяемая библиотека, которая должна быть собрана из нескольких исходных файлов (объектные файлы перечисляются в `OBJS`)

`PROGRAM`

> исполняемая программа, которая должна быть собрана (объектные файлы перечисляются в `OBJS`)

Также можно установить следующие переменные:

`EXTENSION`

> имена расширений(я); для каждого имени вы должны предоставить файл *`расширение`*`.control`, который будет установлен в *`префикс`*`/share/extension`

`MODULEDIR`

подкаталог в каталоге *префикс*`/share`, в который должны устанавливаться файлы DATA и DOCS (если не задан, подразумевается `extension`, если установлена переменная `EXTENSION`, или `contrib` в противном случае)

`DATA`

произвольные файлы, которые должны быть установлены в *префикс*`/share/$MODULEDIR`

`DATA_built`

произвольные файлы, которые должны быть сначала собраны, а затем установлены в *префикс*`/share/$MODULEDIR`

`DATA_TSEARCH`

произвольные файлы, которые должны быть установлены в *префикс*`/share/tsearch_data`

`DOCS`

произвольные файлы, которые должны быть установлены в *префикс*`/doc/$MODULEDIR`

`HEADERS`
`HEADERS_built`

Файлы, которые будут устанавливаться (и, возможно, собираться) в *префикс*`/include/server/$MODULEDIR/$MODULE_big`.

В отличие от файлов `DATA_built`, файлы в `HEADERS_built` не удаляются при выполнении цели `clean`; если вы хотите удалять их, добавьте их в `EXTRA_CLEAN` или напишите собственные правила для этого.

`HEADERS_$MODULE`
`HEADERS_built_$MODULE`

Файлы, которые будут устанавливаться (если требуется, после сборки) в *префикс*`/include/server/$MODULEDIR/$MODULE`, где в качестве `$MODULE` должно задаваться имя модуля, фигурирующее в `MODULES` или `MODULE_big`.

В отличие от файлов `DATA_built`, файлы в `HEADERS_built_$MODULE` не удаляются при выполнении цели `clean`; если вы хотите удалять их, добавьте их в `EXTRA_CLEAN` или напишите собственные правила для этого.

Для одного модуля вполне можно использовать обе переменные в любом сочетании, если только в вашем списке `MODULES` не присутствуют два имени, отличающиеся только префиксом `built_`, что приводит к неоднозначности. В этом (очень маловероятном) случае следует использовать только переменные `HEADERS_built_$MODULE`.

`SCRIPTS`

скрипты (не двоичные файлы), которые должны быть установлены в *префикс*`/bin`

`SCRIPTS_built`

скрипты (не двоичные файлы), которые должны быть сначала собраны, а затем установлены в *префикс*`/bin`

`REGRESS`

список тестов регрессий (без суффикса), см. ниже

`REGRESS_OPTS`

дополнительные параметры, передаваемые pg_regress

`ISOLATION`

список изоляционных тестов, см. ниже

`ISOLATION_OPTS`

дополнительные параметры, передаваемые pg_isolation_regress

`TAP_TESTS`

переключатель, определяющий, нужно ли запускать TAP-тесты; см. ниже

`NO_INSTALL`

не определять цель `install`; это полезно для модулей тестирования, результаты сборки которых не нужно устанавливать

`NO_INSTALLCHECK`

не определять цель `installcheck`; это полезно, если тестам требуется особая конфигурация или pg_regress не используется

`EXTRA_CLEAN`

дополнительные файлы, которые должны быть удалены при `make clean`

`PG_CPPFLAGS`

флаги, добавляемые перед другими в `CPPFLAGS`

`PG_CFLAGS`

флаги, добавляемые в `CFLAGS`

`PG_CXXFLAGS`

флаги, добавляемые в `CXXFLAGS`

`PG_LDFLAGS`

флаги, добавляемые перед другими в `LDFLAGS`

`PG_LIBS`

будет добавлено в строку компоновки `PROGRAM`

`SHLIB_LINK`

будет добавлено в строку компоновки `MODULE_big`

`PG_CONFIG`

путь к программе pg_config в инсталляции Postgres Pro, с которой будет выполняться сборка (обычно указывается просто `pg_config`, и используется первый экземпляр, найденный по пути в `PATH`)

Поместите этот сборочный файл под именем `Makefile` в каталог, где находится ваше расширение. После этого выполните `make`, чтобы скомпилировать, а затем `make install`, чтобы установить ваш модуль. По умолчанию расширение компилируется и устанавливается для той инсталляции Postgres Pro, которая соответствует экземпляру `pg_config`, найденному первым при поиске по пути в `PATH`. Чтобы использовать другую инсталляцию, вы можете задать в `PG_CONFIG` путь к её экземпляру `pg_config` либо внутри сборочного файла, либо в командном файле `make`.

Вы также можете запустить `make` в каталоге вне каталога исходного дерева вашего расширения, если хотите отделить каталог сборки. Эта процедура называется сборкой с *VPATH* и выполняется так:

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
make -f /path/to/extension/source/tree/Makefile install
```

Также вы можете подготовить каталог для сборки с VPATH таким же образом, как это делается в коде ядра сервера. Как один из вариантов, для этого можно воспользоваться скриптом ядра `config/prep_buildtree`. Затем вы сможете выполнить сборку, установив переменную `VPATH` для `make` таким образом:

```
make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install
```

Эта процедура поддерживает самые разные расположения каталогов.

Скрипты, перечисленные в переменной `REGRESS`, используются для регрессионного тестирования модуля, и вызвать их можно командой `make installcheck` после `make install`. Для проведения тестирования необходим работающий сервер Postgres Pro. Файлы скриптов, перечисленные в `REGRESS`, должны размещаться в подкаталоге `sql/` каталога расширения. Эти файлы должны иметь расширение `.sql`, но указывать его в списке `REGRESS` в сборочном файле не нужно. Для каждого теста также должен создаваться файл с ожидаемым выводом в подкаталоге `expected/`, с тем же базовым именем и расширением `.out`. Команда `make installcheck` выполнит каждый тест в psql и сравнит полученный вывод с ожидаемым. Все выявленные различия будут записаны в файл `regression.diffs` в формате команды `diff -c`. Заметьте, что при попытке запустить тест без файла ожидаемого вывода этот тест будет отмечен как «проблемный», поэтому убедитесь, что все такие файлы присутствуют.

Скрипты, перечисленные в переменной `ISOLATION`, используются для тестирования работы модуля при параллельной нагрузке, и вызвать их можно командой `make installcheck` после `make install`. Для проведения тестирования необходим работающий сервер Postgres Pro. Файлы скриптов, перечисленные в `ISOLATION`, должны размещаться в подкаталоге `specs/` каталога вашего расширения. Эти файлы должны иметь расширение `.spec`, которое не должно включаться в список `ISOLATION` в Makefile. Для каждого теста также должен создаваться файл с ожидаемым выводом в подкаталоге `expected/`, с тем же базовым именем и расширением `.out`. Команда `make installcheck` выполнит каждый тест и сравнит полученный вывод с ожидаемым. Все выявленные различия будут записаны в файл `output_iso/regression.diffs` в формате команды `diff -c`. Заметьте, что при попытке запустить тест без файла ожидаемого вывода этот тест будет отмечен как «проблемный», поэтому убедитесь, что все такие файлы присутствуют.

Переменная `TAP_TESTS` включает использование TAP-тестов. Данные, получаемые на каждом прогоне теста, помещаются в подкаталог `tmp_check/`.

Подсказка

Проще всего для этого создать пустые файлы ожидаемого вывода, а затем выполнить тест (при этом, конечно, будут выявлены несоответствия). Изучите полученные файлы результатов, сохранённые в каталоге `results/` (для тестов `REGRESS`) или каталоге `output_iso/results/` (для тестов в `ISOLATION`), и, если они соответствуют вашим ожиданиям от теста, скопируйте их в `expected/`.

Глава 37. Триггеры

В этой главе содержится общая информация о разработке триггерных функций. Триггерные функции могут быть написаны на большинстве доступных процедурных языков, включая PL/pgSQL (Глава 41), PL/Tcl (Глава 42), PL/Perl (Глава 43) и PL/Python (Глава 44). После прочтения этого раздела, следует обратиться к разделу, посвящённому любимому процедурному языку, чтобы узнать специфические для него детали разработки триггеров.

Триггерные функции можно писать и на C, хотя большинство людей находит, что проще использовать один из процедурных языков. В настоящее время невозможно написать триггерную функцию на чистом SQL.

37.1. Обзор механизма работы триггеров

Триггер является указанием, что база данных должна автоматически выполнить заданную функцию, всякий раз когда выполнен определённый тип операции. Триггеры можно использовать с таблицами (секционированными и обычными), с представлениями и с внешними таблицами.

Для обычных и сторонних таблиц можно определять триггеры, которые будут срабатывать до или после любой из команд `INSERT`, `UPDATE` или `DELETE`; либо один раз для каждой модифицируемой строки, либо один раз для оператора SQL. Триггеры на `UPDATE` можно установить так, чтобы они срабатывали, только когда в предложении `SET` оператора `UPDATE` упоминаются определённые столбцы. Также триггеры могут срабатывать для операторов `TRUNCATE`. Если происходит событие триггера, для обработки этого события в установленный момент времени вызывается функция триггера.

Для представлений триггеры могут быть определены для выполнения вместо операций `INSERT`, `UPDATE` и `DELETE`. Такие триггеры `INSTEAD OF` вызываются единожды для каждой строки, которая должна быть изменена в этом представлении. Именно функция триггера отвечает за то, чтобы произвести необходимые изменения в нижележащих базовых таблицах представления и должным образом возвращать изменённые строки, чтобы они появлялись в представлении. Триггеры для представлений тоже могут быть определены так, что они будут выполняться единожды для всего оператора SQL, до или после операций `INSERT`, `UPDATE` или `DELETE`. Однако такие триггеры срабатывают, только если для представления определён триггер `INSTEAD OF`. В противном случае все операторы, обращающиеся к представлению, должны быть переписаны в виде операторов, обращающихся к нижележащим базовым таблицам, и тогда будут срабатывать триггеры, установленные для этих таблиц.

Триггерная функция должна быть создана до триггера. Она должна быть объявлена без аргументов и возвращать тип `trigger`. (Триггерная функция получает данные на вход посредством специально переданной структуры `TriggerData`, а не в форме обычных аргументов.)

После создания триггерной функции создаётся триггер с помощью CREATE TRIGGER. Одна и та же триггерная функция может быть использована для нескольких триггеров.

Postgres Pro предлагает как *построчные*, так и *операторные* триггеры. В случае построчного триггера триггерная функция вызывается один раз для каждой строки, затронутой оператором, запустившим триггер. Операторный же триггер, напротив, вызывается только один раз при выполнении соответствующего оператора, независимо от количества строк, которые он затрагивает. В частности оператор, который не затрагивает никаких строк, всё равно приведёт к срабатыванию операторного триггера. Эти два типа триггеров также называют триггерами *уровня строк* и триггерами *уровня оператора*, соответственно. Триггеры на `TRUNCATE` могут быть определены только на уровне оператора, а не на уровне строк.

Триггеры также классифицируются в соответствии с тем, срабатывают ли они до, после или вместо операции. Они называются триггерами `BEFORE`, `AFTER` и `INSTEAD OF`, соответственно. Триггеры `BEFORE` уровня оператора срабатывают до того, как оператор начинает делать что-либо, тогда как триггеры `AFTER` уровня оператора срабатывают в самом конце работы оператора. Эти типы триггеров могут быть определены для таблиц, представлений или сторонних таблиц. Триггеры `BEFORE` уровня строки срабатывают непосредственно перед обработкой конкретной строки, в то время как триггеры `AFTER` уровня строки срабатывают в конце работы всего оператора (но до любого из триггеров `AFTER` уровня оператора). Эти типы триггеров могут определяться только для таблиц, в том числе сторонних, но не для представлений. Триггеры `INSTEAD OF` могут определяться только для представлений и только на уровне строк: они срабатывают для каждой строки сразу после того как строка представления идентифицирована как подлежащая обработке.

Если триггер `AFTER` — это *триггер ограничения*, его выполнение может быть отложено не до конца работы оператора, а до конца транзакции. В любом случае триггер выполняется в рамках той же транзакции, к которой относится вызвавший его оператор, поэтому если или оператор, или триггер вызывает ошибку, оба действия отменяются.

Если команда `INSERT` содержит предложение `ON CONFLICT DO UPDATE`, могут срабатывать триггеры уровня строк `BEFORE INSERT` и затем `BEFORE UPDATE`. Такие операции могут быть сложными, если триггеры не идемпотентны, поскольку изменения, выполненные триггером `BEFORE INSERT`, будут видны при выполнении триггера `BEFORE UPDATE`, в том числе изменения столбцов `EXCLUDED`.

Заметьте, что триггеры `UPDATE` уровня оператора вызываются при `ON CONFLICT DO UPDATE` независимо от того, будут ли изменены какие-либо строки в результате `UPDATE` (и даже в случае, когда альтернативный путь `UPDATE` вообще не выбирается). При выполнении запроса `INSERT` с предложением `ON CONFLICT DO UPDATE` сначала выполняются триггеры `BEFORE INSERT`, затем триггеры `BEFORE UPDATE`, потом триггеры `AFTER UPDATE` и, наконец, `AFTER INSERT` (речь идёт о триггерах на уровне операторов).

Оператор, нацеленный на родительскую таблицу в иерархии наследования или секционирования, не вызывает срабатывания триггеров уровня оператора для задействованных дочерних таблиц; срабатывать будут только такие триггеры для родительской таблицы. Однако если для этих дочерних таблиц установлены триггеры уровня строк, они будут срабатывать.

Если оператор `UPDATE` в секционированной таблице должен переместить строку в другую секцию, это перемещение реализуется в результате выполнения `DELETE` в исходной секции и последующего `INSERT` в новой секции. При этом в исходной секции срабатывают все триггеры `BEFORE UPDATE` и `BEFORE DELETE` уровня строк. Затем в целевой секции срабатывают все триггеры `BEFORE INSERT` уровня строк. Следует иметь в виду, что в случаях, когда все эти триггеры модифицируют перемещаемую строку, полученный результат может быть неожиданным. Если рассматривать триггеры `AFTER ROW`, то применяться будут триггеры `AFTER DELETE` и `AFTER INSERT`, но не триггеры `AFTER UPDATE`, так как команда `UPDATE` заменяется на `DELETE` и `INSERT`. Если же рассматривать триггеры уровня операторов, ни триггеры `DELETE`, ни триггеры `INSERT` не будут срабатывать, даже если производится перемещение строк; сработают только триггеры `UPDATE`, установленные в целевой таблице оператора `UPDATE`.

Для команды `MERGE` нет отдельных триггеров. Вместо этого срабатывают триггеры команд `UPDATE`, `DELETE` и `INSERT` в зависимости от того, какие действия указаны в запросе `MERGE` (триггеры уровня оператора) и какие действия выполняются (триггеры уровня строк).

При выполнении команды `MERGE` срабатывают триггеры уровня оператора `BEFORE` и `AFTER` при событиях, указанных в действиях команды `MERGE`, независимо от того, было ли действие в итоге выполнено. Это не отличается от поведения команды `UPDATE`, для которой также срабатывают триггеры уровня оператора, когда строки не меняются. Триггеры уровня строк срабатывают только при фактическом добавлении, изменении или удалении строки. Таким образом, абсолютно нормально, когда для определённых действий срабатывают триггеры уровня оператора и при этом не срабатывают триггеры уровня строк.

Триггерные функции, вызываемые триггерами операторов, должны всегда возвращать `NULL`. Триггерные функции, вызываемые триггерами строк, могут вернуть строку таблицы (значение типа `HeapTuple`). У триггера уровня строки, срабатывающего до операции, есть следующий выбор:

- Можно вернуть `NULL`, чтобы пропустить операцию для текущей строки. Это указывает исполнителю запросов, что не нужно выполнять операцию со строкой вызвавшей триггер (вставку, изменение или удаление конкретной строки в таблице).

- Возвращаемая строка для триггеров `INSERT` или `UPDATE` будет именно той, которая будет вставлена или обновлена в таблице. Это позволяет триггерной функции изменять вставляемую или обновляемую строку.

Если в триггере `BEFORE` уровня строки не планируется использовать любой из этих вариантов, то нужно аккуратно вернуть в качестве результата ту же строку, которая была передана на вход (то есть строку `NEW` для триггеров `INSERT` и `UPDATE`, или строку `OLD` для триггеров `DELETE`).

Триггер уровня строки `INSTEAD OF` должен вернуть либо `NULL`, чтобы указать, что он не модифицирует базовые таблицы представления, либо он должен вернуть строку представления, полученную на входе (строку `NEW` для операций `INSERT` и `UPDATE` или строку `OLD` для операций `DELETE`). Отличное от `NULL` возвращаемое значение сигнализирует, что триггер выполнил необходимые изменения данных в представлении. Это приведёт к увеличению счётчика количества строк, затронутых командой. Для операций `INSERT` и `UPDATE` (и только для них) триггер может изменить строку `NEW` перед тем как её вернуть. В результате будут изменены данные, возвращаемые `INSERT RETURNING` или `UPDATE RETURNING`, что полезно, когда представление должно возвращать не те данные, что были получены.

Возвращаемое значение игнорируется для триггеров уровня строки, вызываемых после операции, поэтому они могут возвращать `NULL`.

Генерируемые столбцы заслуживают отдельного внимания. Сохраняемые генерируемые столбцы вычисляются после триггеров `BEFORE` и перед триггерами `AFTER`. Таким образом, в триггерах `AFTER` можно наблюдать сгенерированное значение. В триггерах `BEFORE` строка `OLD`, как можно было ожидать, содержит предыдущее значение, однако в строке `NEW` ещё не содержится новое сгенерированное значение, и обращаться к нему не следует. На уровне языка C содержимое столбца в этот момент считается неопределённым; более высокоуровневые языки должны блокировать обращения к сохраняемому генерируемому столбцу в строке `NEW` внутри триггера `BEFORE`. Изменённые в триггере `BEFORE` значения генерируемого столбца игнорируются и будут перезаписаны.

Если есть несколько триггеров на одно и то же событие для одной и той же таблицы, то они будут вызываться в алфавитном порядке по имени триггера. Для триггеров `BEFORE` и `INSTEAD OF` потенциально изменённая строка, возвращаемая одним триггером, становится входящей строкой для следующего триггера. Если любой из триггеров `BEFORE` или `INSTEAD OF` возвращает `NULL`, операция для этой строки прекращается и последующие триггеры (для этой строки) не срабатывают.

В определении триггера можно указать логическое условие `WHEN`, которое будет проверяться, чтобы посмотреть, нужно ли запускать триггер. В триггерах уровня строки в условии `WHEN` можно проверять старые и/или новые значения столбцов строки. (В триггерах уровня оператора также можно использовать условие `WHEN`, хотя в этом случае это не так полезно.) В триггерах `BEFORE` условие `WHEN` вычисляется непосредственно перед тем, как триггерная функция будет выполнена, поэтому использование `WHEN` существенно не отличается от выполнения той же проверки в самом начале триггерной функции. Однако в триггерах `AFTER` условие `WHEN` вычисляется сразу после обновления строки и от этого зависит, будет ли поставлено в очередь событие запуска триггера в конце оператора или нет. Поэтому, когда условие `WHEN` в триггере `AFTER` не возвращает истину, не требуется ни постановка события в очередь, ни повторная выборка этой строки в конце оператора. Это может существенно ускорить работу операторов, изменяющих большое количество строк, с триггером, который должен сработать только для нескольких. В триггерах `INSTEAD OF` не поддерживается использование условий `WHEN`.

Как правило, триггеры `BEFORE` уровня строки используются для проверки или модификации данных, которые будут вставлены или изменены. Например, триггер `BEFORE` можно использовать для вставки текущего времени в столбец `timestamp` или проверки, что два элемента строки согласованы между собой. Триггеры `AFTER` уровня строки наиболее разумно использовать для каскадного обновления данных в других таблицах или проверки согласованности сделанных изменений с данными в других таблицах. Причина для такого разделения работы в том, что триггер `AFTER` видит окончательное значение строки, в то время как для триггера `BEFORE` это не так, ведь могут быть другие триггеры `BEFORE`, которые сработают позже. Если нет особых причин для выбора между триггерами `BEFORE` или `AFTER`, то триггер `BEFORE` предпочтительнее, так как не требует сохранения информации об операции до конца работы оператора.

Если триггерная функция выполняет команды SQL, эти команды могут заново запускать триггеры. Это известно как каскадные триггеры. Прямых ограничений на количество каскадных уровней не существует. Вполне возможно, что каскадные вызовы приведут к рекурсивному срабатыванию одного и того же триггера. Например, в триггере `INSERT` может выполняться команда, которая добавляет строку в эту же таблицу, тем самым опять вызывая триггер на `INSERT`. Обязанность программиста не допускать бесконечную рекурсию в таких случаях.

Если в ограничении внешнего ключа указаны ссылочные действия (а именно каскадное изменение или удаление данных), они выполняются обычными SQL-командами `UPDATE` или `DELETE` в отношении целевой таблицы. В частности, на эти изменения сработают все существующие для целевой таблицы триггеры. Если от такого триггера меняется или блокируется действие этих команд, это в конечном счёте может привести к нарушению ссылочной целостности. При разработке триггеров следует этого избегать.

При определении триггера можно указывать аргументы. Цель включения аргументов в определение триггера в том, чтобы позволить разным триггерам с аналогичными требованиями вызывать одну и ту же функцию. В качестве примера можно создать обобщенную триггерную функцию, которая принимает два аргумента с именами столбцов и записывает текущего пользователя в первый аргумент и текущий штамп времени во второй. При правильном написании такая триггерная функция будет независима от конкретной таблицы, для которой она будет запускаться. Таким образом, одна и та же функция может использоваться при выполнении `INSERT` в любую таблицу с соответствующими столбцами, чтобы, например, автоматически отслеживать создание записей в транзакционной таблице. Для триггеров `UPDATE` аргументы также могут использоваться для отслеживания последних сделанных изменений.

У каждого языка программирования, поддерживающего триггеры, есть свой собственный метод доступа из триггерной функции к входным данным триггера. Входные данные триггера включают в себя тип события (например, `INSERT` или `UPDATE`), а также любые аргументы, перечисленные в `CREATE TRIGGER`. Для триггеров уровня строки входные данные также включают строку `NEW` для триггеров `INSERT` и `UPDATE` и/или строку `OLD` для триггеров `UPDATE` и `DELETE`.

Триггеры уровня оператора по умолчанию не имеют возможностей для проверки отдельных строк, модифицированных оператором. Но триггер `AFTER STATEMENT` может запросить создание для него *переходных таблиц*, чтобы ему были доступны наборы затрагиваемых операцией строк. Триггерам `AFTER ROW` также могут предоставляться переходные таблицы, чтобы они могли видеть все изменения в таблице, а не только изменения в отдельных строках, для которых они срабатывают. Метод обращения к переходным таблицам определяется применяемым языком программирования, но обычно переходные таблицы представляются как временные таблицы только для чтения, к которым в триггерной функции можно обращаться, выполняя SQL-команды.

37.2. Видимость изменений в данных

Если в триггерной функции выполняются SQL-команды и эти команды обращаются к таблице, на которую создан триггер, то необходимо знать правила видимости данных, потому что они определяют, будут ли видеть эти SQL-команды изменения в данных, для которых сработал триггер. Кратко:

- Триггеры уровня оператора следуют простым правилам видимости: никакие из изменений, произведённых оператором, не видны в триггерах `BEFORE`, тогда как в триггерах `AFTER` видны все изменения.
- Изменение данных (вставка, обновление или удаление), заставляющее сработать триггер, *не видно* для команд SQL, выполняемых в триггере `BEFORE` уровня строки, потому что это изменение ещё не произошло.
- Тем не менее команды SQL, выполняемые в триггере `BEFORE` уровня строки, *будут* видеть изменения данных в строках, которые уже были обработаны в этом операторе. Это требует осто-

рожности, так как порядок обработки строк в целом непредсказуемый; команда SQL, обрабатывающая множество строк, может делать это в любом порядке.

- Аналогично, триггер `INSTEAD OF` уровня строки увидит изменения данных, внесённые при предыдущих вызовах триггера `INSTEAD OF` для этой же внешней команды.
- Когда срабатывает триггер `AFTER` уровня строки, все изменения сделанные оператором уже выполнены и видны в вызываемой триггерной функции.

Если триггерная функция написана на одном из стандартных процедурных языков, вышеприведённые утверждения применимы, только если функция объявлена как `VOLATILE`. Функции объявленные как `STABLE` или `IMMUTABLE` в любом случае не будут видеть изменений, сделанных вызывающим оператором.

Дополнительную информацию о правилах видимости данных можно найти в Разделе 45.5. Пример в Разделе 37.4 содержит демонстрацию этих правил.

37.3. Триггерные функции на языке C

В этом разделе описываются низкоуровневые детали интерфейса для триггерных функций. Эта информация необходима только при разработке триггерных функций на языке C. При использовании языка более высокого уровня эти детали обрабатываются не видны. В большинстве случаев стоит рассмотреть возможность использования процедурного языка, прежде чем начать разрабатывать триггеры на C. В документации по каждому процедурному языку объясняется, как создавать триггеры на этом языке.

Триггерные функции должны использовать интерфейс функций «версии 1».

Когда функция вызывается диспетчером триггеров, ей не передаются обычные аргументы, но передаётся указатель «context», ссылающийся на структуру `TriggerData`. Функции на C могут проверить, вызваны ли они диспетчером триггеров или нет, выполнив макрос:

```
CALLED_AS_TRIGGER(fcinfo)
```

который разворачивается в:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Если возвращается истина, то `fcinfo->context` можно безопасно привести к типу `TriggerData *` и использовать указатель на структуру `TriggerData`. Функция *не* должна изменять структуру `TriggerData` или любые данные, которые на неё указывают.

`struct TriggerData` определяется в `commands/trigger.h`:

```
typedef struct TriggerData
{
    NodeTag          type;
    TriggerEvent     tg_event;
    Relation         tg_relation;
    HeapTuple        tg_trigtuple;
    HeapTuple        tg_newtuple;
    Trigger         *tg_trigger;
    TupleTableSlot  *tg_trigslot;
    TupleTableSlot  *tg_newslot;
    Tuplestorestate *tg_oldtable;
    Tuplestorestate *tg_newtable;
    const Bitmapset *tg_updatedcols;
} TriggerData;
```

где элементы определяются следующим образом:

`type`

Всегда `T_TriggerData`.

`tg_event`

Описывает событие, для которого вызывается функция. Можно использовать следующие макросы для получения информации о `tg_event`:

`TRIGGER_FIRED_BEFORE(tg_event)`

Возвращает истину, если триггер сработал до операции.

`TRIGGER_FIRED_AFTER(tg_event)`

Возвращает истину, если триггер сработал после операции.

`TRIGGER_FIRED_INSTEAD(tg_event)`

Возвращает истину, если триггер сработал вместо операции.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Возвращает истину, если триггер сработал на уровне строки.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Возвращает истину, если триггер сработал на уровне оператора.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Возвращает истину, если триггер сработал для операции `INSERT`.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Возвращает истину, если триггер сработал для операции `UPDATE`.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Возвращает истину, если триггер сработал для операции `DELETE`.

`TRIGGER_FIRED_BY_TRUNCATE(tg_event)`

Возвращает истину, если триггер сработал для операции `TRUNCATE`.

`tg_relation`

Указатель на структуру, описывающую таблицу, для которой сработал триггер. Подробнее об этой структуре в `utils/rel.h`. Самое интересное здесь это `tg_relation->rd_att` (дескриптор записей таблицы) и `tg_relation->rd_rel->relname` (имя таблицы; имеет тип `NameData`, а не `char*`; используйте `SPI_getrelname(tg_relation)`, чтобы получить тип `char*` если потребуется копия имени).

`tg_trigtuple`

Указатель на строку, для которой сработал триггер. Это строка, которая вставляется, обновляется или удаляется. При срабатывании триггера для `INSERT` или `DELETE` это значение нужно вернуть из функции, только если не планируется изменять строку (в случае `INSERT`) или пропускать операцию для этой строки.

`tg_newtuple`

Для триггера на `UPDATE` это указатель на новую версию строки либо `NULL`, если триггер на `INSERT` или `DELETE`. Это значение нужно вернуть из функции в случае `UPDATE`, если не планируется изменять строку или пропускать операцию для этой строки.

`tg_trigger`

Указатель на структуру с типом `Trigger`, определённую в `utils/reltrigger.h`:

```
typedef struct Trigger
```

```
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    bool        tgisclone;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
    char       *tgoldtable;
    char       *tgnewtable;
} Trigger;
```

где `tgname` — имя триггера, `tgnargs` — количество аргументов в `tgargs`, и `tgargs` — массив указателей на аргументы, указанные в команде `CREATE TRIGGER`. Остальные члены структуры предназначены для внутреннего использования.

`tg_trigslot`

Слот, содержащий `tg_trigtuple`, или указатель `NULL`, если такой строки нет.

`tg_newslot`

Слот, содержащий `tg_newtuple`, или указатель `NULL`, если такой строки нет.

`tg_oldtable`

Указатель на структуру типа `Tuplestorestate`, содержащую ноль или несколько строк в формате, определяемом содержимым `tg_relation`, или указатель `NULL`, если переходное отношение `OLD TABLE` отсутствует.

`tg_newtable`

Указатель на структуру типа `Tuplestorestate`, содержащую ноль или несколько строк в формате, определяемом содержимым `tg_relation`, или указатель `NULL`, если переходное отношение `NEW TABLE` отсутствует.

`tg_updatedcols`

Для триггеров `UPDATE` — битовая карта, в которой отмечается, какие столбцы изменила команда, вызвавшая срабатывание триггера. Используя её, универсальные триггерные функции могут оптимизировать свои действия, не обращая внимания на столбцы, которые не были изменены.

Определить, вошёл ли в битовую карту столбец с атрибутом под номером `attnum` (считая с 1), можно так: `bms_is_member(attnum - FirstLowInvalidHeapAttributeNumber, trigdata->tg_updatedcols))`.

Для всех остальных триггеров содержит `NULL`.

Чтобы обращаться к переходным таблицам в запросах, выполняемых через SPI, используйте SPI_register_trigger_data.

Триггерная функция должна возвращать указатель `HeapTuple` или указатель `NULL` (но *не* SQL значение `null`, то есть не нужно устанавливать *`isNull`* в истину). Не забудьте, что если не планируете менять обрабатываемую триггером строку, то нужно вернуть либо `tg_trigtuple`, либо `tg_newtuple`.

37.4. Полный пример триггера

Вот очень простой пример триггерной функции, написанной на C. (Примеры триггеров для процедурных языков могут быть найдены в документации на процедурные языки.)

Функция `trigf` сообщает количество строк в таблице `ttest` и пропускает операцию для строки при попытке вставить пустое значение в столбец `x`. (Таким образом, триггер действует как ограничение `NOT NULL`, но не прерывает транзакцию.)

Вначале определение таблицы:

```
CREATE TABLE ttest (
    x integer
);
```

Теперь исходный код триггерной функции:

```
#include "postgres.h"
#include "fmgr.h"
#include "executor/spi.h"       /* this is what you need to work with SPI */
#include "commands/trigger.h"   /* ... triggers ... */
#include "utils/rel.h"          /* ... and relations */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
```

```
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

После компиляции исходного кода (см. Подраздел 36.10.5) объявляем функцию и триггеры:

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'имя_файла'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();
```

Теперь можно проверить работу триггера:

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Вставка записи пропущена (значение NULL), поэтому триггер AFTER не сработал

=> SELECT * FROM ttest;
 x
---
```

```
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                       ^^^^^^^^
                             вспомните, что говорилось о видимости
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                       ^^^^^^
                             вспомните, что говорилось о видимости
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                       ^^^^^^
                             вспомните, что говорилось о видимости
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

Глава 38. Триггеры событий

В дополнение к триггерам, рассмотренным в Главе 37, Postgres Pro также предоставляет триггеры событий. В отличие от обычных триггеров, которые подключаются к конкретной таблице и работают только с командами DML, триггеры событий определяются на уровне базы данных и работают с командами DDL.

Как и обычные триггеры, триггеры событий можно создавать на любом процедурном языке, поддерживающим триггеры событий, а также на C, но не на чистом SQL.

38.1. Обзор механизма работы триггеров событий

Триггер события срабатывает всякий раз, когда в базе данных, в которой он определён, происходит связанное с ним событие. В настоящий момент поддерживаются следующие события: `login`, `ddl_command_start`, `ddl_command_end`, `table_rewrite` и `sql_drop`. Поддержка дополнительных событий может быть добавлена в будущих выпусках.

Событие `login` происходит, когда аутентифицированный пользователь входит в систему. Любые ошибки в процедуре триггера для этого события могут помешать успешному входу в систему. Таких ошибок можно избежать, задав для параметра event_triggers значение `false` в строке подключения или файле конфигурации. Кроме того, можно перезапустить сервер в однопользовательском режиме (поскольку триггеры событий в этом режиме отключены). За дополнительными сведениями об однопользовательском режиме обратитесь к справке по postgres. Событийный триггер `login` также срабатывает на резервных серверах. Во избежание проблем с подключением к серверам, работающие на них триггеры `login` не должны ничего записывать в базу данных. Кроме того, рекомендуется избегать длительных запросов в событийных триггерах `login`. Обратите внимание, что отмена подключения в psql не отменит текущий триггер `login`.

Событие `ddl_command_start` происходит непосредственно перед выполнением команд `CREATE`, `ALTER`, `DROP`, `SECURITY LABEL`, `COMMENT`, `GRANT` и `REVOKE`. Проверка на существование объекта перед срабатыванием триггера не производится. В качестве исключения, однако, это событие не происходит для команд DDL, обращающихся к общим объектам кластера базы данных — базам данных, табличным пространствам, ролям, а также к самим триггерам событий. Событие `ddl_command_start` также происходит непосредственно перед выполнением команды `SELECT INTO`, так как она равнозначна команде `CREATE TABLE AS`.

Событие `ddl_command_end` происходит непосредственно после выполнения команд из того же набора. Чтобы получить дополнительную информацию об операциях DDL, повлёкших произошедшее событие, вызовите функцию `pg_event_trigger_ddl_commands()`, возвращающую множество, из кода обработчика события `ddl_command_end` (см. Раздел 9.30). Заметьте, что этот триггер срабатывает после того, как эти действия имели место (но до фиксации транзакции), так что в системных каталогах можно увидеть уже изменённое состояние.

Событие `sql_drop` происходит непосредственно перед событием `ddl_command_end` для команд, которые удаляют объекты базы данных. Для получения списка удалённых объектов используйте возвращающую набор строк функцию `pg_event_trigger_dropped_objects()` в триггере события `sql_drop` (см. Раздел 9.30). Обратите внимание, что триггер выполняется после удаления объектов из таблиц системного каталога, поэтому их невозможно больше увидеть.

Событие `table_rewrite` происходит непосредственно перед тем, как таблица будет перезаписана в результате определённых действий команд `ALTER TABLE` и `ALTER TYPE`. Хотя перезапись таблицы может быть вызвана и другими управляющими операторами, в частности `CLUSTER` и `VACUUM`, событие `table_rewrite` для них не вызывается. Найти OID таблицы, которая была перезаписана, можно с помощью функции `pg_event_trigger_table_rewrite_oid()` (см. Раздел 9.30). Причину (причины) перезаписи можно посмотреть с помощью функции `pg_event_trigger_table_rewrite_reason()`.

Триггеры событий (как и прочие функции) не могут выполняться в прерванной транзакции. Поэтому, если команда DDL завершается ошибкой, соответствующие триггеры `ddl_command_end` не сработают. И наоборот, если триггер `ddl_command_end` завершился с ошибкой, последующие триггеры событий не сработают, так же как и сама команда не будет выполняться. Похожим образом, если триггер `ddl_command_end` завершится ошибкой, действие команды DDL будет отменено, как это происходит при возникновении ошибки внутри транзакции.

Полный список команд, которые поддерживаются триггерами событий, можно найти в Разделе 38.2.

Для создания триггера события используется команда CREATE EVENT TRIGGER. Предварительно нужно создать функцию, со специальным возвращаемым типом `event_trigger`. Данная функция не обязана возвращать значение (и может не возвращать). Возвращаемый тип служит лишь указанием на то, что функция будет вызываться из триггера события.

Если есть несколько триггеров на одно и то же событие, то они будут вызываться в алфавитном порядке по имени триггера.

В определении триггера можно использовать условие `WHEN`, чтобы, например, триггер `ddl_command_start` срабатывал только для отдельных команд, которые нужно перехватить. Триггеры событий часто используются для ограничения диапазона DDL-команд, доступных пользователям.

38.2. Матрица срабатывания триггеров событий

В Таблице 38.1 перечислены команды, для которых поддерживаются триггеры событий.

Таблица 38.1. Поддержка триггеров событий командами DDL

Тег команды	`ddl_command_start`	`ddl_command_end`	`sql_drop`	`table_rewrite`	**Примечания**
`ALTER AGGREGATE`	X	X	-	-	
`ALTER COLLATION`	X	X	-	-	
`ALTER CONVERSION`	X	X	-	-	
`ALTER DOMAIN`	X	X	-	-	
`ALTER DEFAULT PRIVILEGES`	X	X	-	-	
`ALTER EXTENSION`	X	X	-	-	
`ALTER FOREIGN DATA WRAPPER`	X	X	-	-	
`ALTER FOREIGN TABLE`	X	X	X	-	
`ALTER FUNCTION`	X	X	-	-	
`ALTER LANGUAGE`	X	X	-	-	
`ALTER LARGE OBJECT`	X	X	-	-	
`ALTER MATERIALIZED VIEW`	X	X	-	X	
`ALTER OPERATOR`	X	X	-	-	
`ALTER OPERATOR CLASS`	X	X	-	-	
`ALTER OPERATOR FAMILY`	X	X	-	-	
`ALTER POLICY`	X	X	-	-	
`ALTER PROCEDURE`	X	X	-	-	
`ALTER PUBLICATION`	X	X	-	-	
`ALTER ROUTINE`	X	X	-	-	
`ALTER SCHEMA`	X	X	-	-	
`ALTER SEQUENCE`	X	X	-	-	

Тег команды	`ddl_command_start`	`ddl_command_end`	`sql_drop`	`table_rewrite`	**Примечания**
`ALTER SERVER`	X	X	-	-	
`ALTER STATISTICS`	X	X	-	-	
`ALTER SUBSCRIPTION`	X	X	-	-	
`ALTER TABLE`	X	X	X	X	
`ALTER TEXT SEARCH CONFIGURATION`	X	X	-	-	
`ALTER TEXT SEARCH DICTIONARY`	X	X	-	-	
`ALTER TEXT SEARCH PARSER`	X	X	-	-	
`ALTER TEXT SEARCH TEMPLATE`	X	X	-	-	
`ALTER TRIGGER`	X	X	-	-	
`ALTER TYPE`	X	X	-	X	
`ALTER USER MAPPING`	X	X	-	-	
`ALTER VIEW`	X	X	-	-	
`COMMENT`	X	X	-	-	Только для локальных объектов
`CREATE ACCESS METHOD`	X	X	-	-	
`CREATE AGGREGATE`	X	X	-	-	
`CREATE CAST`	X	X	-	-	
`CREATE COLLATION`	X	X	-	-	
`CREATE CONVERSION`	X	X	-	-	
`CREATE DOMAIN`	X	X	-	-	
`CREATE EXTENSION`	X	X	-	-	
`CREATE FOREIGN DATA WRAPPER`	X	X	-	-	
`CREATE FOREIGN TABLE`	X	X	-	-	
`CREATE FUNCTION`	X	X	-	-	
`CREATE INDEX`	X	X	-	-	
`CREATE LANGUAGE`	X	X	-	-	
`CREATE MATERIALIZED VIEW`	X	X	-	-	
`CREATE OPERATOR`	X	X	-	-	
`CREATE OPERATOR CLASS`	X	X	-	-	
`CREATE OPERATOR FAMILY`	X	X	-	-	
`CREATE POLICY`	X	X	-	-	
`CREATE PROCEDURE`	X	X	-	-	
`CREATE PUBLICATION`	X	X	-	-	
`CREATE RULE`	X	X	-	-	
`CREATE SCHEMA`	X	X	-	-	

Тег команды	ddl_ command_ start	ddl_ command_ end	sql_drop	table_ rewrite	**Примечания**
CREATE SEQUENCE	X	X	-	-	
CREATE SERVER	X	X	-	-	
CREATE STATISTICS	X	X	-	-	
CREATE SUBSCRIPTION	X	X	-	-	
CREATE TABLE	X	X	-	-	
CREATE TABLE AS	X	X	-	-	
CREATE TEXT SEARCH CONFIGURATION	X	X	-	-	
CREATE TEXT SEARCH DICTIONARY	X	X	-	-	
CREATE TEXT SEARCH PARSER	X	X	-	-	
CREATE TEXT SEARCH TEMPLATE	X	X	-	-	
CREATE TRIGGER	X	X	-	-	
CREATE TYPE	X	X	-	-	
CREATE USER MAPPING	X	X	-	-	
CREATE VIEW	X	X	-	-	
DROP ACCESS METHOD	X	X	X	-	
DROP AGGREGATE	X	X	X	-	
DROP CAST	X	X	X	-	
DROP COLLATION	X	X	X	-	
DROP CONVERSION	X	X	X	-	
DROP DOMAIN	X	X	X	-	
DROP EXTENSION	X	X	X	-	
DROP FOREIGN DATA WRAPPER	X	X	X	-	
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP MATERIALIZED VIEW	X	X	X	-	
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP PROCEDURE	X	X	X	-	
DROP PUBLICATION	X	X	X	-	
DROP ROUTINE	X	X	X	-	

Тег команды	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	**Примечания**
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP STATISTICS	X	X	X	-	
DROP SUBSCRIPTION	X	X	X	-	
DROP TABLE	X	X	X	-	
DROP TEXT SEARCH CONFIGURATION	X	X	X	-	
DROP TEXT SEARCH DICTIONARY	X	X	X	-	
DROP TEXT SEARCH PARSER	X	X	X	-	
DROP TEXT SEARCH TEMPLATE	X	X	X	-	
DROP TRIGGER	X	X	X	-	
DROP TYPE	X	X	X	-	
DROP USER MAPPING	X	X	X	-	
DROP VIEW	X	X	X	-	
GRANT	X	X	-	-	Только для локальных объектов
IMPORT FOREIGN SCHEMA	X	X	-	-	
REFRESH MATERIALIZED VIEW	X	X	-	-	
REINDEX	X	X	-	-	
REVOKE	X	X	-	-	Только для локальных объектов
SECURITY LABEL	X	X	-	-	Только для локальных объектов
SELECT INTO	X	X	-	-	

38.3. Триггерные функции событий на языке C

В этом разделе описываются низкоуровневые детали интерфейса для событийных триггерных функций. Эта информация необходима только при разработке событийных триггерных функций событий на языке C. При использовании языка более высокого уровня эти детали не видны. В большинстве случаев стоит рассмотреть возможность использования процедурного языка, прежде чем начать разрабатывать событийные триггеры на C. В документации по каждому процедурному языку объясняется, как создавать событийные триггеры на этом языке.

Триггерные функции событий должны использовать «version 1» интерфейса диспетчера функций.

Когда функция вызывается диспетчером триггеров событий, ей не передаются обычные аргументы, но передаётся указатель «context», ссылающийся на структуру `EventTriggerData`. Функции на C могут проверить вызваны ли они диспетчером триггеров событий или нет выполнив макрос:

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

который разворачивается в:

```
EventTriggerData
```

Если возвращается истина, то `fcinfo->context` можно безопасно привести к типу `EventTriggerData *` и использовать указатель на структуру `EventTriggerData`. Функция *не* должна изменять структуру `EventTriggerData` или любые данные, которые на неё указывают.

`struct EventTriggerData` определена в `commands/event_trigger.h`:

```
typedef struct EventTriggerData
{
    NodeTag     type;
    const char *event;      /* имя события */
    Node       *parsetree;  /* дерево разбора */
    CommandTag  tag;        /* тег команды */
} EventTriggerData;
```

со следующими членами структуры:

`type`

Всегда `T_EventTriggerData`.

`event`

Описывает событие, для которого вызывается функция. Возможные значения: `"login"`, `"ddl_command_start"`, `"ddl_command_end"`, `"sql_drop"`, `"table_rewrite"`. Суть этих событий описывается в Разделе 38.1.

`parsetree`

Указатель на дерево разбора команды. Детали можно посмотреть в исходном коде Postgres Pro. Структура дерева разбора может быть изменена без предупреждений.

`tag`

Тег команды, для которой сработал триггер события. Например `"CREATE FUNCTION"`.

Функция триггера события должна возвращать указатель `NULL` (но *не* SQL значение null, то есть не нужно устанавливать *`isNull`* в истину).

38.4. Полный пример триггера события

Вот очень простой пример функции для триггера события, написанной на C. (Примеры триггеров для процедурных языков могут быть найдены в документации на процедурные языки.)

Функция `noddl` выдаёт ошибку при каждом вызове. Триггер с этой функцией определяется для события `ddl_command_start`. Это предотвращает работу любых DDL-команд (за исключением тех, о которых говорилось в Разделе 38.1).

Теперь исходный код триггерной функции:

```
#include "postgres.h"

#include "commands/event_trigger.h"
#include "fmgr.h"

PG_MODULE_MAGIC;
```

```
PG_FUNCTION_INFO_V1(noddl);

Datum
noddl(PG_FUNCTION_ARGS)
{
    EventTriggerData *trigdata;

    if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* internal error */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    ereport(ERROR,
            (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
             errmsg("command \"%s\" denied",
                    GetCommandTagName(trigdata->tag))));

    PG_RETURN_NULL();
}
```

После компиляции исходного кода (см. Подраздел 36.10.5) объявляем функцию и триггеры:

```
CREATE FUNCTION noddl() RETURNS event_trigger
    AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
    EXECUTE FUNCTION noddl();
```

Теперь проверим работу триггера:

```
=# \dy
                     List of event triggers
 Name  |       Event       | Owner | Enabled | Function | Tags
-------+-------------------+-------+---------+----------+------
 noddl | ddl_command_start | dim   | enabled | noddl    |
(1 row)

=# CREATE TABLE foo(id serial);
ОШИБКА:  command "CREATE TABLE" denied
```

В этой ситуации, для запуска DDL-команд, нужно либо удалить триггер события, либо отключить его. Может быть удобным отключить триггер на время выполнения транзакции:

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

(Вспомним, что триггеры событий не обрабатывают DDL-команды для самих триггеров событий.)

38.5. Пример событийного триггера, обрабатывающего перезапись таблицы

Благодаря существованию события `table_rewrite`, можно реализовать политику перезаписи таблиц, допускающую перезапись только в определённое время обслуживания.

Следующий пример демонстрирует реализацию такой политики.

```
CREATE OR REPLACE FUNCTION no_rewrite()
```

```
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
---
--- Реализация локальной политики перезаписи таблиц:
---   перезапись public.foo не допускается,
---   другие таблицы могут перезаписываться между 1 часом ночи и 6 часами утра,
---   если только их размер не превышает 100 блоков
---
DECLARE
  table_oid oid := pg_event_trigger_table_rewrite_oid();
  current_hour integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF pg_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
        RAISE EXCEPTION 'you''re not allowed to rewrite the table %',
                        table_oid::regclass;
  END IF;

  SELECT INTO pages relpages FROM pg_class WHERE oid = table_oid;
  IF pages > max_pages
  THEN
        RAISE EXCEPTION 'rewrites only allowed for table with less than % pages',
                        max_pages;
  END IF;

  IF current_hour NOT BETWEEN 1 AND 6
  THEN
        RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';
  END IF;
END;
$$;

CREATE EVENT TRIGGER no_rewrite_allowed
                  ON table_rewrite
   EXECUTE FUNCTION no_rewrite();
```

38.6. Пример событийного триггера, обрабатывающего вход в базу данных

Триггер события `login` может использоваться для регистрации входа пользователей в систему, для проверки подключения и назначения ролей в зависимости от обстоятельств или для инициализации данных сеанса. Очень важно, чтобы любой событийный триггер, использующий событие `login`, проверял, не находится ли база данных в состоянии восстановления перед выполнением каких-либо операций записи. Попытка выполнить запись на резервном сервере вызовет ошибку, что не позволит подключиться к нему.

Следующий пример демонстрирует эти возможности.

```
-- создание тестовых таблиц и ролей
CREATE TABLE user_login_log (
  "user" text,
  "session_start" timestamp with time zone
);
CREATE ROLE day_worker;
CREATE ROLE night_worker;
```

```
-- пример триггерной функции
CREATE OR REPLACE FUNCTION init_session()
  RETURNS event_trigger SECURITY DEFINER
  LANGUAGE plpgsql AS
$$
DECLARE
  hour integer = EXTRACT('hour' FROM current_time at time zone 'utc');
  rec boolean;
BEGIN
-- 1. Запрет на вход с 2 до 4 часов ночи
IF hour BETWEEN 2 AND 4 THEN
  RAISE EXCEPTION 'Login forbidden';
END IF;

-- Следующие проверки могут не выполняться на резервных серверах, поэтому
-- до выполнения каких-либо действий нужно проверить, не находится ли
-- база данных в режиме восстановления.
SELECT pg_is_in_recovery() INTO rec;
IF rec THEN
  RETURN;
END IF

-- 2. Назначение ролей. В дневное время назначается роль day_worker,
-- в ночное — night_worker
IF hour BETWEEN 8 AND 20 THEN
  EXECUTE 'REVOKE night_worker FROM ' || quote_ident(session_user);
  EXECUTE 'GRANT    day_worker TO '   || quote_ident(session_user);
ELSE
  EXECUTE 'REVOKE day_worker FROM ' || quote_ident(session_user);
  EXECUTE 'GRANT  night_worker TO ' || quote_ident(session_user);
END IF;

-- 3. Инициализация данных сеанса пользователя
CREATE TEMP TABLE session_storage (x float, y integer);
ALTER TABLE session_storage OWNER TO session_user;

-- 4. Протоколирование времени соединения
INSERT INTO user_login_log VALUES (session_user, current_timestamp);

END;
$$;

-- определение триггера
CREATE EVENT TRIGGER init_session
  ON login
  EXECUTE FUNCTION init_session();
ALTER EVENT TRIGGER init_session ENABLE ALWAYS;
```

Глава 39. Система правил

В этой главе обсуждается система правил, реализованная в Postgres Pro. Промышленные системы правил по сути довольно простые, но при их использовании приходится сталкиваться с множеством неочевидных вещей.

В некоторых других базах данных определяются активные правила баз данных, которые обычно реализуются в виде процедур и триггеров. Так же их можно реализовать и в Postgres Pro.

Система правил (точнее говоря, система правил перезаписи запросов) полностью отличается от механизма хранимых процедур и триггеров. Она изменяет запросы по заданным правилам, а затем передаёт модифицированный запрос планировщику для планирования и выполнения. Это очень мощное средство, подходящее для решения множества задач, например, для определения представлений и процедур на языке запросов или реализации версионности. Теоретические основы и преимущества этой системы правил также описаны в ston90b и ong90 (на английском языке).

39.1. Дерево запроса

Чтобы понять, как работает система правил, нужно знать, когда она вызывается, что принимает на вход и какой результат выдаёт.

Система правил внедрена между анализатором запросов и планировщиком. Она принимает разобранный запрос, одно дерево запроса, и определённые пользователем правила перезаписи, тоже представленные деревьями с некоторой дополнительной информацией, и создаёт некоторое количество деревьев запросов в результате. Таким образом, на входе и выходе этой системы оказывается то, что может сформировать анализатор запросов, и как следствие, всё, с чем работает эта система, представимо в виде операторов SQL.

Так что же такое дерево запроса? Это внутреннее представление оператора SQL, в котором все образующие его части хранятся отдельно. Эти деревья можно увидеть в журнале сервера, если установить параметры конфигурации `debug_print_parse`, `debug_print_rewritten` или `debug_print_plan`. Действия правил также хранятся в виде деревьев запросов, в системном каталоге `pg_rewrite`. Они не форматируются как при выводе в журнал, но содержат точно такую же информацию.

Для прочтения неформатированного дерева требуется некоторый навык. Но так как представления дерева запросов в виде SQL достаточно, чтобы понять систему правил, в этой главе не будет рассказываться, как их читать.

Читая SQL-представления деревьев запросов в этой главе, необходимо понимать, на какие части разбивается оператор, когда он преобразуется в структуру дерева запроса. Дерево запроса состоит из следующих частей:

тип команды

> Это простое значение, говорящее, какая команда (`SELECT`, `INSERT`, `UPDATE` или `DELETE`) сгенерировала дерево запросов.

список отношений

> Список отношений представляет собой массив отношений, используемых в запросе. В запросе `SELECT` он включает отношения, указанные после ключевого слова `FROM`.
>
> Каждый элемент списка отношений представляет таблицу или представление и говорит, с каким именем они упоминаются в других частях запроса. В дереве запросов записываются номера элементов списка отношений, а не их имена, поэтому для него неактуальна проблема дублирования имён, как для оператора SQL. Такая проблема может возникнуть при объединении списков отношений, образованных разными правилами. В этой главе данная ситуация рассматриваться не будет.

результирующее отношение

Индекс в списке отношений, указывающий на отношение, которое будет получать результаты запроса.

В запросах `SELECT` результирующее отношение отсутствует. (Особый случай `SELECT INTO` практически равнозначен `CREATE TABLE` с последующим `INSERT ... SELECT` и здесь отдельно не рассматривается.)

Для команд `INSERT`, `UPDATE` и `DELETE` результирующим отношением будет таблица (или представление!), в которой будут происходить изменения.

выходной список

Выходной список — это список выражений, определяющих результат запроса. В случае `SELECT`, это выражения, которые образуют окончательный набор выходных данных. Они соответствуют выражениям, записанным между ключевыми словами `SELECT` и `FROM`. (Указание `*` — это просто краткое обозначение имён всех столбцов отношения. Анализатор разворачивает его в список отдельных столбцов, так что система правил никогда не видит его.)

Командам `DELETE` не нужен обычный выходной список, так как они не выдают никакие результаты. Вместо этого планировщик добавляет в пустой выходной список специальную запись CTID, чтобы исполнитель мог найти удаляемую строку. (CTID добавляется, когда результирующее отношение — обычная таблица. Если это представление, планировщиком добавляется переменная, содержащая всю строку, как рассказывается в Подразделе 39.2.4.)

Для команд `INSERT` выходной список описывает новые строки, которые должны попасть в результирующее отношение. Он включает выражения в предложении `VALUES` или предложении `SELECT` в `INSERT ... SELECT`. На первом этапе процесс перезаписи добавляет элементы выходного списка для столбцов, которым ничего не присвоила исходная команда, но имеющих значения по умолчанию. Все остальные столбцы (без заданного значения и значения по умолчанию) планировщик заполняет константой NULL.

Для команд `UPDATE` выходной список описывает новые строки, которые должны заменить старые. В системе правил он содержит только выражения из части `SET столбец = выражение`. Для пропущенных столбцов планировщик вставляет выражения, копирующие значения из старой строки в новую. Так же, как и с командой `DELETE`, при этом добавляется CTID или переменная со всей строкой, чтобы исполнитель мог найти изменяемую старую строку.

Каждая запись в выходном списке содержит выражение, которое может быть константой, переменной, указывающей на столбец отношения в таблице отношений, параметром или деревом выражений, образованным из констант, переменных, операторов, вызовов функций и т. д.

условие фильтра

Условие фильтра запроса — это выражение, во многом похожее на те, что содержатся в выходном списке. Результат этого выражения — логический, он говорит, должна ли выполняться операция (`INSERT`, `UPDATE`, `DELETE` или `SELECT`) для данной строки в результате. Оно соответствует предложению `WHERE` SQL-оператора.

дерево соединения

Дерево соединения запроса показывает структуру предложения `FROM`. Для простых запросов вида `SELECT ... FROM a, b, c`, дерево соединения — это просто список элементов `FROM`, так как они могут соединяться в любом порядке. Но с выражениями `JOIN`, особенно с внешними соединениями, приходится соединять отношения именно в заданном порядке. В этом случае дерево соединения отражает структуру выражений `JOIN`. Ограничения, связанные с конкретными предложениями `JOIN` (из выражений `ON` или `USING`), тоже сохраняются в виде условных выражений, добавленных к соответствующим узлам дерева соединения. Как оказалось, выражение `WHERE` верхнего уровня тоже удобно хранить как условие, добавленное к элементу верхнего уровня дерева соединения. Поэтому в дереве соединения на самом деле представляются оба предложения оператора `SELECT` — `FROM` и `WHERE`.

другие

Другие части дерева запроса, например, предложение `ORDER BY`, в данном контексте не представляют интереса. Система правил выполняет в них некоторые подстановки, применяя правила, но это не имеет непосредственного отношения к основам системы правил.

39.2. Система правил и представления

Представления в Postgres Pro реализованы на основе системы правил. По сути представление — это пустая таблица (фактически без данных) с правилом `ON SELECT DO INSTEAD`. Традиционно это правило называется `_RETURN`. Поэтому такое представление, как

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

, — это практически то же, что и

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

, однако такие параметры в запросе указывать нельзя, поскольку для таблиц не может быть правила `ON SELECT`.

Кроме того, для представления могут быть другие типы правил `DO INSTEAD`, которые позволяют выполнять команды `INSERT`, `UPDATE` и `DELETE` с представлением, несмотря на нехватку нижележащего хранилища. Эта тема освещается более подробно в Подразделе 39.2.4.

39.2.1. Как работают правила `SELECT`

Правила `ON SELECT` применяются ко всем запросам на последнем этапе, даже если это команда `INSERT`, `UPDATE` или `DELETE`. Эти правила отличаются от правил других видов тем, что они модифицируют непосредственно дерево запросов, а не создают новое. Поэтому мы начнём описание с правил `SELECT`.

В настоящее время возможно только одно действие в правиле `ON SELECT` и это должно быть безусловное действие `SELECT`, выполняемое в режиме `INSTEAD`. Это ограничение было введено, чтобы сделать правила достаточно безопасными для применения обычными пользователями, так что действие правил `ON SELECT` сводится к реализации представлений.

В примерах этой главы рассматриваются два представления с соединением, которые выполняют некоторые вычисления, и которые, в свою очередь, используются другими представлениями. Первое из этих двух представлений затем модифицируется, к нему добавляются правила для операций `INSERT`, `UPDATE` и `DELETE`, так что в итоге получается представление, которое работает как обычная таблица с некоторыми необычными функциями. Это не самый простой пример для начала, поэтому понять некоторые вещи будет сложнее. Но лучше иметь один пример, поэтапно охватывающий все обсуждаемые здесь темы, чем несколько различных, при восприятии которых в итоге может возникнуть путаница.

Таблицы, которые понадобятся нам для описания системы правил, выглядят так:

```
CREATE TABLE shoe_data (
    shoename   text,          -- первичный ключ
    sh_avail   integer,       -- число имеющихся пар
    slcolor    text,          -- предпочитаемый цвет шнурков
    slminlen   real,          -- минимальная длина шнурков
    slmaxlen   real,          -- максимальная длина шнурков
    slunit     text           -- единица длины
```

```
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- первичный ключ
    sl_avail   integer,       -- число имеющихся пар
    sl_color   text,          -- цвет шнурков
    sl_len     real,          -- длина шнурков
    sl_unit    text           -- единица длины
);

CREATE TABLE unit (
    un_name    text,          -- первичный ключ
    un_fact    real           -- коэффициент для перевода в см
);
```

Как можно догадаться, в них хранятся данные обувной фабрики.

Представления создаются так:

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           least(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

Команда `CREATE VIEW` для представления `shoelace` (самого простого из имеющихся) создаёт отношение `shoelace` и запись в `pg_rewrite` о правиле перезаписи, которое должно применяться, когда в запросе на выборку задействуется отношение `shoelace`. Для этого правила не задаются условия применения (о них рассказывается ниже, в описании правил не для `SELECT`, так как правила `SELECT` в настоящее бывают только безусловными) и оно действует в режиме `INSTEAD`. Заметьте, что условия применения отличаются от условий фильтра запроса, например, действие для нашего правила содержит условие фильтра. Действие правила выражается одним деревом запроса, которое является копией оператора `SELECT` в команде, создающей представление.

Примечание

Два дополнительных элемента списка отношений `NEW` и `OLD`, которые можно увидеть в соответствующей строке `pg_rewrite`, не представляют интереса для правил `SELECT`.

Сейчас мы наполним таблицы `unit` (единицы измерения), `shoe_data` (данные о туфлях) и `shoelace_data` (данные о шнурках) и выполним простой запрос к представлению:

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0, 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9, 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60, 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40, 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Это самый простой запрос `SELECT`, который можно выполнить с нашими представлениями, и мы воспользуемся этим, чтобы объяснить азы правил представлений. Запрос `SELECT * FROM shoelace` интерпретируется анализатором запросов и преобразуется в дерево запроса:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

Это дерево передаётся в систему правил, которая проходит по списку отношений и проверяет, есть ли какие-либо правила для этих отношений. Обрабатывая элемент отношения `shoelace` (сейчас он единственный), система правил находит правило `_RETURN` с деревом запроса:

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

Чтобы развернуть представление, механизм перезаписи просто формирует новый элемент для списка отношений — подзапрос, содержащий дерево действия правила, и подставляет этот элемент вместо исходного, на который ссылалось представление. Получившееся перезаписанное дерево запроса будет почти таким как дерево запроса:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

Однако есть одно различие: в списке отношений подзапроса будут содержаться два дополнительных элемента: `shoelace old` и `shoelace new`. Эти элементы не принимают непосредственного участия в запросе, так как они не задействованы в дереве соединения подзапроса и в целевом списке. Механизм перезаписи использует их для хранения информации о проверке прав доступа, которая изначально хранилась в элементе, указывающем на представление. Таким образом, исполнитель будет по-прежнему проверять, имеет ли пользователь необходимые права для доступа к представлению, хотя в перезаписанном запросе это представление не фигурирует непосредственно.

Так было применено первое правило. Система правил продолжит проверку оставшихся элементов списка отношений на верхнем уровне запроса (в данном случае таких элементов нет) и рекурсивно проверит элементы списка отношений в добавленном подзапросе, не ссылаются ли они на представления. (Но `old` и `new` разворачиваться не будут — иначе мы получили бы бесконечную рекурсию!) В этом примере для `shoelace_data` и `unit` нет правил перезаписи, так что перезапись завершается и результат, полученный выше, передаётся планировщику.

Сейчас мы хотим написать запрос, который выбирает туфли из имеющихся в данный момент, для которых есть подходящие шнурки (по цвету и длине) и число готовых пар больше или равно двум.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;

 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

На этот раз анализатор запроса выводит такое дерево:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

Первое правило применяется к представлению `shoe_ready` и в результате получается дерево запроса:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
```

```
                least(rsh.sh_avail, rsl.sl_avail) AS total_avail
           FROM shoe rsh, shoelace rsl
          WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

Подобным образом, правила для `shoe` и `shoelace` подставляются в список отношений, что даёт окончательное дерево запроса:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
                       s.sl_avail,
                       s.sl_color,
                       s.sl_len,
                       s.sl_unit,
                       s.sl_len * u.un_fact AS sl_len_cm
                  FROM shoelace_data s, unit u
                 WHERE s.sl_unit = u.un_name) rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

Это может показаться неэффективным, но планировщик преобразует этот запрос в одноуровневое дерево, «подтягивая» подзапросы в главный запрос, а затем планирует соединения так же, как и при явной записи с соединениями. Таким образом, упрощение дерева запросов является оптимизацией, которая производится независимо от перезаписи запросов.

39.2.2. Правила представлений не для SELECT

До этого в описании правил представлений не затрагивались два компонента дерева запросов — тип команды и результирующее отношение. На самом деле, тип команды не важен для правил представления, но результирующее отношение может повлиять на работу механизма перезаписи, потому что если это представление, требуются дополнительные операции.

Есть только несколько отличий между деревом запроса для `SELECT` и деревом для другой команды. Очевидно, у них различные типы команд, и для команды, отличной от `SELECT`, результирующее отношение указывает на элемент в списке отношений, куда должен попасть результат. Все остальные компоненты в точности те же. Поэтому, например, если взять таблицы `t1` и `t2` со столбцами `a` и `b`, деревья запросов для этих операторов:

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

будут практически одинаковыми. В частности:

- Списки отношений содержат элементы для таблиц `t1` и `t2`.
- Выходные списки содержат одну переменную, указывающую на столбец `b` элемента-отношения для таблицы `t2`.
- Выражения условий сравнивают столбцы `a` обоих элементов-отношений на равенство.
- Деревья соединений показывают простое соединение между `t1` и `t2`.

Как следствие, для обоих деревьев строятся похожие планы выполнения, с соединением двух таблиц. Для `UPDATE` планировщик добавляет в выходной список недостающие столбцы из `t1` и окончательное дерево становится таким:

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

В результате исполнитель, обрабатывающий соединение, выдаёт тот же результат, что и запрос:

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

Но с `UPDATE` есть маленькая проблема: часть плана исполнителя, в которой выполняется соединение, не представляет, для чего предназначены результаты соединения. Она просто выдаёт результирующий набор строк. Фактически есть одна команда `SELECT`, а другая, `UPDATE`, обрабатывается исполнителем выше, где он уже знает, что это команда `UPDATE` и что результат должен попасть в таблицу `t1`. Но какие из строк таблицы должны заменяться новыми?

Для решения этой проблемы в выходной список операторов `UPDATE` (и `DELETE`) добавляется ещё один элемент: идентификатор текущего кортежа (Current Tuple ID, CTID). Это системный столбец, содержащий номер блока в файле и позицию строки в блоке. Зная таблицу, по CTID можно получить исходную строку в `t1`, подлежащую изменению. С добавленным в выходной список CTID запрос фактически выглядит так:

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Теперь мы перейдём ещё к одной особенности Postgres Pro. Старые строки таблицы не переписываются, поэтому `ROLLBACK` выполняется быстро. С командой `UPDATE` в таблицу вставляется новая строка результата (без CTID) и в заголовке старой строки, на которую указывает CTID, в поля `cmax` и `xmax` записываются текущий счётчик команд и идентификатор текущей транзакции. Таким образом, старая строка оказывается скрытой и после фиксирования транзакции процесс очистки может окончательно удалить неактуальную версию строки.

Зная всё это, мы можем применять правила представлений абсолютно таким же образом к любой команде — никаких различий нет.

39.2.3. Преимущества представлений в Postgres Pro

Выше было показано, как система правил внедряет определения представлений в исходное дерево запроса. Во втором примере простой запрос `SELECT` к одному представлению создал окончательное дерево запроса, соединяющее 4 таблицы (таблица `unit` использовалась дважды с разными именами).

Преимущество реализации представлений через систему правил заключается в том, что планировщик получает в одном дереве запроса всю информацию о таблицах, которые нужно прочитать, о том, как связаны эти таблицы, об условиях в представлениях, а также об условиях, заданных в исходном запросе. И всё это имеет место, когда сам исходный запрос представляет собой соединение представлений. Планировщик должен выбрать лучший способ выполнения запроса, и чем больше информации он получит, тем лучше может быть его выбор. И то, как в Postgres Pro реализована система правил, гарантирует, что ему поступает вся информация, собранная о запросе на данный момент.

39.2.4. Изменение представления

Но что произойдёт, если записать имя представления в качестве целевого отношения команды `INSERT`, `UPDATE`, `DELETE` или `MERGE`? Если проделать подстановки, описанные выше, будет получено дерево запроса, в котором результирующее отношение указывает на элемент-подзапрос, что не будет работать. Однако Postgres Pro даёт ряд возможностей, чтобы сделать представления изменяемыми. Эти возможности перечислены в порядке увеличения сложности для пользователя: автоматически заменять нижележащие таблицы представления, выполнять пользовательский триггер или переписать запрос, используя механизм правил. Все варианты действий описаны ниже.

Если подзапрос выбирает данные из одного базового отношения и он достаточно прост, механизм перезаписи может автоматически заменить его нижележащим базовым отношением, чтобы команды `INSERT`, `UPDATE`, `DELETE` или `MERGE` обращались к базовому отношению. Представления, «достаточно простые» для этого, называются *автоматически изменяемыми*. Подробнее виды представлений, которые могут изменяться автоматически, описаны в CREATE VIEW.

Эту задачу также можно решить, создав триггер `INSTEAD OF` для представления (см. CREATE TRIGGER). В этом случае перезапись будет работать немного по-другому. Для `INSERT` механизм перезаписи не делает с представлением ничего, оставляя его результирующим отношением запроса. Для `UPDATE`, `DELETE` и `MERGE` ему по-прежнему придётся разворачивать запрос представления, чтобы получить «старые» строки, которые эта команда попытается изменить, удалить или объединить. Поэтому представление разворачивается как обычно, но в запрос добавляется ещё один элемент списка отношений, указывающий на представление в роли результирующего отношения.

При этом возникает проблема идентификации строк в представлении, подлежащих изменению. Вспомните, что когда результирующее отношение является таблицей, в выходной список добавляется специальное поле CTID, указывающее на физическое расположение изменяемых строк. Но это не будет работать, когда результирующее отношение — представление, так как в представлениях нет CTID, потому что их строки физически нигде не находятся. Вместо этого, для операций `UPDATE`, `DELETE` или `MERGE` в выходной список добавляется специальный элемент `wholerow` (вся строка), который разворачивается в содержимое всех столбцов представления. Используя этот элемент, исполнитель передаёт строку «old» в триггер `INSTEAD OF`. Какие именно строки должны изменяться фактически, будет решать сам триггер, исходя из полученных значений старых и новых строк.

Кроме того, пользователь может определить правила `INSTEAD`, в которых задать действия замены для команд `INSERT`, `UPDATE` и `DELETE` с представлением. Эти правила обычно преобразуют команду в другую команду, изменяющую одну или несколько таблиц, а не представление. Эта тема освещается в Разделе 39.4. Обратите внимание, что это не работает с командой `MERGE`, которая сейчас не поддерживает правила для целевого отношения, кроме правил `SELECT`.

Заметьте, что такие правила вычисляются сначала, перезаписывая исходный запрос до того, как он будет планироваться и выполняться. Поэтому, если для представления определены и триггеры `INSTEAD OF`, и правила для `INSERT`, `UPDATE` или `DELETE`, сначала вычисляются правила, а в зависимости от их действия, триггеры могут не вызываться вовсе.

Автоматическая перезапись запросов `INSERT`, `UPDATE`, `DELETE` или `MERGE` с простыми представлениями всегда производится в последнюю очередь. Таким образом, если у представления есть правила или триггеры, они переопределяют поведение автоматически изменяемых представлений.

Если для представления не определены правила `INSTEAD` или триггеры `INSTEAD OF`, и запрос не удаётся автоматически переписать в виде обращения к нижележащему базовому отношению, возникает ошибка, потому что исполнитель не сможет изменить такое представление.

39.3. Материализованные представления

Материализованные представления в Postgres Pro основаны на системе правил, как и представления, но их содержимое сохраняется как таблица. Основное отличие между:

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

и этой командой:

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

состоит в том, что материализованное представление впоследствии нельзя будет изменить непосредственно, а запрос, создающий материализованное представление, сохраняется точно так же, как запрос представления, и получить актуальные данные в материализованном представлении можно так:

```
REFRESH MATERIALIZED VIEW mymatview;
```

Информация о материализованном представлении в системных каталогах Postgres Pro ничем не отличается от информации о таблице или представлении. Поэтому для анализатора запроса материализованное представление является просто отношением, как таблица или представление. Когда запрос обращается к материализованному представлению, данные возвращаются непосредственно из него, как из таблицы; правило применяется, только чтобы его наполнить.

Хотя обращение к данным в материализованном представлении часто выполняется гораздо быстрее, чем обращение к нижележащим таблицам напрямую или через представление, данные в нём не всегда актуальные (но иногда это вполне приемлемо). Рассмотрим таблицу с данными продаж:

```
CREATE TABLE invoice (
    invoice_no    integer        PRIMARY KEY,
    seller_no     integer,       -- идентификатор продавца
    invoice_date  date,          -- дата продажи
    invoice_amt   numeric(13,2)  -- сумма продажи
);
```

Если пользователям нужно быстро обработать исторические данные, возможно их интересуют только общие показатели, а полнота данных на текущий момент не важна:

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT
      seller_no,
      invoice_date,
      sum(invoice_amt)::numeric(13,2) as sales_amt
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY
      seller_no,
      invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

Это материализованное представление может быть полезно для построения графика в информационной панели менеджеров по продажам. Для ежесуточного обновления статистики можно запланировать задание по расписанию, которое будет выполнять этот оператор:

```
REFRESH MATERIALIZED VIEW sales_summary;
```

Ещё одно применение материализованного представления — предоставить быстрый доступ к данным, получаемым с удалённой системы через обёртку сторонних данных. Ниже приведён простой пример с обёрткой `file_fdw`, с замерами времени, но так как при этом использовался кеш локальной системы, выигрыш в производительности при обращении к удалённой системе обычно будет гораздо больше, чем показано здесь. Заметьте, что мы также использовали возможность добавить индекс в материализованное представление, тогда как `file_fdw` индексы не поддерживает; при других видах доступа к сторонним данным такого преимущества может не быть.

Подготовка:

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
```

```
CREATE FOREIGN TABLE words (word text NOT NULL)
  SERVER local_file
  OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION pg_trgm;
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

Теперь давайте проверим написание слова. Сначала непосредственно через обёртку `file_fdw`:

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

 count
-------
     0
(1 row)
```

Выполнив `EXPLAIN ANALYZE`, мы получаем:

```
 Aggregate  (cost=21763.99..21764.00 rows=1 width=0) (actual time=188.180..188.181
 rows=1 loops=1)
   ->  Foreign Scan on words  (cost=0.00..21761.41 rows=1032 width=0) (actual
 time=188.177..188.177 rows=0 loops=1)
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 479829
         Foreign File: /usr/share/dict/words
         Foreign File Size: 4953699
 Planning time: 0.118 ms
 Execution time: 188.273 ms
```

Если же теперь обратиться к материализованному представлению, запрос выполнится гораздо быстрее:

```
 Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44 rows=1 width=0) (actual
 time=0.039..0.039 rows=0 loops=1)
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Planning time: 0.164 ms
 Execution time: 0.117 ms
```

В любом случае слово записано неправильно, поэтому давайте попробуем найти то, что имелось в виду. Сначала опять через `file_fdw` и `pg_trgm`:

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

     word
---------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
 Caterpillar's
 caterer
 caterer's
 caters
 catered
(10 rows)
```

```
 Limit  (cost=11583.61..11583.64 rows=10 width=32) (actual time=1431.591..1431.594
 rows=10 loops=1)
```

```
   ->  Sort  (cost=11583.61..11804.76 rows=88459 width=32) (actual
 time=1431.589..1431.591 rows=10 loops=1)
         Sort Key: ((word <-> 'caterpiler'::text))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on words  (cost=0.00..9672.05 rows=88459 width=32) (actual
 time=0.057..1286.455 rows=479829 loops=1)
               Foreign File: /usr/share/dict/words
               Foreign File Size: 4953699
 Planning time: 0.128 ms
 Execution time: 1431.679 ms
```

Затем через материализованное представление:

```
 Limit  (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10
 loops=1)
   ->  Index Scan using wrd_trgm on wrd  (cost=0.29..37020.87 rows=479829 width=10)
 (actual time=187.219..188.252 rows=10 loops=1)
         Order By: (word <-> 'caterpiler'::text)
 Planning time: 0.196 ms
 Execution time: 198.640 ms
```

Если периодическое обновление данных из другого источника в локальной базе данных вас устраивает, этот подход может дать значительный выигрыш в скорости.

39.4. Правила для `INSERT`, `UPDATE` и `DELETE`

Правила, определяемые для команд `INSERT`, `UPDATE` и `DELETE`, значительно отличаются от правил представлений, описанных в предыдущих разделах. Во-первых, команда `CREATE RULE` позволяет создавать правила со следующими особенностями:

- Они могут не определять действия.
- Они могут определять несколько действий.
- Они могут действовать в режиме `INSTEAD` или `ALSO` (по умолчанию).
- Становятся полезными псевдоотношения `NEW` и `OLD`.
- Они могут иметь условия применения.

Во-вторых, они не модифицируют само исходное дерево запроса. Вместо этого они создают несколько новых деревьев запросов и могут заменить исходное.

Внимание

Во многих случаях для задач, выполнимых с использованием правил для `INSERT`/`UPDATE`/`DELETE`, лучше применять триггеры. Оформляются триггеры чуть сложнее, но понять их смысл гораздо проще. К тому же с правилами могут быть получены неожиданные результаты, когда исходный запрос содержит изменчивые функции: в процессе исполнения правил эти функции могут вызываться большее число раз, чем ожидается.

Кроме того, в некоторых случаях эти типы правил вообще нельзя применять; а именно, с предложениями `WITH` в исходном запросе и с вложенными подзапросами `SELECT` с множественным присваиванием в списке `SET` запросов `UPDATE`. Это объясняется тем, что копирование этих конструкций в запрос правила привело бы к многократному вычислению вложенного запроса, что пошло бы в разрез с выраженными намерениями автора запроса.

39.4.1. Как работают правила для изменения

Запомните синтаксис:

```
CREATE [ OR REPLACE ] RULE имя AS ON событие
    TO таблица [ WHERE условие ]
```

```
    DO [ ALSO | INSTEAD ] { NOTHING | команда | ( команда ; команда ... ) }
```

В дальнейшем, под *правилами для изменения* подразумеваются правила, определяемые для команд `INSERT`, `UPDATE` или `DELETE`.

Правила для изменения применяются системой правил, когда результирующее отношение и тип команды в дереве запроса совпадает с объектом и событием, заданным в команде `CREATE RULE`. Для такого правила система правил создаёт список деревьев запросов. Изначально этот список пуст. С правилом может быть связано ноль (ключевое слово `NOTHING`), одно или несколько действий. Простоты ради мы рассмотрим правило с одним действием. Правило может иметь, а может не иметь условия применения, и действует в режиме `INSTEAD` или `ALSO` (по умолчанию).

Что такое условие применения правила? Это условие, которое говорит, когда нужно, а когда не нужно применять действия правила. В этом условии можно обращаться к псевдоотношениям `NEW` и/или `OLD`, которые представляют целевое отношение (но с особым значением).

Всего есть три варианта формирования деревьев запросов для правила с одним действием.

Без условия применения в режиме `ALSO` или `INSTEAD`

дерево запроса из действия правила с добавленным условием исходного дерева

С условием применения в режиме `ALSO`

дерево запроса из действия правила с условием применения правила и условием, добавленным из исходного дерева

С условием применения в режиме `INSTEAD`

дерево запроса из действия правила с условием применения правила и условием из исходного дерева; также добавляется исходное дерево запроса с условием, обратным условию применения правила

Наконец, для правил `ALSO` в список добавляется исходное дерево запроса без изменений. Так как исходное дерево запроса также добавляют только правила `INSTEAD` с условиями применения, в итоге для правила с одним действием мы можем получить только одно или два дерева запросов.

Для правил `ON INSERT` исходный запрос (если он не перекрывается режимом `INSTEAD`) выполняется перед действиями, добавленными правилами. Поэтому эти действия могут видеть вставленные строки. Но для правил `ON UPDATE` и `ON DELETE` исходный запрос выполняется после действий, добавленных правилами. При таком порядке эти действия будут видеть строки, подлежащие изменению или удалению; иначе бы действия не работали, не найдя строк, соответствующих их условиям применения (эти строки уже будут изменены или удалены).

Деревья запросов, полученные из действий правил, снова попадают в систему перезаписи, где могут применится дополнительные правила, добавляющие или убирающие деревья запроса. Поэтому действия правила должны выполнять команды другого типа или работать с другим результирующим отношением, иначе возникнет бесконечная рекурсия. (Система выявляет подобное рекурсивное разворачивание правил и выдаёт ошибку.)

Деревья запросов, заданные для действий в системном каталоге `pg_rewrite`, представляют собой только шаблоны. Так как они могут обращаться к элементам `NEW` и `OLD` в списке отношений, их можно будет использовать только после некоторых подстановок. В случае ссылки на `NEW` соответствующий элемент ищется в целевом списке исходного запроса. Если он найден, ссылка заменяется выражением этого элемента. В противном случае `NEW` означает то же самое, что и `OLD` (для команды `UPDATE`) или заменяется значением NULL (для команды `INSERT`). Любые ссылки на `OLD` заменяются ссылкой на элемент результирующего отношения в списке отношений.

После того как система применит все правила для изменения, она применяет правила представления к полученному дереву (или деревьям) запроса. Представления не могут добавлять новые действия для изменения, поэтому нет необходимости применять такие правила к результату перезаписи представления.

39.4.1.1. Пошаговый разбор первого правила

Предположим, что нам нужно отслеживать изменения в столбце `sl_avail` таблицы `shoelace_data`. Мы можем создать таблицу для ведения журнала и правило, которое будет добавлять в неё записи по условию, когда для `shoelace_data` выполняется `UPDATE`.

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- шнурки, количество которых изменилось
    sl_avail   integer,       -- новое количество
    log_who    text,          -- кто изменил
    log_when   timestamp      -- когда
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Теперь, если кто-то выполнит:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

мы увидим в таблице журнала:

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

Именно это нам и нужно. При этом внутри происходит следующее. Анализатор запроса создаёт дерево:

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

В системном каталоге находится правило `log_shoelace`, настроенное на изменение (`ON UPDATE`) с условием применения:

```
NEW.sl_avail <> OLD.sl_avail
```

и действием:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

(Это выглядит несколько странно, так как обычно нельзя написать `INSERT ... VALUES ... FROM`. Предложение `FROM` здесь добавлено, просто чтобы показать, что в дереве запроса для ссылок `new` и `old` есть элементы в списке отношений. Они необходимы для того, чтобы к ним могли обращаться переменные в дереве запроса команды `INSERT`.)

Так как это правило `ALSO` с условием применения, система правил должна выдать два дерева запросов: изменённое действие правила и исходное дерево запроса. На первом шаге список отношений исходного запроса вставляется в дерево действия правила и получается:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
```

```
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data;
```

На втором шаге в это дерево добавляется условие применения правила, так что результирующий набор ограничивается строками, в которых меняется `sl_avail`:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail;
```

(Это выглядит ещё более странно, ведь в `INSERT ... VALUES` не записывается и предложение `WHERE`, но планировщик и исполнитель не испытывают затруднений с этим. Они всё равно должны поддерживать эту функциональность для `INSERT ... SELECT`.)

На третьем шаге добавляется условие исходного дерева, что ещё больше ограничивает результирующий набор, оставляя в нём только строки, которые затронул бы исходный запрос:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

На четвёртом шаге ссылки на `NEW` заменяются элементами выходного списка из исходного дерева запроса или переменными из результирующего отношения:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

На последнем, пятом шаге ссылки на `OLD` заменяются ссылками на результирующее отношение:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Вот и всё. Так как правило действует в режиме `ALSO`, мы также выводим исходное дерево запроса. Таким образом, система правил выдаёт список с двумя деревьями запросов, соответствующими этим операторам:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
```

```
 WHERE sl_name = 'sl7';
```

Они выполняются в показанном порядке и именно это должно делать данное правило.

Благодаря заменам и добавленным условиям в журнал не добавится запись, например, при таком исходном запросе:

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

В этом случае исходное дерево запроса не содержит элемент выходного списка для `sl_avail`, так что `NEW.sl_avail` будет заменено переменной `shoelace_data.sl_avail`. Таким образом, дополнительная команда, созданная правилом, будет такой:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Это условие применения не будет выполняться никогда.

Это также будет работать, если исходный запрос изменяет несколько строк. Так, если кто-то выполнит команду:

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

фактически будут изменены четыре строки (`sl1`, `sl2`, `sl3` и `sl4`). Но для `sl3` значение `sl_avail = 0`. В этом случае условие исходного дерева другое, так что это правило выдаёт такое дополнительное дерево запроса:

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

. С таким деревом запроса в журнал определённо будут добавлены три записи. И это абсолютно правильно.

Здесь мы видим, почему важно, чтобы исходное дерево запроса выполнялось в конце. Если бы оператор `UPDATE` выполнился сначала, все строки уже получили бы нулевые значения, так что записывающий в журнал `INSERT` не нашёл бы строк, в которых `0 <> shoelace_data.sl_avail`.

39.4.2. Сочетание с представлениями

Есть один простой вариант защититься от ранее упомянутой возможности выполнять `INSERT`, `UPDATE` или `DELETE` для представлений, когда это нежелательно — создать правила, просто отбрасывающие деревья этих запросов. В нашем случае они будут выглядеть так:

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

Если теперь кто-то попытается выполнить одну из этих операций с представлением `shoe`, система правил применит эти правила. Так как это правила без действий в режиме `INSTEAD`, результирующий список деревьев запроса будет пуст и весь запрос аннулируется, так что после работы системы правил будет нечего оптимизировать и выполнять.

Более сложный вариант — использовать систему правил для создания правил, преобразующих дерево запроса в выполняющее нужную операцию с реальными таблицами. Чтобы реализовать это с представлением `shoelace`, мы создадим следующие правила:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

Если вы хотите поддерживать также запросы к представлению с `RETURNING`, вам надо создать правила с предложениями `RETURNING`, которые будут вычислять строки представления. Это обычно довольно тривиально для представлений с одной нижележащей таблицей, но несколько затруднительно для представлений с соединением, таких как `shoelace`. Например, для `INSERT` это будет выглядеть так:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

Заметьте, что это одно правило поддерживает запросы и `INSERT`, и `INSERT RETURNING` к этому представлению — предложение `RETURNING` просто игнорируется при обычном `INSERT`.

Теперь предположим, что на фабрику прибывает партия шнурков с объёмной сопроводительной накладной. Но вы не хотите вручную вносить по одной записи в представление `shoelace`. Вместо этого можно создать две маленькие таблицы: в первую вы будете вставлять записи из накладной, а вторая пригодится для специального приёма. Для этого мы выполним следующие команды:

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
```

```
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Теперь вы можете наполнить таблицу `shoelace_arrive` данными о поступивших шнурках из накладной:

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

Взгляните на текущие данные:

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Теперь переместите прибывшие шнурки во вторую таблицу:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

Проверьте, что получилось:

```
SELECT * FROM shoelace ORDER BY sl_name;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;
```

```
 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

Чтобы получить эти результаты из одного `INSERT ... SELECT`, была проделана большая работа. Мы подробно опишем всё преобразование дерева запросов в продолжении этой главы. Начнём с дерева, выданного анализатором запроса:

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Теперь применяется первое правило `shoelace_ok_ins`, создающее такое дерево:

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

и отбрасывающее исходный `INSERT` в `shoelace_ok`. Этот переписанный запрос снова поступает в систему правил и второе применяемое правило `shoelace_upd` выдаёт:

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = shoelace.sl_name;
```

Это тоже правило `INSTEAD`, так что предыдущее дерево запроса отбрасывается. Заметьте, что этот запрос по-прежнему использует представление `shoelace`. Но система правил ещё не закончила свою работу, она продолжает и применяет правило `_RETURN`, так что мы получаем:

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

Наконец, применяется правило `log_shoelace` и выдаётся дополнительное дерево запроса:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u,
       shoelace_data old, shoelace_data new
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

Теперь, обработав все правила, система правил выдаёт построенные деревья запросов.

В итоге мы получаем два дерева запросов, равнозначные следующим операторам SQL:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

В результате вся операция, в ходе которой данные, поступающие из одного отношения, вставляются в другое, вставка преобразуется в изменение третьего, что затем становится изменением четвёртого, и запись об этом изменении добавляется в пятое, сводится к двум запросам.

Здесь можно заметить маленькую не очень красивую деталь. Как видно, в этих двух запросах таблица `shoelace_data` фигурирует в списке отношений дважды, тогда как определённо достаточно и одного вхождения. Планировщик не понимает этого и поэтому для дерева запроса `INSERT`, выданного системой правил, будет получен такой план:

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

Тогда как без лишнего элемента в списке отношений мы получили бы:

```
Merge Join
```

```
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

При этом в журнале оказались бы точно такие же записи. Таким образом, применение правил повлекло дополнительное сканирование таблицы `shoelace_data`, в котором не было никакой необходимости. И такое же избыточное сканирование выполняется ещё раз в `UPDATE`. Отнеситесь к этому с пониманием, ведь сделать всё это возможным в принципе было действительно сложно.

И наконец, ещё одна, завершающая демонстрация системы правил Postgres Pro и всей её мощи. Предположим, что вы добавили в базу данных шнурки с экстраординарными цветами:

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Давайте создадим представление, чтобы убедиться, что шнурки (записи в `shoelace`) не подходят ни к каким туфлям. Оно будет определено так:

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Через него мы получаем наши записи:

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

Теперь мы хотим, чтобы шнурки, которые ни к чему не подходят, удалялись из базы данных. Чтобы немного усложнить задачу для Postgres Pro, мы не будем удалять их непосредственно из таблицы. Вместо этого мы создадим ещё одно представление:

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

И удалим их так:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
             WHERE sl_name = shoelace.sl_name);
```

В результате получим:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |        80
 sl2     |        6 | black    |    100 | cm      |       100
 sl7     |        6 | brown    |     60 | cm      |        60
 sl4     |        8 | black    |     40 | inch    |     101.6
 sl3     |       10 | black    |     35 | inch    |      88.9
 sl8     |       21 | brown    |     40 | inch    |     101.6
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
 sl5     |        4 | brown    |      1 | m       |       100
 sl6     |       20 | brown    |    0.9 | m       |        90
(9 rows)
```

Так запрос `DELETE` для представления с ограничивающим условием-подзапросом, использующим в совокупности 4 вложенных/соединённых представления, с одним из которых тоже связано условие с подзапросом, задействующим представление, и где используются вычисляемые столбцы представлений, переписывается и преобразуется в одно дерево запроса, которое удаляет требуемые данные из реальной таблицы.

На практике ситуации, когда необходима такая сложная конструкция, встречаются довольно редко, но тем не менее приятно осознавать, что всё это возможно и работает.

39.5. Правила и права

В результате переписывания запросов системой правил Postgres Pro обращение может происходить не к тем таблицам/представлениям, к которым обращался исходный запрос. С правилами для изменения возможна так же и запись в другие таблицы.

Правила перезаписи не имеют отдельного владельца — владельцем правил перезаписи, определённых для отношения (таблицы или представления), автоматически считается владелец этого отношения. Система правил Postgres Pro меняет поведение стандартного механизма управления доступом. Ко всем отношениям, используемым вследствие применения правил, проверяется доступ владельца правила, но не пользователя, выполняющего запрос (кроме представлений с характеристикой SECURITY INVOKER по правилам `SELECT`, см. `CREATE VIEW`). Это значит, что если речь идёт не о таких представлениях, пользователь должен иметь права, необходимые только для обращения к таблицам/представлениям, которые он явно упоминает в своих запросах.

Например, представим, что у пользователя есть список телефонных номеров, некоторые из которых личные, а некоторые должна знать его ассистентка. Он может построить следующую конструкцию:

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

Никто, кроме него (и суперпользователей базы данных) не сможет обратиться к таблице `phone_data`. Но так как ассистентке было дано (`GRANT`) соответствующее право, она сможет выполнить `SELECT` для представления `phone_number`. Система правил преобразует `SELECT` из `phone_number` в `SELECT` из таблицы `phone_data`. Так как пользователь является владельцем `phone_number`, он же считается владельцем правила, доступ на чтение `phone_data` проверяется для него, и выполнение запроса разрешается. Проверка прав доступа к `phone_number` тоже выполняется, но при этом проверяется пользователь, выполняющий запрос, так что обращаться к этому представлению смогут только сам пользователь и его ассистентка.

Права проверяются правило за правилом. То есть, в данный момент только ассистентка может видеть открытые телефонные номера. Но она может создать другое представление и дать доступ к нему всем (роли `public`), после чего все смогут видеть данные `phone_number` через представление ассистентки. Что она не может сделать, так это создать представление, которое обращается к `phone_data` напрямую. (Вообще она может это сделать, но такое представление не будет работать, так как при любой попытке прочитать его доступ к таблице будет запрещён.) И как только пользователь заметит, что ассистентка открыла доступ к своему представлению `phone_number`, он может лишить её права чтения этого представления. В результате все сразу потеряют доступ и к представлению ассистентки.

Может показаться, что такая проверка «правило-за-правилом» представляет уязвимость, но это не так. Если бы даже этот механизм не работал, ассистентка могла бы создать таблицу со столбцами как в `phone_number` и регулярно копировать туда данные. Тогда это были бы её собственные данные и она могла бы открывать доступ к ним кому угодно. Другими словами, команда `GRANT` означает «Я доверяю тебе». Если кто-то, кому вы доверяете, проделывает такие операции, стоит задуматься и, возможно, лишить его доступа к данным, применив `REVOKE`.

Хотя представления могут применяться для скрытия содержимого определённых столбцов, как описано выше, с их помощью нельзя надёжно скрыть данные в невидимых строках, если только не установлен флаг `security_barrier`. Например, следующее представление небезопасно:

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Может показаться, что всё в порядке, ведь система правил преобразует `SELECT` из `phone_number` в `SELECT` из `phone_data` и добавит ограничивающее условие, чтобы выдавались только строки с полем `phone`, начинающимся не с 412. Но если пользователь может создавать собственные функции, ему будет не сложно заставить планировщик выполнять функцию пользователя перед выражением `NOT LIKE`. Например:

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END;
$$ LANGUAGE plpgsql COST 0.0000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

Так он сможет получить все имена и номера телефонов из таблицы `phone_data` через сообщения `NOTICE`, так как планировщик решит, что лучше выполнить недорогую функцию `tricky` перед более дорогой операцией `NOT LIKE`. И даже если пользователь не имеет права создавать новые функции, он может использовать для подобных атак встроенные функции. (Например, многие функции приведения показывают входные значения в сообщениях об ошибках.)

Подобные соображения распространяются и на правила для изменения. Применительно к примерам предыдущего раздела, владелец таблиц в базе данных может дать кому-нибудь другому для представления `shoelace` права `SELECT`, `INSERT`, `UPDATE` и `DELETE`, а для `shoelace_log` только `SELECT`. Действие правила, добавляющее записи в журнал, всё равно будет выполняться успешно, а этот другой пользователь сможет видеть записи в журнале. Но он не сможет создавать поддельные записи, равно как и модифицировать или удалять существующие. В этом случае нет никакой возможности заставить планировщик изменить порядок операций, так как единственное правило, которое обращается к `shoelace_log` — это безусловный `INSERT`. В более сложных сценариях это может быть не так.

Когда требуется, чтобы представление обеспечивало защиту на уровне строк, к нему нужно применить атрибут `security_barrier`. Это предотвратит утечку содержимого строк из злонамеренно выбранных функций и операторов до того, как строки будут отфильтрованы представлением. Например, показанное выше представление будет безопасным, если создать его так:

```
CREATE VIEW phone_number WITH (security_barrier) AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Представления, созданные с атрибутом `security_barrier`, могут работать гораздо медленнее, чем обычные. И вообще говоря, это неизбежно: самый быстрый план должен быть отвергнут, если он может скомпрометировать защиту. Поэтому данный атрибут по умолчанию не устанавливается.

Планировщик запросов имеет больше свободы, работая с функциями, лишёнными побочных эффектов. Такие функции называются герметичными (`LEAKPROOF`) и включают только простые часто используемые операторы, например, операторы равенства. Планировщик запросов может безопасно вычислять такие функции в любой момент выполнения запроса, так как при вызове их для строк, невидимых пользователю, не просочится никакая информация об этих строках. Более того, функции, которые не принимают аргументы или которым не передаются аргументы из представления с барьером безопасности, можно не помечать как `LEAKPROOF`, чтобы они вышли наружу, так как они никогда не получат данные из представления. И напротив, функции, которые могут вызвать ошибку в зависимости от значений аргументов (например, в случае переполнения или деления на ноль), герметичными не являются, и могут выдать существенную информацию о невидимых строках, если будут выполнены перед фильтрами строк.

Важно понимать, что даже представление, созданное с атрибутом `security_barrier`, остаётся безопасным только в том смысле, что содержимое невидимых строк не будет передаваться потенциально небезопасным функциям. Но пользователь может собрать некоторые сведения о невидимых данных и другими способами; например, он может проанализировать план запроса, полученный с `EXPLAIN`, или замерить время выполнения запросов с этим представлением. Злоумышленник может сделать определённые выводы об объёме невидимых данных или даже получить некоторую информацию о распределении данных или наиболее частых значениях (так как всё это отражается в статистике для оптимизатора и, как следствие, влияет на время выполнения плана или даже на выбор плана). Если возможность атаки через скрытые каналы вызывает опасения, вероятно, будет разумным не предоставлять никакой доступ к этим данным.

39.6. Правила и статус команд

Сервер Postgres Pro возвращает строку состояния команды, например, `INSERT 149592 1`, для каждой получаемой команды. Это довольно прозрачно, когда не задействуются правила, но что произойдёт, если правила перезапишут запрос?

Правила влияют на состояния команды следующим образом:

- Если с запросом не связано безусловное правило `INSTEAD`, то выполняется заданный исходный запрос и его статус выдаётся как обычно. (Но если определены какие-то условные правила `INSTEAD`, к исходному запросу добавляется условие, обратное их условиям применения. Это может повлиять на число обрабатываемых строк и выводимый статус команды.)
- Если с запросом связано безусловное правило `INSTEAD`, исходный запрос не выполняется вовсе. В этом случае сервер возвратит статус команды от последнего запроса, вставленного правилом `INSTEAD` (условным или безусловным), и тип команды исходного запроса (`INSERT`, `UPDATE` или `DELETE`). Если правила не добавили подходящего запроса, в возвращённом статусе команды показывается исходный тип запроса и нули вместо количества строк и OID.

Программист может добиться, чтобы статус команды во втором случае устанавливало нужное правило `INSTEAD`, назначив ему имя, стоящее по алфавиту после других активных правил, чтобы это правило применялось последним.

39.7. Сравнение правил и триггеров

Многие вещи, которые можно сделать с помощью триггеров, можно также реализовать, используя систему правил Postgres Pro. Однако, используя правила, нельзя реализовать, например, некоторые типы ограничений, в частности, внешние ключи. Хотя можно определить правило с ограничивающим условием, которое будет преобразовать команду в `NOTHING`, если значение ключа не находится в другой таблице, но при этом неподходящие данные будут отбрасываться молча, а это не самый лучший вариант. Также, если требуется проверить правильность значений и, обнаружив неверное значение, выдать ошибку, это нужно делать в триггере.

В этой главе мы разберём использование правил для изменения представлений. Все правила, приведённые в примерах этой главы, можно также заменить триггерами `INSTEAD OF` для представлений. Написать такие триггеры часто бывает проще, чем разработать правила, особенно если для изменений применяется сложная логика.

Для тех задач, которые можно решить обоими способами, лучший выбирается в зависимости от характера использования базы данных. Следует учитывать, что триггер срабатывает для каждой обрабатываемой строки, а правило изменяет существующий запрос или создаёт ещё один. Поэтому, если один оператор обрабатывает сразу много строк, правило, добавляющее дополнительную команду, скорее всего, будет работать быстрее, чем триггер, который вызывается для каждой очередной строки и должен каждый раз определять, что с ней делать. Однако триггеры концептуально гораздо проще правил, и использовать их правильно новичкам гораздо проще.

Давайте рассмотрим пример, показывающий, как выбор в пользу правил вместо триггеров оказывается выигрышным в определённой ситуации. Пусть у нас есть две таблицы:

```
CREATE TABLE computer (
    hostname        text,    -- индексированное
    manufacturer    text     -- индексированное
);

CREATE TABLE software (
    software        text,    -- индексированное
    hostname        text     -- индексированное
);
```

Обе таблицы содержат несколько тысяч строк, а индексы по полю `hostname` являются уникальными. Правило или триггер должны реализовать ограничение, которое удалит строки из таблицы `software`, ссылающиеся на удаляемый компьютер. Триггер выполнял бы такую команду:

```
DELETE FROM software WHERE hostname = $1;
```

Так как триггер вызывается для каждой отдельной строки, удаляемой из таблицы `computer`, он может подготовить и сохранить план этой команды, а затем передавать значение `hostname` подготовленному запросу в параметрах. Правило же можно записать так:

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Теперь давайте взглянем на разные варианты удаления. В этом случае:

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

таблица `computer` сканируется по индексу (быстро), и команда, выполняемая триггером, так же будет применять сканирование по индексу (тоже быстро). Дополнительной командой правила будет:

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

Так как созданы все необходимые индексы, планировщик создаст план

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

Таким образом, большого различия в скорости между реализациями с триггером и с правилом не будет.

Теперь мы хотим избавиться от 2000 компьютеров, у которых `hostname` начинается с `old`. Это можно сделать двумя командами. Первая:

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

Правило преобразует её в:

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                       AND software.hostname = computer.hostname;
```

с планом:

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

С другой возможной командой:

```
DELETE FROM computer WHERE hostname ~ '^old';
```

для запроса, преобразованного правилом, получается следующий план:

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
```

```
  ->  Index Scan using soft_hostidx on software
```

Это показывает, что планировщик не понимает, что ограничение по `hostname` в `computer` можно также использовать для сканирования по индексу в `software`, когда несколько условий объединяются с помощью `AND`, что он успешно делает для варианта команды с регулярным выражением. Триггер будет вызываться для каждой из 2000 удаляемых записей о старых компьютерах, и это приведёт к одному сканированию индекса в таблице `computer` и 2000 сканированиям индекса в таблице `software`. Реализация с правилом делает это двумя командами, применяющими индексы. Будет ли правило быстрее при последовательном сканировании, зависит от общего размера таблицы `software`. С другой стороны, выполнение 2000 команд из триггера через менеджер SPI всё равно займёт время, даже если все блоки индекса вскоре окажутся в кеше.

В завершение взгляните на эту команду:

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

Она также может привести к удалению множества строк из таблицы `computer`. Поэтому триггер снова пропустит через исполнитель такое же множество команд. Правило же выдаст следующую команду:

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

План для этой команды снова будет содержать вложенный цикл по двум сканированиям индекса, но на этот раз с другим индексом таблицы `computer`:

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

Во всех этих случаях дополнительные команды будут более-менее независимыми от числа затрагиваемых строк.

Таким образом, правила будут значительно медленнее триггеров, только если их действия приводят к образованию больших и плохо связанных соединений, когда планировщик оказывается бессилен.

Глава 40. Процедурные языки

Postgres Pro позволяет разрабатывать пользовательские функции не только на SQL и C, но и на других языках. Эти языки в целом называются *процедурными языками* (PL, Procedural Language). Если функция написана на процедурном языке, сервер баз данных сам по себе не знает, как интерпретировать её исходный текст. Вместо этого он передаёт эту задачу специальному обработчику, понимающему данный язык. Обработчик может либо выполнить всю работу по разбору, синтаксическому анализу, выполнению кода и т. д., либо действовать как «прослойка» между Postgres Pro и внешним исполнителем языка программирования. Сам обработчик представляет собой функцию на языке C, скомпилированную в виде разделяемого объекта и загружаемую по требованию, как и любая другая функция на C.

В настоящее время стандартный дистрибутив Postgres Pro включает четыре процедурных языка: PL/pgSQL (Глава 41), PL/Tcl (Глава 42), PL/Perl (Глава 43) и PL/Python (Глава 44). Существуют и другие процедурные языки, поддержка которых не включена в базовый дистрибутив. Информацию о них можно найти в Приложении J. Кроме того, пользователи могут реализовать и другие языки; основы разработки нового процедурного языка рассматриваются в Главе 54.

40.1. Установка процедурных языков

Прежде всего, процедурный язык должен быть «установлен» в каждую базу данных, где он будет использоваться. Но процедурные языки, устанавливаемые в базу данных `template1`, автоматически становятся доступными во всех впоследствии создаваемых базах, так как их определения в `template1` будут скопированы командой `CREATE DATABASE`. Таким образом, администратор баз данных может выбрать, какие языки будут доступны в определённых базах данных, и при желании сделать некоторые языки доступными по умолчанию.

Для языков, включённых в стандартный дистрибутив, достаточно выполнить команду `CREATE EXTENSION` *`имя_языка`*, чтобы установить язык в текущую базу данных. Описанная ниже ручная процедура рекомендуется только для установки языков, не упакованных в виде расширений.

Установка процедурного языка вручную

Процедурный язык устанавливается в базу данных в пять этапов, и выполнять их должен администратор баз данных. В большинстве случаев необходимые команды SQL следует упаковать в виде установочного скрипта «расширения», чтобы их можно было выполнить, воспользовавшись командой `CREATE EXTENSION`.

1. Разделяемый объект для обработчика языка должен быть скомпилирован и установлен в соответствующий каталог библиотек. Это в принципе не отличается от сборки и установки дополнительных модулей с обычными функциями на языке C; см. Подраздел 36.10.5. Часто обработчик языка зависит от внешней библиотеки, в которой собственно реализован исполнитель языка программирования; в таких случаях нужно установить и эту библиотеку.

2. Обработчик должен быть объявлен командой

   ```
   CREATE FUNCTION имя_функции_обработчика()
       RETURNS language_handler
       AS 'путь-к-разделяемому-объекту'
       LANGUAGE C;
   ```

   Специальный тип возврата `language_handler` говорит СУБД, что эта функция не возвращает какой-либо определённый тип данных SQL, и значит её нельзя использовать непосредственно в операторах SQL.

3. (Optional) Дополнительно обработчик языка может предоставить функцию обработки «внедрённого кода», которая будет выполнять анонимные блоки кода (команды (`DO`), написанные на этом языке. Если для языка есть обработчик внедрённого кода, объявите его такой командой:

   ```
   CREATE FUNCTION имя_обработчика_внедрённого_кода(internal)
       RETURNS void
   ```

```
    AS 'путь-к-разделяемому-объекту'
    LANGUAGE C;
```

4. (Optional) Кроме того, обработчик языка может предоставить функцию «проверки», которая будет проверять корректность определения функции, собственно не выполняя её. Функция проверки, если она существует, вызывается командой `CREATE FUNCTION`. Если такая функция для языка определена, объявите её такой командой:

   ```
   CREATE FUNCTION имя_функции_проверки(oid)
       RETURNS void
       AS 'путь-к-разделяемому-объекту'
       LANGUAGE C STRICT;
   ```

5. Наконец, процедурный язык должен быть объявлен командой

   ```
   CREATE [TRUSTED] LANGUAGE имя_языка
       HANDLER имя_функции_обработчика
       [INLINE имя_обработчика_внедрённого_кода]
       [VALIDATOR имя_функции_проверки] ;
   ```

   Необязательное ключевое слово `TRUSTED` (доверенный) указывает, что язык не предоставляет пользователю доступ к данным, которого он не имел бы без него. Доверенные языки предназначены для обычных пользователей баз данных, не имеющих прав суперпользователя, и их можно использовать для безопасного создания функций и процедур. Так как функции PL выполняются внутри сервера баз данных, флаг `TRUSTED` следует устанавливать только для тех языков, которые не позволяют обращаться к внутренним механизмам сервера или файловой системе. Языки PL/pgSQL, PL/Tcl и PL/Perl считаются доверенными; языки PL/TclU, PL/PerlU и PL/PythonU предоставляют неограниченную функциональность, и их *не* следует помечать как доверенные.

Примере 40.1 показывает, как выполняется процедура ручной установки для языка PL/Perl.

Пример 40.1. Установка PL/Perl вручную

Следующая команда говорит серверу баз данных, где найти разделяемый объект для функции-обработчика языка PL/Perl:

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

Для PL/Perl реализованы обработчик внедрённого кода и функция проверки, так что их мы тоже объявим:

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;
```

Следующая команда:

```
CREATE TRUSTED LANGUAGE plperl
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

определяет, что ранее объявленные функции должны вызываться для функций и процедур с атрибутом языка `plperl`.

В стандартной инсталляции Postgres Pro обработчик языка PL/pgSQL уже собран и установлен в каталог «библиотек»; более того, сам язык PL/pgSQL установлен во всех базах данных. Если при сборке сконфигурирована поддержка Tcl, то обработчики для PL/Tcl и PL/TclU собираются и устанавливаются в каталог библиотек, но сам язык по умолчанию в базы данных не устанавливается. Подобным образом, если сконфигурирована поддержка Perl, собираются и устанавливаются обработчики PL/Perl и PL/PerlU, а при включении поддержки Python устанавливается обработчик PL/PythonU, но в базы данных эти языки по умолчанию не устанавливаются.

Глава 41. PL/pgSQL — процедурный язык SQL

41.1. Обзор

PL/pgSQL это процедурный язык для СУБД Postgres Pro. Целью проектирования PL/pgSQL было создание загружаемого процедурного языка, который:

- используется для создания функций, процедур и триггеров,
- добавляет управляющие структуры к языку SQL,
- может выполнять сложные вычисления,
- наследует все пользовательские типы, функции, процедуры и операторы,
- может быть определён как доверенный язык,
- прост в использовании.

Функции PL/pgSQL могут использоваться везде, где допустимы встроенные функции. Например, можно создать функции со сложными вычислениями и условной логикой, а затем использовать их при определении операторов или в индексных выражениях.

В версии PostgreSQL 9.0 и выше PL/pgSQL устанавливается по умолчанию. Тем не менее это по-прежнему загружаемый модуль и администраторы, особо заботящиеся о безопасности, могут удалить его при необходимости.

41.1.1. Преимущества использования PL/pgSQL

Postgres Pro и большинство других СУБД используют SQL в качестве языка запросов. SQL хорошо переносим и прост в изучении. Однако каждый оператор SQL выполняется индивидуально на сервере базы данных.

Это значит, что ваше клиентское приложение должно каждый запрос отправлять на сервер, ждать пока он будет обработан, получать результат, делать некоторые вычисления, затем отправлять последующие запросы на сервер. Всё это требует межпроцессного взаимодействия, а также несёт нагрузку на сеть, если клиент и сервер базы данных расположены на разных компьютерах.

PL/pgSQL позволяет сгруппировать блок вычислений и последовательность запросов *внутри* сервера базы данных, таким образом, мы получаем силу процедурного языка и простоту использования SQL при значительной экономии накладных расходов на клиент-серверное взаимодействие.

- Исключаются дополнительные обращения между клиентом и сервером
- Промежуточные ненужные результаты не передаются между сервером и клиентом
- Есть возможность избежать многочисленных разборов одного запроса

В результате это приводит к значительному увеличению производительности по сравнению с приложением, которое не использует хранимых функций.

Кроме того, PL/pgSQL позволяет использовать все типы данных, операторы и функции SQL.

41.1.2. Поддерживаемые типы данных аргументов и возвращаемых значений

Функции на PL/pgSQL могут принимать в качестве аргументов все поддерживаемые сервером скалярные типы данных или массивы и возвращать в качестве результата любой из этих типов. Они могут принимать и возвращать любой именованный составной тип (тип кортежа). Также есть возможность объявить функцию на PL/pgSQL как принимающую `record`, то есть ей может быть передан любой составной тип, или как возвращающую `record`, то есть её результатом будет кортеж, столбцы которого определит спецификация вызывающего запроса, как описано в Подразделе 7.2.1.4.

Использование маркера `VARIADIC` позволяет объявлять функции на PL/pgSQL с переменным числом аргументов. Это работает точно так же, как и для функций на SQL, как описано в Подразделе 36.5.6.

Функции на PL/pgSQL могут также принимать и возвращать полиморфные типы, описанные в Подразделе 36.2.5, вследствие чего фактические типы данных, обрабатываемые функцией, могут меняться от вызова к вызову. Примеры приведены в Подразделе 41.3.1.

Функции на PL/pgSQL могут возвращать «множества» (или таблицы) любого типа, которые могут быть возвращены в виде одного объекта. Такие функции генерируют вывод, выполняя команду `RETURN NEXT` для каждого элемента результирующего набора или `RETURN QUERY` для вывода результата запроса.

Наконец, при отсутствии полезного возвращаемого значения функция на PL/pgSQL может возвращать `void`. (С другой стороны, её также можно оформить в виде процедуры.)

Функции на PL/pgSQL можно объявить с выходными параметрами вместо явного задания типа возвращаемого значения. Это не добавляет никаких фундаментальных возможностей языку, но часто бывает удобно, особенно для возвращения нескольких значений. Нотация `RETURNS TABLE` может использоваться вместо `RETURNS SETOF`.

Конкретные примеры рассматриваются в Подразделе 41.3.1 и Подразделе 41.6.1.

41.2. Структура PL/pgSQL

Функции, написанные на PL/pgSQL, определяются на сервере командами CREATE FUNCTION. Такая команда обычно выглядит, например, так:

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'тело функции'
LANGUAGE plpgsql;
```

Если рассматривать `CREATE FUNCTION`, тело функции представляет собой просто текстовую строку. Часто для написания тела функции удобнее заключать эту строку в доллары (см. Подраздел 4.1.2.4), а не в обычные апострофы. Если не применять заключение в доллары, все апострофы или обратные косые черты в теле функции придётся экранировать, дублируя их. Почти во всех примерах в этой главе тело функций заключается в доллары.

PL/pgSQL это блочно-структурированный язык. Текст тела функции должен быть *блоком*. Структура блока:

```
[ <<метка>> ]
[ DECLARE
    объявления ]
BEGIN
    операторы
END [ метка ];
```

Каждое объявление и каждый оператор в блоке должны завершаться символом «;» (точка с запятой). Блок, вложенный в другой блок, должен иметь точку с запятой после `END`, как показано выше. Однако финальный `END`, завершающий тело функции, не требует точки с запятой.

Подсказка

Распространённой ошибкой является добавление точки с запятой сразу после `BEGIN`. Это неправильно и приведёт к синтаксической ошибке.

`Метка` требуется только тогда, когда нужно идентифицировать блок в операторе `EXIT`, или дополнить имена переменных, объявленных в этом блоке. Если метка указана после `END`, то она должна совпадать с меткой в начале блока.

Ключевые слова не чувствительны к регистру символов. Как и в обычных SQL-командах, идентификаторы неявно преобразуются к нижнему регистру, если они не взяты в двойные кавычки.

Комментарии в PL/pgSQL коде работают так же, как и в обычном SQL. Двойное тире (`--`) начинает комментарий, который завершается в конце строки. Блочный комментарий начинается с `/*` и завершается `*/`. Блочные комментарии могут быть вложенными.

Любой оператор в выполняемой секции блока может быть *вложенным блоком*. Вложенные блоки используются для логической группировки нескольких операторов или локализации области действия переменных для группы операторов. Во время выполнения вложенного блока переменные, объявленные в нём, скрывают переменные внешних блоков с такими же именами. Чтобы получить доступ к внешним переменным, нужно дополнить их имена меткой блока. Например:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 30
    quantity := 50;
    --
    -- Вложенный блок
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 80
        RAISE NOTICE 'Во внешнем блоке quantity = %', outerblock.quantity;  --
 Выводится 50
    END;

    RAISE NOTICE 'Сейчас quantity = %', quantity;  -- Выводится 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Примечание

Существует скрытый «внешний блок», окружающий тело каждой функции на PL/pgSQL. Этот блок содержит объявления параметров функции (если они есть), а также некоторые специальные переменные, такие как `FOUND` (см. Подраздел 41.5.5). Этот блок имеет метку, совпадающую с именем функции, таким образом, параметры и специальные переменные могут быть дополнены именем функции.

Важно не путать использование `BEGIN`/`END` для группировки операторов в PL/pgSQL с одноимёнными SQL-командами для управления транзакциями. `BEGIN`/`END` в PL/pgSQL служат только для группировки предложений; они не начинают и не заканчивают транзакции. Управление транзакциями в PL/pgSQL описывается в Разделе 41.8. Кроме того, блок с предложением `EXCEPTION` по сути создаёт вложенную транзакцию, которую можно отменить, не затрагивая внешнюю транзакцию. Подробнее это описано в Подразделе 41.6.8.

41.3. Объявления

Все переменные, используемые в блоке, должны быть определены в секции объявления. (За исключением переменной-счётчика цикла `FOR`, которая объявляется автоматически. Для цикла по диапазону чисел автоматически объявляется целочисленная переменная, а для цикла по результатам курсора - переменная типа `record`.)

Переменные PL/pgSQL могут иметь любой тип данных SQL, такой как `integer`, `varchar`, `char`.

Примеры объявления переменных:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Общий синтаксис объявления переменной:

```
имя [ CONSTANT ] тип [ COLLATE имя_правила_сортировки ] [ NOT NULL ] [ { DEFAULT | := |
 = } выражение ];
```

Предложение `DEFAULT`, если присутствует, задаёт начальное значение, которое присваивается переменной при входе в блок. Если отсутствует, то переменная инициализируется SQL-значением `NULL`. Указание `CONSTANT` предотвращает изменение значения переменной после инициализации, таким образом, значение остаётся постоянным в течение всего блока. Параметр `COLLATE` определяет правило сортировки, которое будет использоваться для этой переменной (см. Подраздел 41.3.6). Если указано `NOT NULL`, то попытка присвоить `NULL` во время выполнения приведёт к ошибке. Все переменные, объявленные как `NOT NULL`, должны иметь непустые значения по умолчанию. Можно использовать знак равенства (=) вместо совместимого с PL/SQL `:=`.

Значение по умолчанию вычисляется и присваивается переменной каждый раз при входе в блок (не только при первом вызове функции). Так, например, если переменная типа `timestamp` имеет функцию `now()` в качестве значения по умолчанию, это приведёт к тому, что переменная всегда будет содержать время текущего вызова функции, а не время, когда функция была предварительно скомпилирована.

Примеры:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
transaction_time CONSTANT timestamp with time zone := now();
```

После объявления переменной её значение можно использовать в последующих выражениях инициализации в этом же блоке, например:

```
DECLARE
  x integer := 1;
  y integer := x + 1;
```

41.3.1. Объявление параметров функции

Переданные в функцию параметры именуются идентификаторами `$1`, `$2` и т. д. Дополнительно, для улучшения читаемости, можно объявить псевдонимы для параметров `$n`. Либо псевдоним, либо цифровой идентификатор используются для обозначения параметра.

Создать псевдоним можно двумя способами. Предпочтительный способ это дать имя параметру в команде `CREATE FUNCTION`, например:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Другой способ это явное объявление псевдонима при помощи синтаксиса:

```
имя ALIAS FOR $n;
```

Предыдущий пример для этого стиля выглядит так:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Примечание

Эти два примера не полностью эквивалентны. В первом случае на `subtotal` можно ссылаться как `sales_tax.subtotal`, а во втором случае такая ссылка невозможна. (Если бы к внутреннему блоку была добавлена метка, то `subtotal` можно было бы дополнить этой меткой.)

Ещё несколько примеров:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- вычисления, использующие v_string и index
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Когда функция на PL/pgSQL объявляется с выходными параметрами, им выдаются цифровые идентификаторы `$n` и для них можно создавать псевдонимы точно таким же способом, как и для обычных входных параметров. Выходной параметр это фактически переменная, стартующая с NULL и которой присваивается значение во время выполнения функции. Возвращается последнее присвоенное значение. Например, функция sales_tax может быть переписана так:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Обратите внимание, что мы опустили `RETURNS real` — хотя можно было и включить, но это было бы излишним.

Чтобы вызвать функцию с параметрами `OUT`, при вызове функции не указывайте выходные параметры:

```
SELECT sales_tax(100.00);
```

Выходные параметры наиболее полезны для возвращения нескольких значений. Простейший пример:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
```

```
END;
$$ LANGUAGE plpgsql;

SELECT * FROM sum_n_product(2, 4);
 sum | prod
-----+------
   6 |    8
```

Как обсуждалось в Подразделе 36.5.4, здесь фактически создаётся анонимный тип `record` для возвращения результата функции. Если используется предложение `RETURNS`, то оно должно выглядеть как `RETURNS record`.

Выходные параметры поддерживаются и в процедурах, например:

```
CREATE PROCEDURE sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

При вызове процедуры нужно указать все параметры. Вызывая процедуру на чистом SQL, вместо выходных параметров можно указать `NULL`:

```
CALL sum_n_product(2, 4, NULL, NULL);
 sum | prod
-----+------
   6 |    8
```

Однако в вызове процедуры из PL/pgSQL для каждого выходного параметра нужно указать переменную; эта переменная будет получать результат вызова. За подробностями обратитесь к Подразделу 41.6.3.

Есть ещё способ объявить функцию на PL/pgSQL с использованием `RETURNS TABLE`, например:

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

Это в точности соответствует объявлению одного или нескольких параметров `OUT` и указанию `RETURNS SETOF` *`некий_тип`*.

Для функции на PL/pgSQL, возвращающей полиморфный тип (см. Подраздел 36.2.5), создаётся специальный параметр `$0`. Его тип данных соответствует типу, фактически возвращаемому функцией, который устанавливается на основании фактических типов входных параметров. Это позволяет функции обращаться к фактически возвращаемому типу данных, как показано в Подразделе 41.3.3. Параметр `$0` инициализируется в `NULL` и его можно изменять внутри функции. Таким образом, его можно использовать для хранения возвращаемого значения, хотя это необязательно. Параметру `$0` можно дать псевдоним. В следующем примере функция работает с любым типом данных, поддерживающим оператор `+`:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
```

```
END;
$$ LANGUAGE plpgsql;
```

Такой же эффект получается при объявлении одного или нескольких выходных параметров полиморфного типа. При этом `$0` не создаётся; выходные параметры сами используются для этой цели. Например:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

На практике может быть полезнее объявить полиморфную функцию, используя семейство типов `anycompatible`, чтобы входные аргументы автоматически сводились к общему типу. Например:

```
CREATE FUNCTION add_three_values(v1 anycompatible, v2 anycompatible, v3 anycompatible)
RETURNS anycompatible AS $$
BEGIN
    RETURN v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

Показанная функция позволяет выполнить такой вызов:

```
SELECT add_three_values(1, 2, 4.7);
```

В данном случае целочисленные аргументы будут автоматически приведены к numeric. В функции же, использующей `anyelement`, вам нужно было бы преобразовать все три аргумента к одному вручную.

41.3.2. `ALIAS`

`новое_имя` `ALIAS FOR` *`старое_имя`*`;`

Синтаксис `ALIAS` более общий, чем предполагалось в предыдущем разделе: псевдонимы можно объявлять для любых переменных, а не только для параметров функции. Основная практическая польза в том, чтобы назначить другие имена переменным с предопределёнными названиями, такими как `NEW` или `OLD` в триггерной функции.

Примеры:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

Поскольку `ALIAS` даёт два различных способа именования одних и тех же объектов, то его неограниченное использование может привести к путанице. Лучше всего использовать `ALIAS` для переименования предопределённых имён.

41.3.3. Наследование типов данных

`имя таблица.столбец``%TYPE`
`имя переменная``%TYPE`

Конструкция `%TYPE` предоставляет тип данных столбца таблицы или ранее объявленной переменной PL/pgSQL. Её можно использовать для объявления переменных, содержащих значения из базы данных. Например, для объявления переменной с таким же типом, как и столбец `user_id` в таблице `users`, нужно написать:

```
user_id users.user_id%TYPE;
```

Кроме того, после `%TYPE` можно записать оформление массива, таким образом создав переменную, содержащую массив значений того типа, на который она ссылается:

```
user_ids users.user_id%TYPE[];
user_ids users.user_id%TYPE ARRAY[4];  -- равнозначно примеру выше
```

Так же, как и в случае с объявлением столбцов таблицы, являющихся массивом, неважно, используется ли несколько пар квадратных скобок или конкретная размерность массива, — Postgres Pro рассматривает все массивы заданного типа элемента как массивы одного типа, вне зависимости от размерности. (См. Подраздел 8.15.1.)

Используя `%TYPE`, не нужно знать тип данных структуры, на которую вы ссылаетесь. И самое главное, если в будущем тип данных изменится (например: тип данных для `user_id` поменяется с `integer` на `real`), то вам может не понадобиться изменять определение функции.

Использование `%TYPE` особенно полезно в полиморфных функциях, поскольку типы данных, необходимые для внутренних переменных, могут меняться от одного вызова к другому. Соответствующие переменные могут быть созданы с применением `%TYPE` к аргументам и возвращаемому значению функции.

41.3.4. Типы кортежей

имя *имя_таблицы*`%ROWTYPE;`
имя *имя_составного_типа*`;`

Переменная составного типа называется *переменной-кортежем* (или переменной *типа кортежа*). Значением такой переменной может быть целый кортеж, полученный в результате выполнения запроса `SELECT` или `FOR`, при условии, что набор столбцов запроса соответствует заявленному типу переменной. Доступ к отдельным полям значения кортежа осуществляется как обычно, через точку, например `rowvar.field`.

Переменная-кортеж может быть объявлена с таким же типом, как и строка в существующей таблице или представлении, используя нотацию *имя_таблицы*`%ROWTYPE`; или с именем составного типа. (Поскольку каждая таблица имеет соответствующий составной тип с таким же именем, то на самом деле в Postgres Pro не имеет значения, пишете ли вы `%ROWTYPE` или нет. Но использование `%ROWTYPE` более переносимо.)

Как и в случае с `%TYPE`, после `%ROWTYPE` также может следовать оформление массива, которое объявляет переменную, содержащую массив составного типа, на который она ссылается.

Параметры функции могут быть составного типа (строки таблицы). В этом случае соответствующий идентификатор `$n` будет переменной-кортежем, поля которой можно выбирать, например `$1.user_id`.

Ниже приведён пример использования составных типов. `table1` и `table2` это существующие таблицы, имеющие, по меньшей мере, перечисленные столбцы:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;

SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

41.3.5. Тип `record`

имя `RECORD;`

Переменные типа `record` похожи на переменные-кортежи, но они не имеют предопределённой структуры. Они приобретают фактическую структуру от строки, которая им присваивается командами `SELECT` или `FOR`. Структура переменной типа `record` может меняться каждый раз при присваивании значения. Следствием этого является то, что пока значение не присвоено первый раз, переменная типа `record` не имеет структуры и любая попытка получить доступ к отдельному полю приведёт к ошибке во время выполнения.

Обратите внимание, что `RECORD` это не подлинный тип данных, а только лишь заполнитель. Также следует понимать, что функция на PL/pgSQL, имеющая тип возвращаемого значения `record`, это не то же самое, что и переменная типа `record`, хотя такая функция может использовать переменную типа `record` для хранения своего результата. В обоих случаях фактическая структура строки неизвестна во время создания функции, но для функции, возвращающей `record`, фактическая структура определяется во время разбора вызывающего запроса, в то время как переменная типа `record` может менять свою структуру на лету.

41.3.6. Упорядочение переменных PL/pgSQL

Когда функция на PL/pgSQL имеет один или несколько параметров сортируемых типов данных, правило сортировки определяется при каждом вызове функции в зависимости от правил сортировки фактических аргументов, как описано в Разделе 22.2. Если оно определено успешно (т. е. среди аргументов нет конфликтов между неявными правилами сортировки), то все соответствующие параметры неявно трактуются как имеющее это правило сортировки. Внутри функции это будет влиять на поведение операторов, зависящих от используемого правила сортировки. Рассмотрим пример:

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

В первом случае `less_than` будет использовать для сравнения общее правило сортировки для `text_field_1` и `text_field_2`, в то время как во втором случае будет использоваться правило `C`.

Кроме того, определённое для вызова функции правило сортировки также будет использоваться для любых локальных переменных соответствующего типа. Таким образом, функция не станет работать по-другому, если её переписать так:

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

Если параметров с типами данных, поддерживающими сортировку, нет, или для параметров невозможно определить общее правило сортировки, тогда для параметров и локальных переменных применяются правила, принятые для их типа данных по умолчанию (которые обычно совпадают с правилами сортировки по умолчанию, принятыми для базы данных, но могут отличаться для переменных доменных типов).

Локальная переменная может иметь правило сортировки, отличное от правила по умолчанию. Для этого используется параметр `COLLATE` в объявлении переменной, например:

```
DECLARE
```

```
    local_a text COLLATE "en_US";
```

Этот параметр переопределяет правило сортировки, которое получила бы переменная в соответствии с вышеуказанными правилами.

И, конечно же, можно явно указывать параметр `COLLATE` для конкретных операций внутри функции, если к ним требуется применить конкретное правило сортировки. Например:

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

Как и в обычной SQL-команде, это переопределяет правила сортировки, связанные с полями таблицы, параметрами и локальными переменными, которые используются в данном выражении.

41.4. Выражения

Все выражения, используемые в операторах PL/pgSQL, обрабатываются основным исполнителем SQL-сервера. Например, для вычисления такого выражения:

```
IF выражение THEN ...
```

PL/pgSQL отправит следующий запрос исполнителю SQL:

```
SELECT выражение
```

При формировании команды `SELECT` все вхождения имён переменных PL/pgSQL заменяются параметрами запроса, как подробно описано в Подразделе 41.11.1. Это позволяет один раз подготовить план выполнения команды `SELECT` и повторно использовать его в последующих вычислениях с различными значениями переменных. Таким образом, при первом использовании выражения, по сути происходит выполнение команды `PREPARE`. Например, если мы объявили две целочисленные переменные `x` и `y`, и написали:

```
IF x < y THEN ...
```

то, что реально происходит за сценой, эквивалентно:

```
PREPARE имя_оператора(integer, integer) AS SELECT $1 < $2;
```

и затем, эта подготовленная команда исполняется (`EXECUTE`) для каждого оператора `IF` с текущими значениями переменных PL/pgSQL, переданных как значения параметров. Обычно эти детали не важны для пользователей PL/pgSQL, но их полезно знать при диагностировании проблем. Более подробно об этом рассказывается в Подразделе 41.11.2.

Поскольку *`выражение`* преобразуется в команду `SELECT`, оно может содержать те же предложения, что и обычная команда `SELECT`, но не может включать предложения `UNION`, `INTERSECT` или `EXCEPT` на верхнем уровне запроса. Например, так можно проверить, является ли таблица непустой:

```
IF count(*) > 0 FROM my_table THEN ...
```

поскольку *`выражение`* между `IF` и `THEN` разбирается, как если бы это была команда `SELECT count(*) > 0 FROM my_table`. Команда `SELECT` должна выдавать один столбец и не более одной строки. (Если она не возвращает ни одной строки, считается, что возвращается NULL.)

41.5. Основные операторы

В этом и последующих разделах описаны все типы операторов, которые понимает PL/pgSQL. Все, что не признается в качестве одного из этих типов операторов, считается командой SQL и отправляется для исполнения в основную машину базы данных, как описано в Подразделе 41.5.2.

41.5.1. Присваивания

Присваивание значения переменной PL/pgSQL записывается в виде:

```
переменная { := | = } выражение;
```

Как описывалось ранее, выражение в таком операторе вычисляется с помощью SQL-команды `SELECT`, посылаемой в основную машину базы данных. Выражение должно получить одно значение (возможно, значение строки, если это переменная-кортеж или переменная типа `record`). Целевая переменная может быть простой переменной (возможно, дополненной именем блока); полем в целевом кортеже или записи; или элементом или срезом целевого массива. Для присваивания можно использовать знак равенства (=) вместо совместимого с PL/SQL `:=`.

Если тип данных результата выражения не соответствует типу данных переменной, это значение будет преобразовано к нужному типу с использованием приведения присваивания (см. Раздел 10.4). В случае отсутствия приведения присваивания для этой пары типов, интерпретатор PL/pgSQL попытается преобразовать значение результата через текстовый формат, то есть применив функцию вывода типа результата, а за ней функцию ввода типа переменной. Заметьте, что при этом функция ввода может выдавать ошибки времени выполнения, если не воспримет строковое представление значения результата.

Примеры:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
my_array[j] := 20;
my_array[1:3] := array[1,2,3];
complex_array[n].realpart = 12.3;
```

41.5.2. Выполнение команд SQL

Вообще говоря, в функции на PL/pgSQL можно выполнить любую команду SQL, не возвращающую строк, просто написав эту команду. Например, можно создать таблицу и заполнить её данными, написав

```
CREATE TABLE mytable (id int primary key, data text);
INSERT INTO mytable VALUES (1,'one'), (2,'two');
```

Если команда всё же возвращает строки (например, `SELECT` или `INSERT`/`UPDATE`/`DELETE`/`MERGE` с предложением `RETURNING`), есть два варианта действий. Если команда возвращает максимум одну строку или вам интересна только первая строка результата, напишите команду как обычно, но добавьте предложение `INTO` для захвата вывода, как описано в Подразделе 41.5.3. Чтобы обрабатывать все результирующие строки, запишите команду в качестве источника данных для цикла `FOR`, как описано в Подразделе 41.6.6.

Обычно недостаточно выполнять только статически определённые SQL-команды. Как правило, желательно, чтобы в команде использовались различные значения данных или она менялась более кардинально, например использовала разные имена таблиц от запуска к запуску. Это можно сделать двумя способами.

Значения переменных PL/pgSQL могут автоматически вставляться в оптимизируемые SQL-команды: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` и определённые служебные команды, содержащие вышеперечисленные, например `EXPLAIN` и `CREATE TABLE ... AS SELECT`. Имя любой переменной PL/pgSQL в текстах этих команд рассматривается как параметр, и значение переменной подставляется в качестве значения параметра во время выполнения. Это в точности совпадает с описанной ранее обработкой для выражений; за подробностями обратитесь к Подразделу 41.11.1.

При выполнении оптимизируемой SQL-команды таким образом, PL/pgSQL может кешировать и повторно использовать план выполнения команды, как обсуждается в Подразделе 41.11.2.

Неоптимизируемые SQL-команды (также называемые служебными командами) не принимают параметры запроса. Поэтому в таких командах не работает автоматическая замена переменных PL/pgSQL. Чтобы включить изменяемый текст в служебную команду, запускаемую из PL/pgSQL, необходимо составить текст команды в виде строки и выполнить её в `EXECUTE`, как описано в Подразделе 41.5.4.

Если нужно изменять команду, не просто передавая ей разные значения данных, а например меняя имя таблицы, также нужно использовать оператор `EXECUTE`.

Иногда бывает полезно вычислить значение выражения или запроса `SELECT`, но отказаться от результата, например, при вызове функции, у которой есть побочные эффекты, но нет полезного результата. Для этого в PL/pgSQL, используется оператор `PERFORM`:

```
PERFORM запрос;
```

Эта команда выполняет *запрос* и отбрасывает результат. *Запросы* пишутся таким же образом, как и в команде SQL `SELECT`, но ключевое слово `SELECT` заменяется на `PERFORM`. Для запросов `WITH` после `PERFORM` нужно поместить запрос в скобки. (В этом случае запрос может вернуть только одну строку.) Переменные PL/pgSQL будут подставлены в запрос так же, как описано выше, план запроса также кешируется. Кроме того, специальная переменная `FOUND` принимает значение true, если запрос возвращает, по крайней мере, одну строку, или false, если не возвращает ни одной строки (см. Подраздел 41.5.5).

Примечание

Можно предположить, что такой же результат получается непосредственно командой `SELECT`, но в настоящее время использование `PERFORM` является единственным способом. Команда SQL, которая может возвращать строки, например `SELECT`, будет отклонена с ошибкой, если не имеет предложения `INTO`, как описано в следующем разделе.

Пример:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

41.5.3. Выполнение команды, возвращающей одну строку

Результат SQL-команды, возвращающей одну строку (возможно из нескольких столбцов), может быть присвоен переменной типа `record`, переменной-кортежу или списку скалярных переменных. Для этого нужно к основной команде SQL добавить предложение `INTO`. Так, например:

```
SELECT выражения_select INTO [STRICT] цель FROM ...;
INSERT ... RETURNING выражения INTO [STRICT] цель;
UPDATE ... RETURNING выражения INTO [STRICT] цель;
DELETE ... RETURNING выражения INTO [STRICT] цель;
MERGE ... RETURNING выражения INTO [STRICT] цель;
```

где *цель* может быть переменной типа `record`, переменной-кортежем или разделённым запятыми списком скалярных переменных, полей записи/строки. Переменные PL/pgSQL подставляются в оставшуюся часть команды (то есть везде, кроме предложения `INTO`), как было описано выше, и план выполнения кешируется так же. Это работает для команд `SELECT`, `INSERT`/`UPDATE`/`DELETE`/`MERGE` с предложением `RETURNING` и определённых служебных команд, возвращающих результат в виде набора строк (таких как `EXPLAIN`). За исключением предложения `INTO`, это те же SQL-команды, как их можно написать вне PL/pgSQL.

Подсказка

Обратите внимание, что данная интерпретация `SELECT` с `INTO` полностью отличается от Postgres Pro команды `SELECT INTO`, где в `INTO` указывается вновь создаваемая таблица. Если вы хотите в функции на PL/pgSQL создать таблицу, основанную на результате команды `SELECT`, используйте синтаксис `CREATE TABLE ... AS SELECT`.

Если результат команды присваивается переменной-кортежу или списку переменных, то они должны в точности соответствовать по количеству и типам данных столбцам результата, иначе произойдёт ошибка во время выполнения. Если используется переменная типа `record`, то она автоматически приводится к типу строки результата команды.

Предложение INTO может появиться практически в любом месте SQL-команды. Обычно его записывают непосредственно перед или сразу после списка *`выражения_select`* в `SELECT` или в конце команды для команд других типов. Рекомендуется следовать этому соглашению на случай, если правила разбора PL/pgSQL ужесточатся в будущих версиях.

Если указание `STRICT` отсутствует в предложении `INTO`, то *`цели`* присваивается первая строка, возвращённая командой; или NULL, если команда не вернула строки. (Заметим, что понятие «первая строка» определяется неоднозначно без `ORDER BY`.) Все остальные строки результата после первой отбрасываются. Можно проверить специальную переменную `FOUND` (см. Подраздел 41.5.5), чтобы определить, была ли возвращена запись:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'Сотрудник % не найден', myname;
END IF;
```

Если добавлено указание `STRICT`, то команда должна вернуть ровно одну строку или произойдёт ошибка во время выполнения: либо `NO_DATA_FOUND` (нет строк), либо `TOO_MANY_ROWS` (более одной строки). Можно использовать секцию исключений в блоке для обработки ошибок, например:

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'Сотрудник % не найден', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'Сотрудник % уже существует', myname;
END;
```

После успешного выполнения команды с указанием `STRICT`, значение переменной `FOUND` всегда принимает значение true.

Для `INSERT`/`UPDATE`/`DELETE`/`MERGE` с `RETURNING`, PL/pgSQL возвращает ошибку, если выбрано более одной строки, даже в том случае, когда указание `STRICT` отсутствует. Так происходит потому, что у этих команд нет возможности, типа `ORDER BY`, указать какая из задействованных строк должна быть возвращена.

Если для функции включён режим `print_strict_params`, то при возникновении ошибки, связанной с нарушением условия `STRICT`, в детальную (`DETAIL`) часть сообщения об ошибке будет включена информация о параметрах, переданных команде. Изменить значение `print_strict_params` можно установкой параметра `plpgsql.print_strict_params`. Но это повлияет только на функции, скомпилированные после изменения. Для конкретной функции можно использовать указание компилятора, например:

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END;
$$ LANGUAGE plpgsql;
```

В случае сбоя будет сформировано примерно такое сообщение об ошибке

```
ERROR:  query returned no rows
```

```
DETAIL:  parameters: username = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

Примечание

С указанием `STRICT` поведение `SELECT INTO` и связанных операторов соответствует принятому в Oracle PL/SQL.

41.5.4. Выполнение динамически формируемых команд

Часто требуется динамически формировать команды внутри функций на PL/pgSQL, то есть такие команды, в которых при каждом выполнении могут использоваться разные таблицы или типы данных. Обычно PL/pgSQL кеширует планы выполнения (как описано в Подразделе 41.11.2), но в случае с динамическими командами это не будет работать. Для исполнения динамических команд предусмотрен оператор `EXECUTE`:

```
EXECUTE строка-команды [ INTO [STRICT] цель ] [ USING выражение [, ... ] ];
```

где *`строка-команды`* это выражение, формирующее строку (типа `text`) с текстом команды, которую нужно выполнить. Необязательная *`цель`* — это переменная-запись, переменная-кортеж или разделённый запятыми список простых переменных и полей записи/кортежа, куда будут помещены результаты команды. Необязательные выражения в `USING` формируют значения, которые будут вставлены в команду.

В сформированном тексте команды замена имён переменных PL/pgSQL на их значения проводиться не будет. Все необходимые значения переменных должны быть вставлены в командную строку при её построении, либо нужно использовать параметры, как описано ниже.

Также нет никакого кеширования плана для команд, выполняемых с помощью `EXECUTE`. Вместо этого план создаётся каждый раз при выполнении. Таким образом, строка команды может динамически создаваться внутри функции для выполнения действий с различными таблицами и столбцами.

Предложение `INTO` указывает, куда должны быть помещены результаты SQL-команды, возвращающей строки. Если используется переменная-кортеж или список переменных, то они должны в точности соответствовать структуре результата команды; если используется переменная типа `record`, она автоматически приводится к типу строки результата запроса. Если возвращается несколько строк, то только первая будет присвоена переменной(ым) в `INTO`. Если не возвращается ни одной строки, то присваивается NULL. Без предложения `INTO` результаты команды отбрасываются.

С указанием `STRICT` команда должна вернуть ровно одну строку, иначе выдаётся сообщение об ошибке.

В тексте команды можно использовать значения параметров, ссылки на параметры обозначаются как `$1`, `$2` и т. д. Эти символы указывают на значения, находящиеся в предложении `USING`. Такой метод зачастую предпочтительнее, чем вставка значений в команду в виде текста: он позволяет исключить во время выполнения дополнительные расходы на преобразования значений в текст и обратно, и не открывает возможности для SQL-инъекций, не требуя применять экранирование или кавычки для спецсимволов. Пример:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

Обратите внимание, что символы параметров можно использовать только вместо значений данных. Если же требуется динамически формировать имена таблиц или столбцов, их необходимо вставлять в виде текста. Например, если в предыдущем запросе необходимо динамически задавать имя таблицы, можно сделать следующее:

```
EXECUTE 'SELECT count(*) FROM '
```

```
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

Более аккуратным решением будет использование указания `%I` с функцией `format()`, позволяющее вставить имя таблицы или столбца, которое будет автоматически заключено в кавычки:

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

(Этот пример задействует SQL-правило, согласно которому строковые значения, разделённые символом новой строки, соединяются вместе.)

Ещё одно ограничение состоит в том, что символы параметров могут использоваться только в оптимизируемых SQL-командах (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` и некоторых командах, содержащих одну из них). В операторы других типов (обычно называемые служебными) значения нужно вставлять в текстовом виде, даже если это просто значения данных.

Команда `EXECUTE` с неизменяемым текстом и параметрами USING (как в первом примере выше), функционально эквивалентна команде, записанной напрямую в PL/pgSQL, в которой переменные PL/pgSQL автоматически заменяются значениями. Важное отличие в том, что `EXECUTE` при каждом исполнении заново строит план команды с учётом текущих значений параметров, тогда как PL/pgSQL строит общий план выполнения и кеширует его при повторном использовании. В тех случаях, когда наилучший план выполнения сильно зависит от значений параметров, может быть полезно использовать `EXECUTE` для гарантии того, что не будет выбран общий план.

В настоящее время команда `SELECT INTO` не поддерживается в `EXECUTE`, вместо этого нужно выполнять обычный `SELECT` и указать `INTO` для самой команды `EXECUTE`.

Примечание

Оператор `EXECUTE` в PL/pgSQL не имеет отношения к одноимённому SQL-оператору сервера Postgres Pro. Серверный `EXECUTE` не может напрямую использоваться в функциях на PL/pgSQL (и в этом нет необходимости).

Пример 41.1. Использование кавычек в динамических запросах

При работе с динамическими командами часто приходится иметь дело с экранированием одинарных кавычек. Рекомендуемым методом для взятия текста в кавычки в теле функции является экранирование знаками доллара. (Если имеется унаследованный код, не использующий этот метод, пожалуйста, обратитесь к обзору в Подразделе 41.12.1, это поможет сэкономить усилия при переводе кода к более приемлемому виду.)

Динамические значения требуют особого внимания, так как они могут содержать апострофы. Например, можно использовать функцию `format()` (предполагается, что тело функции заключается в доллары, так что апострофы дублировать не нужно):

```
EXECUTE format('UPDATE tbl SET %I = $1 '
   'WHERE key = $2', colname) USING newvalue, keyvalue;
```

Также можно напрямую вызывать функции заключения в кавычки:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
```

```
        || quote_literal(keyvalue);
```

Этот пример демонстрирует использование функций `quote_ident` и `quote_literal` (см. Раздел 9.4). Для надёжности, выражения, содержащие идентификаторы столбцов и таблиц должны использовать функцию `quote_ident` при добавлении в текст запроса. А для выражений со значениями, которые должны быть обычными строками, используется функция `quote_literal`. Эти функции выполняют соответствующие шаги, чтобы вернуть текст, заключённый в двойные или одинарные кавычки соответственно и с правильно экранированными специальными символами.

Так как функция `quote_literal` помечена как `STRICT`, то она всегда возвращает NULL, если переданный ей аргумент имеет значение NULL. В приведённом выше примере, если `newvalue` или `keyvalue` были NULL, вся строка с текстом запроса станет NULL, что приведёт к ошибке в `EXECUTE`. Для предотвращения этой проблемы используйте функцию `quote_nullable`, которая работает так же, как `quote_literal` за исключением того, что при вызове с пустым аргументом возвращает строку 'NULL'. Например:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

Если вы имеете дело со значениями, которые могут быть пустыми, то, как правило, нужно использовать `quote_nullable` вместо `quote_literal`.

Как обычно, необходимо убедиться, что значения NULL в запросе не принесут неожиданных результатов. Например, следующее условие `WHERE`

```
'WHERE key = ' || quote_nullable(keyvalue)
```

никогда не выполнится, если `keyvalue` — NULL, так как применение `=` с операндом, имеющим значение NULL, всегда даёт NULL. Если требуется, чтобы NULL обрабатывалось как обычное значение, то условие выше нужно переписать так:

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(В настоящее время `IS NOT DISTINCT FROM` работает менее эффективно, чем `=`, так что используйте этот способ, только если это действительно необходимо. Подробнее особенности NULL и `IS DISTINCT` описаны в Разделе 9.2.)

Обратите внимание, что использование знака $ полезно только для взятия в кавычки фиксированного текста. Плохая идея написать этот пример так:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
        || quote_literal(keyvalue);
```

потому что `newvalue` может также содержать `$$`. Эта же проблема может возникнуть и с любым другим разделителем, используемым после знака `$`. Поэтому, чтобы безопасно заключить заранее неизвестный текст в кавычки, *нужно* использовать соответствующие функции: `quote_literal`, `quote_nullable`, или `quote_ident`.

Динамические операторы SQL также можно безопасно сформировать, используя функцию `format` (см. Подраздел 9.4.1). Например:

```
EXECUTE format('UPDATE tbl SET %I = %L '
   'WHERE key = %L', colname, newvalue, keyvalue);
```

Указание `%I` равнозначно вызову `quote_ident`, а `%L` — вызову `quote_nullable`. Функция `format` может применяться в сочетании с предложением `USING`:

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

Эта форма лучше, так как с ней переменные обрабатываются в их собственном формате данных, а не преобразуются безусловно в текст, чтобы затем выводиться с использованием `%L`. Она также и более эффективна.

Более объёмный пример использования динамической команды и `EXECUTE` можно увидеть в Примере 41.10. В нём создаётся и динамически выполняется команда `CREATE FUNCTION` для определения новой функции.

41.5.5. Статус выполнения команды

Определить результат команды можно несколькими способами. Во-первых, можно воспользоваться командой `GET DIAGNOSTICS`, имеющей форму:

```
GET [ CURRENT ] DIAGNOSTICS переменная { = | := } элемент [ , ... ];
```

Эта команда позволяет получить системные индикаторы состояния. Слово `CURRENT` не несёт смысловой нагрузки (но см. также описание `GET STACKED DIAGNOSTICS` в Подразделе 41.6.8.1). Каждый *элемент* представляется ключевым словом, указывающим, какое значение состояния нужно присвоить заданной *переменной* (она должна иметь подходящий тип данных, чтобы принять его). Доступные в настоящее время элементы состояния показаны в Таблице 41.1. Вместо принятого в стандарте SQL присваивания (=) можно применять присваивание с двоеточием (:=). Например:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

Таблица 41.1. Доступные элементы диагностики

Имя	Тип	Описание
`ROW_COUNT`	`bigint`	число строк, обработанных последней командой SQL
`PG_CONTEXT`	`text`	строки текста, описывающие текущий стек вызовов (см. Подраздел 41.6.9)
`PG_ROUTINE_OID`	`oid`	OID текущей функции

Второй способ определения статуса выполнения команды заключается в проверке значения специальной переменной `FOUND`, имеющей тип `boolean`. При вызове функции на PL/pgSQL, переменная `FOUND` инициализируется в ложь. Далее, значение переменной изменяется следующими операторами:

- `SELECT INTO` записывает в `FOUND` true, если строка присвоена, или false, если строки не были получены.
- `PERFORM` записывает в `FOUND` true, если строки выбраны (и отброшены) или false, если строки не выбраны.
- `UPDATE`, `INSERT`, `DELETE` и `MERGE` записывают в `FOUND` true, если при их выполнении была задействована хотя бы одна строка, или false, если ни одна строка не была задействована.
- `FETCH` записывают в `FOUND` true, если команда вернула строку, или false, если строка не выбрана.
- `MOVE` записывают в `FOUND` true при успешном перемещении курсора, в противном случае — false.
- `FOR`, как и `FOREACH`, записывает в `FOUND` true, если была произведена хотя бы одна итерация цикла, в противном случае — false. При этом значение `FOUND` будет установлено только после выхода из цикла. Пока цикл выполняется, оператор цикла не изменяет значение переменной. Но другие операторы внутри цикла могут менять значение `FOUND`.
- `RETURN QUERY` и `RETURN QUERY EXECUTE` записывают в `FOUND` true, если запрос вернул хотя бы одну строку, или false, если строки не выбраны.

Другие операторы PL/pgSQL не меняют значение `FOUND`. Помните в частности, что `EXECUTE` изменяет вывод `GET DIAGNOSTICS`, но не меняет `FOUND`.

`FOUND` является локальной переменной в каждой функции PL/pgSQL и любые её изменения, влияют только на текущую функцию.

41.5.6. Не делать ничего

Иногда бывает полезен оператор, который не делает ничего. Например, он может показывать, что одна из ветвей if/then/else сознательно оставлена пустой. Для этих целей используется `NULL`:

```
NULL;
```

В следующем примере два фрагмента кода эквивалентны:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- ошибка игнорируется
END;
```

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- ошибка игнорируется
END;
```

Какой вариант выбрать — дело вкуса.

Примечание

В Oracle PL/SQL не допускаются пустые списки операторов, поэтому `NULL` *обязателен* в подобных ситуациях. В PL/pgSQL разрешается не писать ничего.

41.6. Управляющие структуры

Управляющие структуры, вероятно, наиболее полезная и важная часть PL/pgSQL. С их помощью можно очень гибко и эффективно манипулировать данными Postgres Pro.

41.6.1. Команды, возвращающие значения из функции

Две команды позволяют вернуть данные из функции: `RETURN` и `RETURN NEXT`.

41.6.1.1. `RETURN`

```
RETURN выражение;
```

`RETURN` с последующим выражением прекращает выполнение функции и возвращает значение выражения в вызывающую программу. Эта форма используется для функций PL/pgSQL, которые не возвращают набор строк.

В функции, возвращающей скалярный тип, результирующее выражение автоматически приводится к типу возвращаемого значения. Однако если возвращаемый тип —составной (строка), возвращаемое выражение должно в точности содержать требуемый набор столбцов. При этом может потребоваться явное приведение типов.

Для функции с выходными параметрами просто используйте `RETURN` без выражения. Будут возвращены текущие значения выходных параметров.

Для функции, возвращающей `void`, `RETURN` можно использовать в любом месте, но без выражения после `RETURN`.

Возвращаемое значение функции не может остаться не определённым. Если достигнут конец блока верхнего уровня, а оператор `RETURN` так и не встретился, происходит ошибка времени выполнения. Это не касается функций с выходными параметрами и функций, возвращающих `void`. Для них оператор `RETURN` выполняется автоматически по окончании блока верхнего уровня.

Несколько примеров:

```
-- Функции, возвращающие скалярный тип данных
RETURN 1 + 2;
RETURN scalar_var;

-- Функции, возвращающие составной тип данных
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- требуется приведение типов
```

41.6.1.2. RETURN NEXT и RETURN QUERY

```
RETURN NEXT выражение;
RETURN QUERY query;
RETURN QUERY EXECUTE строка-команды [USING выражение [, ...]];
```

Для функций на PL/pgSQL, возвращающих `SETOF некий_тип`, нужно действовать несколько по-иному. Отдельные элементы возвращаемого значения формируются командами `RETURN NEXT` или `RETURN QUERY`, а финальная команда `RETURN` без аргументов завершает выполнение функции. `RETURN NEXT` используется как со скалярными, так и с составными типами данных. Для составного типа результат функции возвращается в виде таблицы. `RETURN QUERY` добавляет результат выполнения запроса к результату функции. `RETURN NEXT` и `RETURN QUERY` можно свободно смешивать в теле функции, в этом случае их результаты будут объединены.

`RETURN NEXT` и `RETURN QUERY` не выполняют возврат из функции. Они просто добавляют строки в результирующее множество. Затем выполнение продолжается со следующего оператора в функции. Успешное выполнение `RETURN NEXT` и `RETURN QUERY` формирует множество строк результата. Для выхода из функции используется `RETURN`, обязательно без аргументов (или можно просто дождаться окончания выполнения функции).

`RETURN QUERY` имеет разновидность `RETURN QUERY EXECUTE`, предназначенную для динамического выполнения запроса. В текст запроса можно добавить параметры, используя `USING`, так же как и с командой `EXECUTE`.

Для функции с выходными параметрами просто используйте `RETURN NEXT` без аргументов. При каждом исполнении `RETURN NEXT` текущие значения выходных параметров сохраняются для последующего возврата в качестве строки результата. Обратите внимание, что если функция с выходными параметрами должна возвращать множество значений, то при объявлении нужно указывать `RETURNS SETOF`. При этом если выходных параметров несколько, то используется `RETURNS SETOF record`, а если только один с типом *некий_тип*, то `RETURNS SETOF некий_тип`.

Пример использования `RETURN NEXT`:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
```

```
        -- здесь возможна обработка данных
        RETURN NEXT r; -- возвращается текущая строка запроса
    END LOOP;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

Пример использования `RETURN QUERY`:

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- Так как выполнение ещё не закончено, можно проверить, были ли возвращены строки,
    -- и выдать исключение, если нет.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Нет рейсов на дату: %.', $1;
    END IF;

    RETURN;
 END;
$BODY$
LANGUAGE plpgsql;

-- Возвращает доступные рейсы либо вызывает исключение, если таковых нет.
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

Примечание

В текущей реализации `RETURN NEXT` и `RETURN QUERY` результирующее множество накапливается целиком, прежде чем будет возвращено из функции. Если множество очень большое, то это может отрицательно сказаться на производительности, так как при нехватке оперативной памяти данные записываются на диск. В следующих версиях PL/pgSQL это ограничение будет снято. В настоящее время управлять количеством оперативной памяти в подобных случаях можно параметром конфигурации work_mem. При наличии свободной памяти администраторы должны рассмотреть возможность увеличения значения данного параметра.

41.6.2. Завершение процедуры

Процедура не возвращает никакого значения, поэтому она может завершаться без оператора `RETURN`. Если вы хотите досрочно завершить выполнение кода оператором `RETURN`, напишите просто `RETURN` без возвращаемого выражения.

Если у процедуры есть выходные параметры, конечные значения соответствующих им переменных будут выданы вызывающему коду.

41.6.3. Вызов процедуры

Функция, процедура или блок `DO` в PL/pgSQL может вызвать процедуру, используя оператор `CALL`. Выходные параметры при этом обрабатываются не так, как это делает `CALL` в обычном SQL. Каждому параметру `OUT` или `INOUT` для процедуры должна соответствовать переменная в операторе

`CALL`, и этой переменной по завершении процедуры будет присвоено возвращаемое процедурой значение. Например:

```
CREATE PROCEDURE triple(INOUT x int)
LANGUAGE plpgsql
AS $$
BEGIN
    x := x * 3;
END;
$$;

DO $$
DECLARE myvar int := 5;
BEGIN
  CALL triple(myvar);
  RAISE NOTICE 'myvar = %', myvar;  -- выводится 15
END;
$$;
```

Переменная, соответствующая выходному параметру, может быть простой переменной или полем переменной составного типа. В настоящее время она не может быть элементом массива.

41.6.4. Условные операторы

Операторы `IF` и `CASE` позволяют выполнять команды в зависимости от определённых условий. PL/pgSQL поддерживает три формы `IF`:

- `IF ... THEN ... END IF`
- `IF ... THEN ... ELSE ... END IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`

и две формы `CASE`:

- `CASE ... WHEN ... THEN ... ELSE ... END CASE`
- `CASE WHEN ... THEN ... ELSE ... END CASE`

41.6.4.1. `IF-THEN`

```
IF логическое-выражение THEN
    операторы
END IF;
```

`IF-THEN` это простейшая форма `IF`. Операторы между `THEN` и `END IF` выполняются, если условие (*`логическое-выражение`*) истинно. В противном случае они опускаются.

Пример:

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

41.6.4.2. `IF-THEN-ELSE`

```
IF логическое-выражение THEN
    операторы
ELSE
    операторы
END IF;
```

`IF-THEN-ELSE` добавляет к `IF-THEN` возможность указать альтернативный набор операторов, которые будут выполнены, если условие не истинно (в том числе, если условие NULL).

Примеры:

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

41.6.4.3. `IF-THEN-ELSIF`

```
IF логическое-выражение THEN
    операторы
[ELSIF логическое-выражение THEN операторы [ELSIF логическое-выражение THEN операторы
 ...]]
[ELSE операторы]
END IF;
```

В некоторых случаях двух альтернатив недостаточно. `IF-THEN-ELSIF` обеспечивает удобный способ проверки нескольких вариантов по очереди. Условия в `IF` последовательно проверяются до тех пор, пока не будет найдено первое истинное. После этого операторы, относящиеся к этому условию, выполняются, и управление переходит к следующей после `END IF` команде. (Все последующие условия не проверяются.) Если ни одно из условий `IF` не является истинным, то выполняется блок `ELSE` (если присутствует).

Пример:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- остаётся только один вариант: number имеет значение NULL
    result := 'NULL';
END IF;
```

Вместо ключевого слова `ELSIF` можно использовать `ELSEIF`.

Другой вариант сделать то же самое, это использование вложенных операторов `IF-THEN-ELSE`, как в следующем примере:

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

Однако это требует написания соответствующих `END IF` для каждого `IF`, что при наличии нескольких альтернатив делает код более громоздким, чем использование `ELSIF`.

41.6.4.4. Простой `CASE`

```
CASE выражение-поиска
```

```
    WHEN выражение [, выражение [...]] THEN
      операторы
  [WHEN выражение [, выражение [...]] THEN операторы ...]
  [ELSE операторы]
END CASE;
```

Простая форма `CASE` реализует условное выполнение на основе сравнения операндов. *`Выражение-поиска`* вычисляется (один раз) и последовательно сравнивается с каждым *`выражением`* в условиях `WHEN`. Если совпадение найдено, то выполняются соответствующие *`операторы`* и управление переходит к следующей после `END CASE` команде. (Все последующие выражения `WHEN` не проверяются.) Если совпадение не было найдено, то выполняются *`операторы`* в `ELSE`. Но если `ELSE` нет, то вызывается исключение `CASE_NOT_FOUND`.

Пример:

```
CASE x
    WHEN 1, 2 THEN
        msg := 'один или два';
    ELSE
        msg := 'значение, отличное от один или два';
END CASE;
```

41.6.4.5. `CASE` с перебором условий

```
CASE
    WHEN логическое-выражение THEN
      операторы
  [WHEN логическое-выражение THEN операторы ...]
  [ELSE операторы]
END CASE;
```

Эта форма `CASE` реализует условное выполнение, основываясь на истинности логических условий. Каждое *`логическое-выражение`* в предложении `WHEN` вычисляется по порядку до тех пор, пока не будет найдено истинное. Затем выполняются соответствующие *`операторы`* и управление переходит к следующей после `END CASE` команде. (Все последующие выражения `WHEN` не проверяются.) Если ни одно из условий не окажется истинным, то выполняются *`операторы`* в `ELSE`. Но если `ELSE` нет, то вызывается исключение `CASE_NOT_FOUND`.

Пример:

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'значение в диапазоне между 0 и 10';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'значение в диапазоне между 11 и 20';
END CASE;
```

Эта форма `CASE` полностью эквивалента `IF-THEN-ELSIF`, за исключением того, что при невыполнении всех условий и отсутствии `ELSE`, `IF-THEN-ELSIF` ничего не делает, а `CASE` вызывает ошибку.

41.6.5. Простые циклы

Операторы `LOOP`, `EXIT`, `CONTINUE`, `WHILE`, `FOR` и `FOREACH` позволяют повторить серию команд в функции на PL/pgSQL.

41.6.5.1. `LOOP`

```
[<<метка>>]
LOOP
    операторы
END LOOP [ метка ];
```

`LOOP` организует безусловный цикл, который повторяется до бесконечности, пока не будет прекращён операторами `EXIT` или `RETURN`. Для вложенных циклов можно использовать *метку* в операторах `EXIT` и `CONTINUE`, чтобы указать, к какому циклу эти операторы относятся.

41.6.5.2. `EXIT`

```
EXIT [ метка ] [WHEN логическое-выражение];
```

Если *метка* не указана, то завершается самый внутренний цикл, далее выполняется оператор, следующий за `END LOOP`. Если *метка* указана, то она должна относиться к текущему или внешнему циклу, или это может быть метка блока. При этом в именованном цикле/блоке выполнение прекращается, а управление переходит к следующему оператору после соответствующего `END`.

При наличии `WHEN` цикл прекращается, только если *логическое-выражение* истинно. В противном случае управление переходит к оператору, следующему за `EXIT`.

`EXIT` можно использовать со всеми типами циклов, не только с безусловным.

Когда `EXIT` используется для выхода из блока, управление переходит к следующему оператору после окончания блока. Обратите внимание, что для выхода из блока нужно обязательно указывать *метку*. `EXIT` без *метки* не позволяет прекратить работу блока. (Это изменение по сравнению с версиями PostgreSQL до 8.4, в которых разрешалось использовать `EXIT` без *метки* для прекращения работы текущего блока.)

Примеры:

```
LOOP
    -- здесь производятся вычисления
    IF count > 0 THEN
        EXIT;  -- выход из цикла
    END IF;
END LOOP;

LOOP
    -- здесь производятся вычисления
    EXIT WHEN count > 0;  -- аналогично предыдущему примеру
END LOOP;

<<ablock>>
BEGIN
    -- здесь производятся вычисления
    IF stocks > 100000 THEN
        EXIT ablock;  -- выход из блока BEGIN
    END IF;
    -- вычисления не будут выполнены, если stocks > 100000
END;
```

41.6.5.3. `CONTINUE`

```
CONTINUE [ метка ] [WHEN логическое-выражение];
```

Если *метка* не указана, то начинается следующая итерация самого внутреннего цикла. То есть все оставшиеся в цикле операторы пропускаются, и управление переходит к управляющему выражению цикла (если есть) для определения, нужна ли ещё одна итерация цикла. Если *метка* присутствует, то она указывает на метку цикла, выполнение которого будет продолжено.

При наличии `WHEN` следующая итерация цикла начинается только тогда, когда *логическое-выражение* истинно. В противном случае управление переходит к оператору, следующему за `CONTINUE`.

`CONTINUE` можно использовать со всеми типами циклов, не только с безусловным.

Примеры:

```
LOOP
    -- здесь производятся вычисления
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- вычисления для count в диапазоне 50 .. 100
END LOOP;
```

41.6.5.4. `WHILE`

```
[<<метка>>]
WHILE логическое-выражение LOOP
    операторы
END LOOP [ метка ];
```

`WHILE` выполняет серию команд до тех пор, пока истинно *логическое-выражение*. Выражение проверяется непосредственно перед каждым входом в тело цикла.

Пример:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- здесь производятся вычисления
END LOOP;

WHILE NOT done LOOP
    -- здесь производятся вычисления
END LOOP;
```

41.6.5.5. `FOR` (целочисленный вариант)

```
[<<метка>>]
FOR имя IN [REVERSE] выражение .. выражение [BY выражение] LOOP
    операторы
END LOOP [ метка ];
```

В этой форме цикла `FOR` итерации выполняются по диапазону целых чисел. Переменная *имя* автоматически определяется с типом `integer` и существует только внутри цикла (если уже существует переменная с таким именем, то внутри цикла она будет игнорироваться). Выражения для нижней и верхней границы диапазона чисел вычисляются один раз при входе в цикл. Если не указано `BY`, то шаг итерации 1, в противном случае используется значение в `BY`, которое вычисляется, опять же, один раз при входе в цикл. Если указано `REVERSE`, то после каждой итерации величина шага вычитается, а не добавляется.

Примеры целочисленного `FOR`:

```
FOR i IN 1..10 LOOP
    -- внутри цикла переменная i будет иметь значения 1,2,3,4,5,6,7,8,9,10
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- внутри цикла переменная i будет иметь значения 10,9,8,7,6,5,4,3,2,1
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- внутри цикла переменная i будет иметь значения 10,8,6,4,2
END LOOP;
```

Если нижняя граница цикла больше верхней границы (или меньше, в случае `REVERSE`), то тело цикла не выполняется вообще. При этом ошибка не возникает.

Если с циклом `FOR` связана *метка*, к целочисленной переменной цикла можно обращаться по имени, указывая эту *метку*.

41.6.6. Цикл по результатам запроса

Другой вариант `FOR` позволяет организовать цикл по результатам запроса. Синтаксис:

```
[ <<метка>> ]
FOR цель IN запрос LOOP
    операторы
END LOOP [ метка ];
```

Переменная *цель* может быть переменной-кортежем, переменной типа `record` или разделённым запятыми списком скалярных переменных. Переменной *цель* последовательно присваиваются строки результата запроса, и для каждой строки выполняется тело цикла. Пример:

```
CREATE FUNCTION refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing all materialized views...';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP

        -- Здесь "mviews" содержит одну запись с информацией о матпредставлении

        RAISE NOTICE 'Refreshing materialized view %.% (owner: %)...',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema,
 mviews.mv_name);
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Если цикл завершается по команде `EXIT`, то последняя присвоенная строка доступна и после цикла.

В качестве *запроса* в этом типе оператора `FOR` может задаваться любая команда SQL, возвращающая строки. Чаще всего это `SELECT`, но также можно использовать и `INSERT`, `UPDATE`, `DELETE` или `MERGE` с предложением `RETURNING`. Кроме того, возможно применение и некоторых служебных команд, например `EXPLAIN`.

Для переменных PL/pgSQL в тексте запроса выполняется подстановка параметров запроса, план запроса кешируется для возможного повторного использования, как подробно описано в Подразделе 41.11.1 и Подразделе 41.11.2.

Ещё одна разновидность этого типа цикла `FOR-IN-EXECUTE`:

```
[ <<метка>> ]
FOR цель IN EXECUTE выражение_проверки [ USING выражение [, ... ] ] LOOP
    операторы
END LOOP [ метка ];
```

Она похожа на предыдущую форму, за исключением того, что текст запроса указывается в виде строкового выражения. Текст запроса формируется и для него строится план выполнения при каждом входе в цикл. Это даёт программисту выбор между скоростью предварительно разобранного запроса и гибкостью динамического запроса, так же, как и в случае с обычным оператором `EXECUTE`. Как и в `EXECUTE`, значения параметров могут быть добавлены в команду с использованием `USING`.

Ещё один способ организовать цикл по результатам запроса это объявить курсор. Описание в Подразделе 41.7.4.

41.6.7. Цикл по элементам массива

Цикл `FOREACH` очень похож на `FOR`. Отличие в том, что вместо перебора строк SQL-запроса происходит перебор элементов массива. (В целом, `FOREACH` предназначен для перебора выражений составного типа. Варианты реализации цикла для работы с прочими составными выражениями помимо массивов могут быть добавлены в будущем.) Синтаксис цикла `FOREACH`:

```
[ <<метка>> ]
FOREACH цель [ SLICE число ] IN ARRAY выражение LOOP
    операторы
END LOOP [ метка ];
```

Без указания `SLICE`, или если `SLICE` равен 0, цикл выполняется по всем элементам массива, полученного из *выражения*. Переменной *цель* последовательно присваивается каждый элемент массива и для него выполняется тело цикла. Пример цикла по элементам целочисленного массива:

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Обход элементов проводится в том порядке, в котором они сохранялись, независимо от размерности массива. Как правило, *цель* это одиночная переменная, но может быть и списком переменных, когда элементы массива имеют составной тип (записи). В этом случае переменным присваиваются значения из последовательных столбцов составного элемента массива.

При положительном значении `SLICE` `FOREACH` выполняет итерации по срезам массива, а не по отдельным элементам. Значение `SLICE` должно быть целым числом, не превышающим размерности массива. Переменная *цель* должна быть массивом, который получает последовательные срезы исходного массива, где размерность каждого среза задаётся значением `SLICE`. Пример цикла по одномерным срезам:

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

41.6.8. Обработка ошибок

По умолчанию любая возникающая ошибка прерывает выполнение функции на PL/pgSQL и транзакцию, в которой она выполняется. Использование в блоке секции `EXCEPTION` позволяет перехватывать и обрабатывать ошибки. Синтаксис секции `EXCEPTION` расширяет синтаксис обычного блока:

```
[ <<метка>> ]
[ DECLARE
    объявления ]
BEGIN
    операторы
EXCEPTION
    WHEN условие [ OR условие ... ] THEN
        операторы_обработчика
    [ WHEN условие [ OR условие ... ] THEN
          операторы_обработчика
      ... ]
END;
```

Если ошибок не было, то выполняются все *операторы* блока и управление переходит к следующему оператору после `END`. Но если при выполнении *оператора* происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции `EXCEPTION`. В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено, то выполняются соответствующие *операторы_обработчика* и управление переходит к следующему оператору после `END`. Если исключение не найдено, то ошибка передаётся наружу, как будто секции `EXCEPTION` не было. При этом ошибку можно перехватить в секции `EXCEPTION` внешнего блока. Если ошибка так и не была перехвачена, то обработка функции прекращается.

В качестве *условия* может задаваться одно из имён, перечисленных в Приложении A. Если задаётся имя категории, ему соответствуют все ошибки в данной категории. Специальному имени условия `OTHERS` (другие) соответствуют все типы ошибок, кроме `QUERY_CANCELED` и `ASSERT_FAILURE`. (И эти два типа ошибок можно перехватить по имени, но часто это неразумно.) Имена условий воспринимаются без учёта регистра. Условие ошибки также можно задать кодом `SQLSTATE`; например, эти два варианта равнозначны:

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

Если при выполнении *операторов_обработчика* возникнет новая ошибка, то она не может быть перехвачена в этой секции `EXCEPTION`. Ошибка передаётся наружу и её можно перехватить в секции `EXCEPTION` внешнего блока.

При выполнении команд в секции `EXCEPTION` локальные переменные функции на PL/pgSQL сохраняют те значения, которые были на момент возникновения ошибки. Однако все изменения в базе данных, выполненные в блоке, будут отменены. В качестве примера рассмотрим следующий фрагмент:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
```

```
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'перехватили ошибку division_by_zero';
        RETURN x;
END;
```

При присваивании значения переменной `y` произойдёт ошибка `division_by_zero`. Она будет перехвачена в секции `EXCEPTION`. Оператор `RETURN` вернёт значение `x`, увеличенное на единицу, но изменения сделанные командой `UPDATE` будут отменены. Изменения, выполненные командой `INSERT`, которая предшествует блоку, не будут отменены. В результате, база данных будет содержать `Tom Jones`, а не `Joe Jones`.

> ## Подсказка
>
> Наличие секции `EXCEPTION` значительно увеличивает накладные расходы на вход-выход из блока, поэтому не используйте `EXCEPTION` без надобности.

Пример 41.2. Обработка исключений для команд `UPDATE`/`INSERT`

В этом примере обработка исключений помогает выполнить либо команду `UPDATE`, либо `INSERT`, в зависимости от ситуации. Однако в современных приложениях вместо этого приёма рекомендуется использовать `INSERT` с `ON CONFLICT DO UPDATE`. Данный пример предназначен в первую очередь для демонстрации управления выполнением PL/pgSQL:

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- сначала попытаться изменить запись по ключу
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- записи с таким ключом нет, поэтому её нужно добавить
        -- если параллельно будет вставлена запись с таким же ключом,
        -- произойдёт ошибка уникальности
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- здесь не нужно ничего делать,
            -- просто продолжить цикл, чтобы повторить UPDATE.
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

В этом коде предполагается, что ошибка `unique_violation` вызывается самой командой `INSERT`, а не, скажем, внутренним оператором `INSERT` в функции триггера для этой таблицы. Некорректное поведение также возможно, если в таблице будет несколько уникальных индексов; тогда операция будет повторяться вне зависимости от того, нарушение какого индекса вызвало ошибку. Используя средства, рассмотренные далее, можно сделать код более надёжным, проверяя, что перехвачена именно ожидаемая ошибка.

41.6.8.1. Получение информации об ошибке

При обработке исключений часто бывает необходимым получить детальную информацию о произошедшей ошибке. Для этого в PL/pgSQL есть два способа: использование специальных переменных и команда `GET STACKED DIAGNOSTICS`.

Внутри секции `EXCEPTION` специальная переменная `SQLSTATE` содержит код ошибки, для которой было вызвано исключение (список возможных кодов ошибок приведён в Таблице A.1). Специальная переменная `SQLERRM` содержит сообщение об ошибке, связанное с исключением. Эти переменные являются неопределёнными вне секции `EXCEPTION`.

Также в обработчике исключения можно получить информацию о текущем исключении командой `GET STACKED DIAGNOSTICS`, которая имеет вид:

```
GET STACKED DIAGNOSTICS переменная { = | := } элемент [ , ... ];
```

Каждый *элемент* представляется ключевым словом, указывающим, какое значение состояния нужно присвоить заданной *переменной* (она должна иметь подходящий тип данных, чтобы принять его). Доступные в настоящее время элементы состояния показаны в Таблице 41.2.

Таблица 41.2. Элементы диагностики ошибок

Имя	Тип	Описание
`RETURNED_SQLSTATE`	`text`	код исключения, возвращаемый SQLSTATE
`COLUMN_NAME`	`text`	имя столбца, относящегося к исключению
`CONSTRAINT_NAME`	`text`	имя ограничения целостности, относящегося к исключению
`PG_DATATYPE_NAME`	`text`	имя типа данных, относящегося к исключению
`MESSAGE_TEXT`	`text`	текст основного сообщения исключения
`TABLE_NAME`	`text`	имя таблицы, относящейся к исключению
`SCHEMA_NAME`	`text`	имя схемы, относящейся к исключению
`PG_EXCEPTION_DETAIL`	`text`	текст детального сообщения исключения (если есть)
`PG_EXCEPTION_HINT`	`text`	текст подсказки к исключению (если есть)
`PG_EXCEPTION_CONTEXT`	`text`	строки текста, описывающие стек вызовов в момент исключения (см. Подраздел 41.6.9)

Если исключение не устанавливает значение для идентификатора, то возвращается пустая строка.

Пример:

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
```

```
BEGIN
  -- здесь происходит обработка, которая может вызвать исключение
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

41.6.9. Получение информации о месте выполнения

Команда `GET DIAGNOSTICS`, ранее описанная в Подразделе 41.5.5, получает информацию о текущем состоянии выполнения кода (тогда как команда `GET STACKED DIAGNOSTICS`, рассмотренная ранее, выдаёт информацию о состоянии выполнения в момент предыдущей ошибки). Её элемент состояния `PG_CONTEXT` позволяет определить текущее место выполнения кода. `PG_CONTEXT` возвращает текст с несколькими строками, описывающий стек вызова. В первой строке отмечается текущая функция и выполняемая в данный момент команда `GET DIAGNOSTICS`, а во второй и последующих строках отмечаются функции выше по стеку вызовов. Например:

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Стек вызова ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT outer_func();

NOTICE:  --- Стек вызова ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
           1
(1 row)
```

`GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXT` возвращает похожий стек вызовов, но описывает не текущее место, а место, в котором произошла ошибка.

41.7. Курсоры

Вместо того чтобы сразу выполнять весь запрос, есть возможность настроить курсор, инкапсулирующий запрос, и затем получать результат запроса по нескольку строк за раз. Одна из причин так делать заключается в том, чтобы избежать переполнения памяти, когда результат содержит большое количество строк. (Пользователям PL/pgSQL не нужно об этом беспокоиться, так как циклы `FOR` автоматически используют курсоры, чтобы избежать проблем с памятью.) Более интересным вариантом использования является возврат из функции ссылки на курсор, что позволяет вызывающему получать строки запроса. Это эффективный способ получать большие наборы строк из функций.

41.7.1. Объявление курсорных переменных

Доступ к курсорам в PL/pgSQL осуществляется через курсорные переменные, которые всегда имеют специальный тип данных `refcursor`. Один из способов создать курсорную переменную, просто объявить её как переменную типа `refcursor`. Другой способ заключается в использовании синтаксиса объявления курсора, который в общем виде выглядит так:

```
имя [ [ NO ] SCROLL ] CURSOR [ ( аргументы ) ] FOR запрос;
```

(Для совместимости с Oracle, `FOR` можно заменять на `IS`.) С указанием `SCROLL` курсор можно будет прокручивать назад. При `NO SCROLL` прокрутка назад не разрешается. Если ничего не указано, то возможность прокрутки назад зависит от запроса. Если указаны *`аргументы`*, то они должны представлять собой пары *`имя тип_данных`*, разделённые через запятую. Эти пары определяют имена, которые будут заменены значениями параметров в данном запросе. Фактические значения для замены этих имён появятся позже, при открытии курсора.

Примеры:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

Все три переменные имеют тип данных `refcursor`. Первая может быть использована с любым запросом, вторая связана (`bound`) с полностью сформированным запросом, а последняя связана с параметризованным запросом. (`key` будет заменён целочисленным значением параметра при открытии курсора.) Про переменную `curs1` говорят, что она является несвязанной (`unbound`), так как к ней не привязан никакой запрос.

Параметр `SCROLL` нельзя использовать, если в запросе курсора используется `FOR UPDATE/SHARE`. Кроме того, с запросом, включающим изменчивые функции, лучше всего использовать `NO SCROLL`. Реализация `SCROLL` предполагает, что повторное чтение вывода запроса даст согласованные результаты, чего нельзя гарантировать при использовании изменчивой функции.

41.7.2. Открытие курсора

Прежде чем получать строки из курсора, его нужно открыть. (Это эквивалентно действию SQL-команды `DECLARE CURSOR`.) В PL/pgSQL есть три формы оператора `OPEN`, две из которых используются для несвязанных курсорных переменных, а третья для связанных.

> **Примечание**
>
> Связанные курсорные переменные можно использовать с циклом `FOR` без явного открытия курсора, как описано в Подразделе 41.7.4. Цикл `FOR` откроет курсор, а затем закроет его по завершении цикла.

Открытие курсора подразумевает создание внутренней для сервера структуры данных под названием *портал*, который удерживает состояние выполнения для запроса курсора. У портала есть имя, которое должно быть уникальным в рамках сеанса на протяжении существования портала. По умолчанию PL/pgSQL присваивает уникальное имя создаваемому им порталу. Однако если присвоить переменной курсора непустое строковое значение, в качестве имени портала будет использовано именно оно. Использование этой функциональности описано в Подразделе 41.7.3.5.

41.7.2.1. `OPEN FOR` *`query`*

```
OPEN несвязанная_переменная_курсора [[NO] SCROLL] FOR query;
```

Курсорная переменная открывается и получает конкретный запрос для выполнения. Курсор не может уже быть открытым, а курсорная переменная обязана быть несвязанной (то есть просто переменной типа `refcursor`). Запрос должен быть командой `SELECT` или любой другой, которая возвращает строки (к примеру `EXPLAIN`). Запрос обрабатывается так же, как и другие команды SQL в PL/pgSQL: имена переменных PL/pgSQL заменяются на значения, план запроса кешируется для повторного использования. Подстановка значений переменных PL/pgSQL проводится при открытии курсора командой `OPEN`, последующие изменения значений переменных не влияют на работу курсора. `SCROLL` и `NO SCROLL` имеют тот же смысл, что и для связанного курсора.

Пример:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

41.7.2.2. OPEN FOR EXECUTE

```
OPEN несвязанная_переменная_курсора [[NO] SCROLL] FOR EXECUTE строка_запроса
                                     [USING выражение [, ...]];
```

Переменная курсора открывается и получает конкретный запрос для выполнения. Курсор не может быть уже открыт и он должен быть объявлен как несвязанная переменная курсора (то есть, как просто переменная `refcursor`). Запрос задаётся строковым выражением, так же, как в команде `EXECUTE`. Как обычно, это даёт возможность гибко менять план запроса от раза к разу (см. Подраздел 41.11.2). Это также означает, что замена переменных происходит не в самой строке команды. Как и с `EXECUTE`, значения параметров вставляются в динамическую команду, используя `format()` и `USING`. Параметры `SCROLL` и `NO SCROLL` здесь действуют так же, как и со связанным курсором.

Пример:

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING
 keyvalue;
```

В этом примере в текст запроса вставляется имя таблицы с применением `format()`. Значение, сравниваемое с `col1`, вставляется посредством параметра `USING`, так что заключать его в апострофы не нужно.

41.7.2.3. Открытие связанного курсора

```
OPEN связанная_переменная_курсора [( [имя_аргумента :=] значение_аргумента [, ...] )];
```

Эта форма `OPEN` используется для открытия курсорной переменной, которая была связана с запросом при объявлении. Курсор не может уже быть открытым. Список фактических значений аргументов должен присутствовать только в том случае, если курсор объявлялся с параметрами. Эти значения будут подставлены в запрос.

План запроса для связанного курсора всегда считается кешируемым. В этом случае нет эквивалента `EXECUTE`. Обратите внимание, что `SCROLL` и `NO SCROLL` не могут быть указаны в этой форме `OPEN`, возможность прокрутки назад была определена при объявлении курсора.

При передаче значений аргументов можно использовать позиционную или именную нотацию. В позиционной нотации все аргументы указываются по порядку. В именной нотации имя каждого аргумента отделяется от выражения аргумента с помощью `:=`. Это подобно вызову функций, описанному в Разделе 4.3. Также разрешается смешивать позиционную и именную нотации.

Примеры (здесь используются ранее объявленные курсоры):

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

Так как для связанного курсора выполняется подстановка значений переменных, то, на самом деле, существует два способа передать значения в курсор. Либо использовать явные аргументы в `OPEN`, либо неявно, ссылаясь на переменные PL/pgSQL в запросе. В связанном курсоре можно ссылаться только на те переменные, которые были объявлены до самого курсора. В любом случае значение переменной для подстановки в запрос будет определяться на момент выполнения `OPEN`. Вот ещё один способ получить тот же результат с `curs3`, как в примере выше:

```
DECLARE
    key integer;
```

```
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

41.7.3. Использование курсоров

После того как курсор будет открыт, с ним можно работать при помощи описанных здесь операторов.

Работать с курсором необязательно в той же функции, где он был открыт. Из функции можно вернуть значение с типом `refcursor`, что позволит вызывающему продолжить работу с курсором. (Внутри `refcursor` представляет собой обычное строковое имя портала, содержащего активный запрос курсора. Это имя можно передавать, присваивать другим переменным с типом `refcursor` и так далее, при этом портал не нарушается.)

Все порталы неявно закрываются в конце транзакции, поэтому значение `refcursor` можно использовать для ссылки на открытый курсор только до конца транзакции.

41.7.3.1. FETCH

```
FETCH [направление { FROM | IN }] курсор INTO цель;
```

`FETCH` извлекает следующую строку (в указанном направлении) из курсора в *цель*. В качестве *цели* может быть переменная-кортеж, переменная типа `record` или разделённый запятыми список простых переменных, как и в `SELECT INTO`. Если подходящей строки нет, *цели* присваивается NULL. Как и в `SELECT INTO`, проверить, была ли получена запись, можно при помощи специальной переменной `FOUND`. Если строка не найдена, курсор устанавливается после последней строки или перед первой строкой в зависимости от направления движения.

Здесь *направление* может быть любым допустимым в SQL-команде FETCH вариантом, кроме тех, что извлекают более одной строки. А именно: `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` *число*, `RELATIVE` *число*, `FORWARD` или `BACKWARD`. Без указания *направления* подразумевается вариант `NEXT`. Везде, где используется *число*, оно может определяться любым целочисленным выражением (в отличие от SQL-команды `FETCH`, допускающей только целочисленные константы). Значения *направления*, которые требуют перемещения назад, приведут к ошибке, если курсор не был объявлен или открыт с указанием `SCROLL`.

курсор это переменная с типом `refcursor`, которая ссылается на открытый портал курсора.

Примеры:

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

41.7.3.2. MOVE

```
MOVE [направление { FROM | IN }] курсор;
```

`MOVE` перемещает курсор без извлечения данных. `MOVE` работает точно так же как и `FETCH`, но при этом только перемещает курсор и не извлекает строку, к которой он переместился. Для *направления* можно использовать любой вариант, разрешённый в SQL-команде FETCH, включая извлекающие более одной строки. Курсор будет перемещён на последнюю такую строку. (Однако вариант, в котором *направление* является выражением *числа* без ключевого слова, устарел в PL/pgSQL. Этот синтаксис неоднозначен в случаях, когда *направление* опущено, поэтому может завершаться ошибкой, если *число* не является константой.) Как и в `SELECT INTO`, проверить, была ли обнаружена строка для перемещения, можно при помощи специальной переменной `FOUND`. Если строка не найдена, курсор устанавливается после последней строки или перед первой строкой в зависимости от направления движения.

Примеры:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

41.7.3.3. `UPDATE/DELETE WHERE CURRENT OF`

```
UPDATE таблица SET ... WHERE CURRENT OF курсор;
DELETE FROM таблица WHERE CURRENT OF курсор;
```

Когда курсор позиционирован на строку таблицы, эту строку можно изменить или удалить при помощи курсора. Есть ограничения на то, каким может быть запрос курсора (в частности, не должно быть группировок), и крайне желательно использовать указание `FOR UPDATE`. За дополнительными сведениями обратитесь к странице справки DECLARE.

Пример:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

41.7.3.4. `CLOSE`

```
CLOSE курсор;
```

`CLOSE` закрывает связанный с курсором портал. Используется для того, чтобы освободить ресурсы раньше, чем закончится транзакция, или чтобы освободить курсорную переменную для повторного открытия.

Пример:

```
CLOSE curs1;
```

41.7.3.5. Возврат курсора из функции

Курсоры можно возвращать из функции на PL/pgSQL. Это полезно, когда нужно вернуть множество строк и столбцов, особенно если выборки очень большие. Для этого, в функции открывается курсор и его имя возвращается вызывающему (или просто открывается курсор, используя указанное имя портала, каким-либо образом известное вызывающему). Вызывающий затем может извлекать строки из курсора. Курсор может быть закрыт вызывающим или он будет автоматически закрыт при завершении транзакции.

Имя портала, используемое для курсора, может быть указано разработчиком или будет генерироваться автоматически. Чтобы указать имя портала, нужно просто присвоить строку в переменную `refcursor` перед его открытием. Значение строки переменной `refcursor` будет использоваться командой `OPEN` как имя портала. Однако если переменная `refcursor` имеет значение NULL (как и установлено по умолчанию), `OPEN` автоматически генерирует имя, которое не конфликтует с любым существующим порталом, и присваивает его переменной `refcursor`.

> **Примечание**
>
> В версиях Postgres Pro ниже 16 связанные курсорные переменные инициализировались в значения, содержащие собственные имена, а не в NULL, поэтому имя портала совпадало с именем курсорной переменной, заданным по умолчанию. Поведение изменилось во избежание конфликтов между курсорами с одинаковыми именами в разных функциях.

Следующий пример показывает один из способов передачи имени курсора вызывающему:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
```

```
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

В следующем примере используется автоматическая генерация имени курсора:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- для использования курсоров, необходимо начать транзакцию
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

В следующем примере показан один из способов вернуть несколько курсоров из одной функции:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- для использования курсоров необходимо начать транзакцию
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

41.7.4. Обработка курсора в цикле

Один из вариантов цикла `FOR` позволяет перебирать строки, возвращённые курсором. Вот его синтаксис:

```
[ <<метка>> ]
```

```
FOR переменная-запись IN связанная_переменная_курсора [ ( [ имя_аргумента
 := ] значение_аргумента [, ...] ) ] LOOP
    операторы
END LOOP [ метка ];
```

Курсорная переменная должна быть связана с запросом при объявлении. Курсор *не может* быть открытым. Команда `FOR` автоматически открывает курсор и автоматически закрывает при завершении цикла. Список фактических значений аргументов должен присутствовать только в том случае, если курсор объявлялся с параметрами. Эти значения будут подставлены в запрос, как и при выполнении `OPEN` (см. Подраздел 41.7.2.3).

Данная *`переменная-запись`* автоматически определяется как переменная типа `record` и существует только внутри цикла (другие объявленные переменные с таким именем игнорируется в цикле). Каждая возвращаемая курсором строка последовательно присваивается этой переменной и выполняется тело цикла.

41.8. Управление транзакциями

В процедурах, вызываемых командой `CALL`, а также в анонимных блоках кода (в команде `DO`) можно завершать транзакции, выполняя `COMMIT` и `ROLLBACK`. После завершения транзакции этими командами новая будет начата автоматически, поэтому отдельной команды `START TRANSACTION` нет. (Заметьте, что команды `BEGIN` и `END` в PL/pgSQL имеют другой смысл.)

Пример:

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END;
$$;

CALL transaction_test1();
```

Новая транзакция начинается с теми характеристиками, в частности, уровнем изоляции, которые установлены для транзакций по умолчанию. В случаях, когда транзакции фиксируются в цикле, может быть удобнее автоматически начинать следующую транзакцию с теми же характеристиками, что имеет предыдущая. Это позволяют реализовать команды `COMMIT AND CHAIN` и `ROLLBACK AND CHAIN`.

Управление транзакциями возможно только в вызовах `CALL` или `DO` в коде верхнего уровня или во вложенных `CALL` или `DO` без других промежуточных команд. Например, в стеке вызовов `CALL proc1()` → `CALL proc2()` → `CALL proc3()` вторая и третья процедуры могут управлять транзакциями. Но в стеке `CALL proc1()` → `SELECT func2()` → `CALL proc3()` последняя процедура лишена этой возможности из-за промежуточного `SELECT`.

PL/pgSQL не поддерживает точки сохранения (команды `SAVEPOINT`/`ROLLBACK TO SAVEPOINT`/`RELEASE SAVEPOINT`). Типичные схемы использования точек сохранения можно реализовать при помощи блоков с обработчиками исключений (см. Подраздел 41.6.8). Блок с обработчиками исключений фактически формирует подтранзакцию, это означает, что транзакции не могут быть завершены внутри такого блока.

Циклам с курсорами присущи некоторые особенности. Рассмотрите этот пример:

```
CREATE PROCEDURE transaction_test2()
LANGUAGE plpgsql
AS $$
DECLARE
    r RECORD;
BEGIN
    FOR r IN SELECT * FROM test2 ORDER BY x LOOP
        INSERT INTO test1 (a) VALUES (r.x);
        COMMIT;
    END LOOP;
END;
$$;

CALL transaction_test2();
```

Обычно курсоры автоматически закрываются при фиксировании транзакции. Однако курсор, создаваемый внутри цикла подобным образом, автоматически преобразуется в удерживаемый курсор первой командой `COMMIT` или `ROLLBACK`. Это означает, что курсор полностью вычисляется при выполнении первой команды `COMMIT` или `ROLLBACK`, а не для каждой очередной строки. При этом он автоматически удаляется после цикла, так что это происходит практически незаметно для пользователя. Однако стоит учитывать, что любые блокировки таблиц или строк, принимаемые запросом курсора, не будут удерживаться после первого выполнения команд `COMMIT` или `ROLLBACK`.

Команды управления транзакциями не допускаются в циклах с курсором, которыми управляют запросы, производящие не только чтение, но и модификацию данных (например, `UPDATE ... RETURNING`).

41.9. Сообщения и ошибки

41.9.1. Вывод сообщений и ошибок

Команда `RAISE` предназначена для вывода сообщений и вызова ошибок.

```
RAISE [ уровень ] 'формат' [, выражение [, ... ]] [ USING параметр = значение
 [, ... ] ];
RAISE [ уровень ] имя_условия [ USING параметр = выражение [, ... ] ];
RAISE [ уровень ] SQLSTATE 'sqlstate' [ USING параметр = выражение [, ... ] ];
RAISE [ уровень ] USING параметр = выражение [, ... ];
RAISE ;
```

уровень задаёт уровень важности ошибки. Возможные значения: `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` и `EXCEPTION`. По умолчанию используется `EXCEPTION`. `EXCEPTION` вызывает ошибку (что обычно прерывает текущую транзакцию), остальные значения *уровня* только генерируют сообщения с различными уровнями приоритета. Будут ли сообщения конкретного приоритета переданы клиенту или записаны в журнал сервера, или и то, и другое, зависит от конфигурационных переменных log_min_messages и client_min_messages. За дополнительными сведениями обратитесь к Главе 18.

После указания *уровня*, если оно есть, можно задать строку *формата* (это должна быть простая строковая константа, не выражение). Строка формата определяет вид текста об ошибке, который будет выдан. За строкой формата могут следовать необязательные выражения аргументов, которые будут вставлены в сообщение. Внутри строки формата знак `%` заменяется строковым представлением значения очередного аргумента. Чтобы выдать символ `%` буквально, продублируйте его (как `%%`). Число аргументов должно совпадать с числом местозаполнителей `%` в строке формата, иначе при компиляции функции возникнет ошибка.

В следующем примере символ `%` будет заменён на значение `v_job_id`:

```
RAISE NOTICE 'Вызов функции cs_create_job(%)', v_job_id;
```

При помощи `USING` и последующих элементов *`параметр`* = *`выражение`* можно добавить дополнительную информацию к отчёту об ошибке. Все *`выражения`* представляют собой строковые выражения. Возможные ключевые слова для *`параметра`* следующие:

`MESSAGE`

Устанавливает текст сообщения об ошибке. Этот параметр не может использоваться, если в команде `RAISE` присутствует *`формат`* перед `USING`.

`DETAIL`

Предоставляет детальное сообщение об ошибке.

`HINT`

Предоставляет подсказку по вызванной ошибке.

`ERRCODE`

Устанавливает код ошибки (`SQLSTATE`). Код ошибки задаётся либо по имени, как показано в Приложении A, или напрямую, пятисимвольный код `SQLSTATE`.

`COLUMN`
`CONSTRAINT`
`DATATYPE`
`TABLE`
`SCHEMA`

Предоставляет имя соответствующего объекта, связанного с ошибкой.

Этот пример прерывает транзакцию и устанавливает сообщение об ошибке с подсказкой:

```
RAISE EXCEPTION 'Несуществующий ID --> %', user_id
      USING HINT = 'Проверьте ваш пользовательский ID';
```

Следующие два примера демонстрируют эквивалентные способы задания `SQLSTATE`:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

У команды `RAISE` есть и другой синтаксис, в котором в качестве главного аргумента используется имя или код `SQLSTATE` ошибки. Например:

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Предложение `USING` в этом синтаксисе можно использовать для того, чтобы переопределить стандартное сообщение об ошибке, детальное сообщение, подсказку. Ещё один вариант предыдущего примера:

```
RAISE unique_violation USING MESSAGE = 'ID пользователя уже существует: ' || user_id;
```

Ещё один вариант — использовать `RAISE USING` или `RAISE` *`уровень`* `USING`, а всё остальное записать в списке `USING`.

И заключительный вариант, в котором `RAISE` не имеет параметров вообще. Эта форма может использоваться только в секции `EXCEPTION` блока и предназначена для того, чтобы повторно вызвать ошибку, которая сейчас перехвачена и обрабатывается.

> ## Примечание
>
> До версии PostgreSQL 9.1 команда `RAISE` без параметров всегда вызывала ошибку с выходом из блока, содержащего активную секцию `EXCEPTION`. Эту ошибку нельзя было перехва-

тить, даже если `RAISE` в секции `EXCEPTION` поместить во вложенный блок со своей секцией `EXCEPTION`. Это было сочтено удивительным и не совместимым с Oracle PL/SQL.

Если в команде `RAISE EXCEPTION` не задано ни имя, ни код `SQLSTATE`, по умолчанию выдаётся `raise_exception` (`P0001`). Если не задан текст сообщения, по умолчанию в качестве этого текста передаётся имя условия или код SQLSTATE.

Примечание

При задании `SQLSTATE` кода необязательно использовать только список предопределённых кодов ошибок. В качестве кода ошибки может быть любое пятисимвольное значение, состоящее из цифр и/или ASCII символов в верхнем регистре, кроме `00000`. Не рекомендуется использовать коды ошибок, которые заканчиваются на `000`, потому что так обозначаются коды категорий. И чтобы их перехватить, нужно перехватывать целую категорию.

41.9.2. Проверка утверждений

Оператор `ASSERT` представляет удобное средство вставлять отладочные проверки в функции PL/pgSQL.

```
ASSERT условие [ , сообщение ];
```

Здесь *`условие`* — это логическое выражение, которое, как ожидается, должно быть всегда истинным; если это так, оператор `ASSERT` больше ничего не делает. Если же оно возвращает ложь или NULL, этот оператор выдаёт исключение `ASSERT_FAILURE`. (Если ошибка происходит при вычислении *`условия`*, она выдаётся как обычная ошибка.)

Если в нём задаётся необязательное *`сообщение`*, результат этого выражения (если он не NULL) заменяет сообщение об ошибке по умолчанию «assertion failed» (нарушение истинности), в случае, если *`условие`* не выполняется. В обычном случае, когда условие утверждения выполняется, выражение *`сообщения`* не вычисляется.

Проверку утверждений можно включить или отключить с помощью конфигурационного параметра `plpgsql.check_asserts`, принимающего логическое значение; по умолчанию она включена (`on`). Если этот параметр отключён (`off`), операторы `ASSERT` ничего не делают.

Учтите, что оператор `ASSERT` предназначен для выявления программных дефектов, а не для вывода обычных ошибок (для этого используется оператор `RAISE`, описанный выше).

41.10. Триггерные функции

В PL/pgSQL можно создавать триггерные функции, которые будут вызываться при изменениях данных или событиях в базе данных. Триггерная функция создаётся командой `CREATE FUNCTION`, при этом у функции не должно быть аргументов, а типом возвращаемого значения должен быть `trigger` (для триггеров, срабатывающих при изменениях данных) или `event_trigger` (для триггеров, срабатывающих при событиях в базе). Для триггеров автоматически определяются специальные локальные переменные с именами вида `TG_`*`имя`*, описывающие условие, повлёкшее вызов триггера.

41.10.1. Триггеры при изменении данных

Триггер при изменении данных объявляется как функция без аргументов и с типом результата `trigger`. Заметьте, что эта функция должна объявляться без аргументов, даже если ожидается, что она будет получать аргументы, заданные в команде `CREATE TRIGGER` — такие аргументы передаются через `TG_ARGV`, как описано ниже.

Когда функция на PL/pgSQL срабатывает как триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

`NEW record`

Новая строка базы данных для команд `INSERT`/`UPDATE` в триггерах уровня строки. В триггерах уровня оператора и для команды `DELETE` эта переменная имеет значение NULL.

`OLD record`

Старая строка базы данных для команд `UPDATE`/`DELETE` в триггерах уровня строки. В триггерах уровня оператора и для команды `INSERT` эта переменная имеет значение NULL.

`TG_NAME name`

Имя сработавшего триггера.

`TG_WHEN text`

`BEFORE`, `AFTER` или `INSTEAD OF` в зависимости от определения триггера.

`TG_LEVEL text`

`ROW` или `STATEMENT` в зависимости от определения триггера.

`TG_OP text`

Операция, для которой сработал триггер: `INSERT`, `UPDATE`, `DELETE` или `TRUNCATE`.

`TG_RELID oid` (ссылается на `pg_class`.`oid`)

OID таблицы, для которой сработал триггер.

`TG_RELNAME name`

Таблица, для которой сработал триггер. Эта переменная устарела и может стать недоступной в будущих релизах. Вместо неё нужно использовать `TG_TABLE_NAME`.

`TG_TABLE_NAME name`

Таблица, для которой сработал триггер.

`TG_TABLE_SCHEMA name`

Схема таблицы, для которой сработал триггер.

`TG_NARGS integer`

Число аргументов в команде `CREATE TRIGGER`, которые передаются в триггерную функцию.

`TG_ARGV text[]`

Аргументы от оператора `CREATE TRIGGER`. Индекс массива начинается с 0. Для недопустимых значений индекса (< 0 или >= `tg_nargs`) возвращается NULL.

Триггерная функция должна вернуть либо `NULL`, либо запись/строку, соответствующую структуре таблице, для которой сработал триггер.

Если `BEFORE` триггер уровня строки возвращает `NULL`, то все дальнейшие действия с этой строкой прекращаются (т. е. не срабатывают последующие триггеры, команда `INSERT`/`UPDATE`/`DELETE` для этой строки не выполняется). Если возвращается не `NULL`, то дальнейшая обработка продолжается именно с этой строкой. Возвращение строки отличной от начальной `NEW`, изменяет строку, которая будет вставлена или изменена. Поэтому, если в триггерной функции нужно выполнить некоторые действия и не менять саму строку, то нужно возвратить переменную `NEW` (или её эквивалент). Для того чтобы изменить сохраняемую строку, можно поменять отдельные значения в переменной `NEW` и затем её вернуть. Либо создать и вернуть полностью новую переменную. В случае строчного триггера `BEFORE` для команды `DELETE` само возвращаемое значение не имеет прямого эффекта, но оно должно быть отличным от `NULL`, чтобы не прерывать обработку строки. Обратите внимание, что переменная `NEW` всегда `NULL` в триггерах на `DELETE`, поэтому возвращать её не имеет смысла. Традиционной идиомой для триггеров `DELETE` является возврат переменной `OLD`.

Триггеры `INSTEAD OF` (это всегда триггеры уровня строк и они могут применяться только с представлениями) могут возвращать NULL, чтобы показать, что они не выполняли никаких изменений, так что обработку этой строки можно не продолжать (то есть, не вызывать последующие триггеры и не считать строку в числе обработанных строк для окружающих команд `INSERT`/`UPDATE`/`DELETE`). В противном случае должно быть возвращено значение, отличное от NULL, показывающее, что триггер выполнил запрошенную операцию. Для операций `INSERT` и `UPDATE` возвращаемым значением должно быть `NEW`, которое триггерная функция может модифицировать для поддержки предложений `INSERT RETURNING` и `UPDATE RETURNING` (это также повлияет на значение строки, передаваемое последующим триггерам, или доступное под специальным псевдонимом `EXCLUDED` в операторе `INSERT` с предложением `ON CONFLICT DO UPDATE`). Для операций `DELETE` возвращаемым значением должно быть `OLD`.

Возвращаемое значение для строчного триггера AFTER и триггеров уровня оператора (BEFORE или AFTER) всегда игнорируется. Это может быть и NULL. Однако в этих триггерах по-прежнему можно прервать вызвавшую их команду, для этого нужно явно вызвать ошибку.

Пример 41.3 показывает пример триггерной функции в PL/pgSQL.

Пример 41.3. Триггерная функция на PL/pgSQL

Триггер, показанный в этом примере, при любом добавлении или изменении строки в таблице сохраняет в этой строке информацию о текущем пользователе и отметку времени. Кроме того, он требует, чтобы было указано имя сотрудника и зарплата задавалась положительным числом.

```
CREATE TABLE emp (
    empname           text,
    salary            integer,
    last_date         timestamp,
    last_user         text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Проверить, что указаны имя сотрудника и зарплата
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Кто будет работать, если за это надо будет платить?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Запомнить, кто и когда изменил запись
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

Другой вариант ведения журнала изменений для таблицы предполагает создание новой таблицы, которая будет содержать отдельную запись для каждой выполненной команды INSERT, UPDATE, DELETE. Этот подход можно рассматривать как протоколирование изменений таблицы для аудита. Пример 41.4 показывает реализацию соответствующей триггерной функции в PL/pgSQL.

Пример 41.4. Триггерная функция для аудита в PL/pgSQL

Показанный в этом примере триггер гарантирует, что любое добавление, изменение или удаление строки в таблице `emp` будет зафиксировано в таблице `emp_audit` (для аудита). Также он фиксирует текущее время, имя пользователя и тип выполняемой операции.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Добавление строки в emp_audit, которая отражает операцию, выполняемую в emp,
        -- для определения типа операции применяется специальная переменная TG_OP.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), current_user, OLD.*;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), current_user, NEW.*;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), current_user, NEW.*;
        END IF;
        RETURN NULL; -- возвращаемое значение для триггера AFTER игнорируется
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE FUNCTION process_emp_audit();
```

У предыдущего примера есть разновидность, которая использует представление, соединяющее основную таблицу и таблицу аудита, для отображения даты последнего изменения каждой строки. При этом подходе по-прежнему ведётся полный журнал аудита в отдельной таблице, но также имеется представление с упрощенным аудиторским следом. Это представление содержит временную метку, которая вычисляется для каждой строки из данных аудиторской таблицы. Пример 41.5 показывает пример триггера на представление для аудита в PL/pgSQL.

Пример 41.5. Триггерная функция на PL/pgSQL для аудита в представлении

В этом примере триггер, связанный с представлением, делает это представление изменяемым и гарантирует, что любая команда на добавление, изменение или удаление строки в представлении будет записана для аудита в таблицу `emp_audit`. Также записываются временная метка, имя пользователя и тип выполняемой операции. Представление показывает дату последнего изменения для каждой строки.

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
```

```
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --
        -- Выполнить требуемую операцию в emp и добавить в emp_audit строку,
        -- отражающую эту операцию.
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', current_user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', current_user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', current_user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE FUNCTION update_emp_view();
```

Один из вариантов использования триггеров это поддержание в актуальном состоянии отдельной таблицы итогов для некоторой таблицы. В некоторых случаях отдельная таблица с итогами может использоваться в запросах вместо основной таблицы. При этом зачастую время выполнения запросов значительно сокращается. Эта техника широко используется в хранилищах данных, где таблицы фактов могут быть очень большими. Пример 41.6 демонстрирует триггерную функцию на PL/pgSQL, которая поддерживает таблицу итогов для таблицы фактов в хранилище данных.

Пример 41.6. Триггерная функция на PL/pgSQL для ведения таблицы итогов

Представленная здесь схема данных частично основана на примере *Grocery Store* из книги *The Data Warehouse Toolkit* Ральфа Кимбалла (Ralph Kimball).

```
--
-- Основные таблицы: таблица временных периодов и таблица фактов продаж
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Таблица с итогами продаж по периодам
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Функция и триггер для пересчёта столбцов итогов при выполнении
-- команд INSERT, UPDATE, DELETE
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- Вычислить изменение количества/суммы.
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;
```

```
        ELSIF (TG_OP = 'UPDATE') THEN

            -- Запретить изменение time_key -
            -- (это ограничение не должно вызвать неудобств, так как
            -- в основном изменения будут выполняться по схеме DELETE + INSERT).
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                      OLD.time_key, NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;


        -- Внести новые значения в существующую строку итогов или
        -- добавить новую.
        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;

            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- ничего не делать
            END;
        END LOOP insert_update;
```

```
        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE FUNCTION maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

Триггеры `AFTER` также могут использовать *переходные таблицы* для просмотра всего набора строк, изменённых оператором, вызвавшим триггер. Команда `CREATE TRIGGER` назначает имена одной или обеим переходным таблицам, а затем функция может по этим именам обращаться к ним как к временным таблицам только для чтения. Это иллюстрирует Пример 41.7.

Пример 41.7. Организация аудита с переходными таблицами

В данном примере достигается тот же результат, что и в Пример 41.4, но вместо триггера, срабатывающего для каждой строки, в нём используется триггер, срабатывающий единожды для оператора и получающий нужные ему данные в переходной таблице. Это может быть гораздо быстрее, чем вариант с построчным триггером, когда целевой оператор изменяет сразу множество строк. Заметьте, что мы должны объявить отдельные триггеры для каждого вида события, так как предложения `REFERENCING` в каждом случае будут разными. Но это не мешает при желании использовать одну триггерную функцию. (На практике может быть лучше использовать три отдельные функции и не проверять `TG_OP` во время выполнения.)

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Добавление строк в emp_audit, которые отражают операции, выполняемые в emp,
        -- для определения типа операций применяется специальная переменная TG_OP.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit
                SELECT 'D', now(), current_user, o.* FROM old_table o;
```

```
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit
                SELECT 'U', now(), current_user, n.* FROM new_table n;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit
                SELECT 'I', now(), current_user, n.* FROM new_table n;
        END IF;
        RETURN NULL; -- возвращаемое значение для триггера AFTER игнорируется
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit_ins
    AFTER INSERT ON emp
    REFERENCING NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_upd
    AFTER UPDATE ON emp
    REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_del
    AFTER DELETE ON emp
    REFERENCING OLD TABLE AS old_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

41.10.2. Триггеры событий

В PL/pgSQL можно создавать событийные триггеры. Postgres Pro требует, чтобы функция, которая вызывается как событийный триггер, объявлялась без аргументов и типом возвращаемого значения был `event_trigger`.

Когда функция на PL/pgSQL вызывается как событийный триггер, в блоке верхнего уровня автоматически создаются несколько специальных переменных:

`TG_EVENT text`

событие, при котором сработал триггер.

`TG_TAG text`

тег команды, для которой сработал триггер.

Пример 41.8 демонстрирует реализацию функции событийного триггера на PL/pgSQL.

Пример 41.8. Функция событийного триггера на PL/pgSQL

Триггер в этом примере просто выдаёт сообщение `NOTICE` каждый раз, когда выполняется поддерживаемая команда.

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE FUNCTION snitch();
```

41.11. PL/pgSQL изнутри

В этом разделе обсуждаются некоторые детали реализации, которые пользователям PL/pgSQL важно знать.

41.11.1. Подстановка переменных

SQL-операторы и выражения внутри функции на PL/pgSQL могут ссылаться на переменные и параметры этой функции. За кулисами PL/pgSQL заменяет параметры запросов для таких ссылок. Параметры будут заменены только в местах, где они синтаксически допустимы. Как крайний случай, рассмотрим следующий пример плохого стиля программирования:

```
INSERT INTO foo (foo) VALUES (foo(foo));
```

Первый раз `foo` появляется на том месте, где синтаксически должно быть имя таблицы, поэтому замены не будет, даже если функция имеет переменную `foo`. Второй раз `foo` встречается там, где должно быть имя столбца таблицы, поэтому замены не будет и здесь. Третий раз — на месте, где должно быть имя функции, поэтому замены не будет. Только последнее вхождение `foo` является кандидатом на то, чтобы быть ссылкой на переменную функции PL/pgSQL.

Объяснить принцип действия можно ещё и так: при подстановке переменных в SQL-команду могут вставляться только значения данных; при этом нельзя динамически менять объекты базы данных, на которые ссылается команда. (Для этого нужно сконструировать строку команды динамически, как описано в Подразделе 41.5.4.)

Если имена переменных синтаксически не отличаются от названий столбцов таблицы, то возможна двусмысленность и в ссылках на таблицы. Является ли данное имя ссылкой на столбец таблицы или ссылкой на переменную? Изменим предыдущий пример:

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

Здесь `dest` и `src` должны быть именами таблиц, `col` должен быть столбцом `dest`. Однако `foo` и `bar` могут быть как переменными функции, так и столбцами `src`.

По умолчанию, PL/pgSQL выдаст ошибку, если имя в операторе SQL может относиться как к переменной, так и к столбцу таблицы. Ситуацию можно исправить переименованием переменной, переименованием столбца, точной квалификацией неоднозначной ссылки или указанием PL/pgSQL машине, какую интерпретацию предпочесть.

Самое простое решение — переименовать переменную или столбец. Общее правило кодирования предполагает использование различных соглашений о наименовании для переменных PL/pgSQL и столбцов таблиц. Например, если имена переменных всегда имеют вид `v_`*`имя`*, а имена столбцов никогда не начинаются на `v_`, то конфликты исключены.

В качестве альтернативы можно дополнить имена неоднозначных ссылок, чтобы сделать их точными. В приведённом выше примере `src.foo` однозначно бы определялась, как ссылка на столбец таблицы. Чтобы сделать однозначной ссылку на переменную, переменная должна быть объявлена в блоке с меткой, и далее нужно использовать эту метку (см. Раздел 41.2). Например:

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

Здесь `block.foo` ссылается на переменную, даже если в таблице `src` есть столбец `foo`. Параметры функции, а также специальные переменные, такие как `FOUND`, могут быть дополнены именем функции, потому что они неявно объявлены во внешнем блоке, метка которого совпадает с именем функции.

Иногда может быть не очень практичным исправлять таким способом все неоднозначные ссылки в большом куске PL/pgSQL кода. В таких случаях можно указать, чтобы PL/pgSQL разрешал неоднозначные ссылки в пользу переменных (это совместимо с PL/pgSQL до версии PostgreSQL 9.0), или в пользу столбцов таблицы (совместимо с некоторыми другими системами, такими как Oracle).

На уровне всей системы поведение PL/pgSQL регулируется установкой конфигурационного параметра `plpgsql.variable_conflict`, имеющего значения: `error`, `use_variable` или `use_column` (`error` устанавливается по умолчанию при установке системы). Изменение этого параметра влияет на все последующие компиляции операторов в функциях на PL/pgSQL, но не на операторы уже скомпилированные в текущем сеансе. Так как изменение этого параметра может привести к неожиданным изменениям в поведении функций на PL/pgSQL, он может быть изменён только суперпользователем.

Поведение PL/pgSQL можно изменять для каждой отдельной функции, если добавить в начало функции одну из этих специальных команд:

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

Эти команды влияют только на функцию, в которой они записаны и перекрывают действие `plpgsql.variable_conflict`. Пример:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

В команде `UPDATE`, `curtime`, `comment` и `id` будут ссылаться на переменные и параметры функции вне зависимости от того, есть ли столбцы с такими именами в таблице `users`. Обратите внимание, что нужно дополнить именем таблицы ссылку на `users.id` в предложении `WHERE`, чтобы она ссылалась на столбец таблицы. При этом необязательно дополнять ссылку на `comment` в левой части списка `UPDATE`, так как синтаксически в этом месте должно быть имя столбца таблицы `users`. Эту функцию можно было бы записать и без зависимости от значения `variable_conflict`:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

Замена переменных не происходит в строке, исполняемой командой `EXECUTE` или её вариантом. Если нужно вставлять изменяющиеся значения в такую команду, то это делается либо при построении самой командной строки или с использованием `USING`, как показано в Подразделе 41.5.4.

Замена переменных в настоящее время работает только в командах `SELECT`, `INSERT`, `UPDATE`, `DELETE` и командах, содержащих одну из вышеперечисленных (например, `EXPLAIN` и `CREATE TABLE ... AS SELECT`), потому что основной исполнитель SQL допускает использование параметров запроса только в этих командах. Чтобы использовать изменяемые имена или значения в других типах операторов (обычно называемые служебными), необходимо составить текст команды в виде строки и выполнить её в `EXECUTE`.

41.11.2. Кеширование плана

Интерпретатор PL/pgSQL анализирует исходный текст функции и строит внутреннее бинарное дерево инструкций при первом вызове функции (для каждого сеанса). В дерево инструкций полностью переводится вся структура операторов PL/pgSQL, но для выражений и команд SQL, используемых в функции, это происходит не сразу.

При первом выполнении в функции каждого выражения или команды SQL интерпретатор PL/pgSQL разбирает и анализирует команду для создания подготовленного к выполнению оператора с помощью функции `SPI_prepare` менеджера интерфейса программирования сервера. Последующие обращения к этому выражению или команде повторно используют подготовленный к выполнению оператор. Таким образом, SQL-команды, находящиеся в редко посещаемой ветке кода условного оператора, не несут накладных расходов на разбор команд, если они так и не будут выполнены в текущем сеансе. Здесь есть недостаток, заключающийся в том, что ошибки в определённом выражении или команде не могут быть обнаружены, пока выполнение не дойдёт до этой части функции. (Тривиальные синтаксические ошибки обнаружатся в ходе первоначального разбора, но ничего более серьёзного не будет обнаружено до исполнения.)

Кроме того, PL/pgSQL (точнее, менеджер интерфейса программирования сервера) будет пытаться кешировать план выполнения для любого подготовленного к исполнению оператора. При каждом вызове оператора, если не используется план из кеша, генерируется новый план выполнения, и текущие значения параметров (то есть значения переменных PL/pgSQL) могут быть использованы для оптимизации нового плана. Если оператор не имеет параметров или выполняется много раз, менеджер интерфейса программирования сервера рассмотрит вопрос о создании и кешировании (для повторного использования) общего плана, не зависящего от значений параметров. Как правило, это происходит в тех случаях, когда план выполнения не очень чувствителен к имеющимся ссылкам на значения переменных PL/pgSQL. В противном случае выгоднее каждый раз формировать новый план. Более подробно поведение подготовленных операторов рассматривается в PREPARE.

Чтобы PL/pgSQL мог сохранять подготовленные операторы и планы выполнения, команды SQL в коде PL/pgSQL, должны использовать одни и те же таблицы и столбцы при каждом исполнении. А это значит, что в SQL-командах нельзя использовать названия таблиц и столбцов в качестве параметров. Чтобы обойти это ограничение, нужно сконструировать динамическую команду для оператора PL/pgSQL `EXECUTE` — ценой будет разбор и построение нового плана выполнения при каждом вызове.

Изменчивая природа переменных типа `record` представляет ещё одну проблему в этой связи. Когда поля переменной типа `record` используются в выражениях или операторах, типы данных полей не должны меняться от одного вызова функции к другому, так как при анализе каждого выражения будет использоваться тот тип данных, который присутствовал при первом вызове. При необходимости можно использовать `EXECUTE` для решения этой проблемы.

Если функция используется в качестве триггера более чем для одной таблицы, PL/pgSQL независимо подготавливает и кеширует операторы для каждой такой таблицы. То есть создаётся кеш для каждой комбинации триггерная функция + таблица, а не только для каждой функции. Это устраняет некоторые проблемы, связанные с различными типами данных. Например, триггерная функция сможет успешно работать со столбцом `key`, даже если в разных таблицах этот столбец имеет разные типы данных.

Таким же образом, функции с полиморфными типами аргументов имеют отдельный кеш для каждой комбинации фактических типов аргументов, так что различия типов данных не вызывают неожиданных сбоев.

Кеширование операторов иногда приводит к неожиданным эффектам при интерпретации чувствительных ко времени значений. Например, есть разница между тем, что делают эти две функции:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

и

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

В случае `logfunc1`, при анализе `INSERT`, основной анализатор Postgres Pro знает, что строку `'now'` следует толковать как `timestamp`, потому что целевой столбец таблицы `logtable` имеет такой тип данных. Таким образом, `'now'` будет преобразовано в константу `timestamp` при анализе `INSERT`, а затем эта константа будет использоваться в последующих вызовах `logfunc1` в течение всего сеанса. Разумеется, это не то, что хотел программист. Лучше было бы использовать функцию `now()` или `current_timestamp`.

В случае `logfunc2`, основной анализатор Postgres Pro не знает, какого типа будет `'now'` и поэтому возвращает значение типа `text`, содержащее строку `now`. При последующем присваивании локальной переменной `curtime` интерпретатор PL/pgSQL приводит эту строку к типу `timestamp`, вызывая функции `textout` и `timestamp_in`. Таким образом, метка времени будет обновляться при каждом выполнении, как и ожидается программистом. И хотя всё работает как ожидалось, это не слишком эффективно, поэтому использование функции `now()` по-прежнему значительно лучше.

41.12. Советы по разработке на PL/pgSQL

Хороший способ разрабатывать на PL/pgSQL заключается в том, чтобы в одном окне с текстовым редактором по выбору создавать тексты функций, а в другом окне с psql загружать и тестировать эти функции. В таком случае удобно записывать функцию, используя `CREATE OR REPLACE FUNCTION`. Таким образом, можно легко загрузить файл для обновления определения функции. Например:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

В psql, можно загрузить или перезагрузить такой файл определения функции, выполнив:

```
\i filename.sql
```

а затем сразу выполнять команды SQL для тестирования функции.

Ещё один хороший способ разрабатывать на PL/pgSQL связан с использованием GUI инструментов, облегчающих разработку на процедурном языке. Один из примеров такого инструмента pgAdmin, хотя есть и другие. Такие инструменты часто предоставляют удобные возможности, такие как экранирование одинарных кавычек, отладка и повторное создание функций.

41.12.1. Обработка кавычек

Код функции на PL/pgSQL указывается в команде `CREATE FUNCTION` в виде строки. Если записывать строку как обычно, внутри одинарных кавычек, то любой символ одинарной кавычки должен дублироваться, так же как и должен дублироваться каждый знак обратной косой черты (если используется синтаксис с экранированием в строках). Дублирование кавычек в лучшем случае утомительно, а в более сложных случаях код может стать совершенно непонятным, так как легко может потребоваться четыре или более идущих подряд кавычек. Вместо этого при создании тела функции рекомендуется использовать знаки доллара в качестве кавычек (см. Подраздел 4.1.2.4). При таком подходе никогда не потребуется дублировать кавычки, но придётся позаботиться о том, чтобы иметь разные долларовые разделители для каждого уровня вложенности. Например, команду `CREATE FUNCTION` можно записать так:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
```

```
$PROC$ LANGUAGE plpgsql;
```

Внутри можно использовать кавычки для простых текстовых строк и `$$` для разграничения фрагментов SQL-команды, собираемой из отдельных строк. Если нужно взять в кавычки текст, который включает `$$`, можно использовать `$Q$`, и так далее.

Следующая таблица показывает, как применяются знаки кавычек, если не используется экранирование долларами. Это может быть полезно при переводе кода, не использующего экранирование знаками доллара, в нечто более понятное.

1 кавычка

В начале и конце тела функции, например:

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

Внутри такой функции любая кавычка *должна* дублироваться.

2 кавычки

Для строковых литералов внутри тела функции, например:

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_name=''foobar'';
```

При использовании знаков доллара можно просто написать:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

и именно это увидит исполнитель PL/pgSQL в обоих случаях.

4 кавычки

Когда нужны одинарные кавычки в строковой константе внутри тела функции, например:

```
a_output := a_output || '' AND name LIKE ''''foobar'''' AND xyz''
```

К `a_output` будет добавлено: `AND name LIKE 'foobar' AND xyz`

При использовании знаков доллара это записывается так:

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

будьте внимательны, при этом не должно быть внешнего долларового разделителя `$$`.

6 кавычек

Когда нужны одинарные кавычки в строковой константе внутри тела функции, при этом кавычки находятся в конце строковой константы. Например:

```
a_output := a_output || '' AND name LIKE ''''foobar''''''
```

К `a_output` будет добавлено: `AND name LIKE 'foobar'`.

При использовании знаков доллара это записывается так:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 кавычек

Когда нужны две одиночные кавычки в строковой константе (это уже 8 кавычек), примыкающие к концу строковой константы (ещё 2). Вероятно, такое может понадобиться при разработке функции, которая генерирует другие функции, как показано в Примере 41.10. Например:

```
a_output := a_output || '' if v_'' ||
```

```
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

Значение `a_output` затем будет:

```
if v_... like ''...'' then return ''...''; end if;
```

При использовании знаков доллара:

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

где предполагается, что нужны только одиночные кавычки в `a_output`, так как потребуется повторное взятие в кавычки перед использованием.

41.12.2. Дополнительные проверки во время компиляции и во время выполнения

Чтобы помочь найти и предупредить простые, но часто встречающиеся проблемы, PL/PgSQL предоставляет дополнительные *проверки*. Если они включены в конфигурации, то во время компиляции функций будут выдаваться дополнительные сообщения `WARNING` или ошибки `ERROR`. Функция, при компиляции которой выдавалось `WARNING`, при последующем выполнении не будет выдавать это сообщение и её можно протестировать в отдельной среде разработки.

В среде разработки и/или тестирования имеет смысл установить значение "`all`" для параметра `plpgsql.extra_warnings` или `plpgsql.extra_errors`.

Для включения этих проверок предназначены параметры `plpgsql.extra_warnings` (для предупреждений) и `plpgsql.extra_errors` (для ошибок). Каждому из параметров можно присвоить список значений, разделённых запятыми, значение "`none`" или "`all`". По умолчанию используется "`none`". В настоящий момент доступны следующие дополнительные проверки:

`shadowed_variables`

Проверяет, что объявление новой переменной не скрывает ранее объявленную переменную.

`strict_multi_assignment`

Некоторые команды PL/pgSQL допускают присваивание значений сразу нескольким переменным, например `SELECT INTO`. Обычно количество целевых переменных должно совпадать с количеством исходных, хотя PL/pgSQL будет использовать `NULL` вместо пропущенных значений, а дополнительные переменные игнорировать. При включении этой проверки PL/pgSQL будет выдавать предупреждение (`WARNING`) или ошибку (`ERROR`) при несовпадении количества целевых переменных с количеством исходных.

`too_many_rows`

При включении этой проверки PL/pgSQL будет контролировать случаи, когда запрос с предложением `INTO` возвращает больше одной строки. Предложение `INTO` может обработать только одну строку, поэтому запрос, возвращающий несколько строк, как правило оказывается неэффективным и/или недетерминированным, а следовательно, скорее всего, является ошибочным.

Следующий пример показывает эффект присваивания `plpgsql.extra_warnings` значения `shadowed_variables`:

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
```

```
f1 int;
BEGIN
RETURN f1;
END;
$$ LANGUAGE plpgsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

Пример ниже показывает эффект присваивания `plpgsql.extra_warnings` значения `strict_multi_assignment`:

```
SET plpgsql.extra_warnings TO 'strict_multi_assignment';

CREATE OR REPLACE FUNCTION public.foo()
 RETURNS void
 LANGUAGE plpgsql
AS $$
DECLARE
  x int;
  y int;
BEGIN
  SELECT 1 INTO x, y;
  SELECT 1, 2 INTO x, y;
  SELECT 1, 2, 3 INTO x, y;
END;
$$;

SELECT foo();
WARNING:  number of source and target fields in assignment does not match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.
WARNING:  number of source and target fields in assignment does not match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.

 foo
-----

(1 row)
```

41.13. Портирование из Oracle PL/SQL

В этом разделе рассматриваются различия между языками Postgres Pro PL/pgSQL и Oracle PL/SQL, чтобы помочь разработчикам, переносящим приложения из Oracle® в Postgres Pro.

PL/pgSQL во многих аспектах похож на PL/SQL . Это блочно-структурированный, императивный язык, в котором все переменные должны объявляться. Присваивания, циклы и условные операторы в обоих языках похожи. Основные отличия, которые необходимо иметь в виду при портировании с PL/SQL в PL/pgSQL, следующие:

- Если имя, используемое в SQL-команде, может быть как именем столбца таблицы, фигурирующей в команде, так и ссылкой на переменную функции, то PL/SQL считает, что это имя столбца таблицы. Однако PL/pgSQL по умолчанию выдаёт ошибку, сообщая о такой неоднозначности. Установив `plpgsql.variable_conflict` = `use_column`, это поведение можно поменять на принятое в PL/SQL и описанное в Подразделе 41.11.1. В первую очередь, было бы правильно избегать таких двусмысленностей, но если требуется портировать большое количество кода, зависящее от данного поведения, то установка переменной `variable_conflict` может быть лучшим решением.

- В Postgres Pro тело функции должно быть записано в виде строки. Поэтому нужно использовать знак доллара в качестве кавычек или экранировать одиночные кавычки в теле функции. (См. Подраздел 41.12.1.)
- Имена типов данных часто требуют корректировки. Например, в Oracle строковые значения часто объявляются с типом `varchar2`, не являющимся стандартным типом SQL. В Postgres Pro вместо него нужно использовать `varchar` или `text`. Подобным образом, тип `number` нужно заменять на `numeric` или другой числовой тип, если найдётся более подходящий.
- Для группировки функций вместо пакетов используются схемы.
- Так как пакетов нет, нет и пакетных переменных. Это несколько раздражает. Вместо этого можно хранить состояние каждого сеанса во временных таблицах.
- Целочисленные циклы `FOR` с указанием `REVERSE` работают по-разному. В PL/SQL значение счётчика уменьшается от второго числа к первому, в то время как в PL/pgSQL счётчик уменьшается от первого ко второму. Поэтому при портировании нужно менять местами границы цикла. Это печально, но вряд ли будет изменено. (См. Подраздел 41.6.5.5.)
- Циклы `FOR` по запросам (не курсорам) также работают по-разному. Переменная цикла должна быть объявлена, в то время как в PL/SQL она объявляется неявно. Преимущество в том, что значения переменных доступны и после выхода из цикла.
- Существуют некоторые отличия в нотации при использовании курсорных переменных.

41.13.1. Примеры портирования

Пример 41.9 показывает, как портировать простую функцию из PL/SQL в PL/pgSQL.

Пример 41.9. Портирование простой функции из PL/SQL в PL/pgSQL

Функция Oracle PL/SQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
                                                  v_version varchar2)
RETURN varchar2 IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Пройдемся по этой функции и посмотрим различия по сравнению с PL/pgSQL:

- Имя типа `varchar2` нужно сменить на `varchar` или `text`. В примерах данного раздела мы будем использовать `varchar`, но обычно лучше выбрать `text`, если не требуется ограничивать длину строк.
- Ключевое слово `RETURN` в прототипе функции (не в теле функции) заменяется на `RETURNS` в Postgres Pro. Кроме того, `IS` становится `AS`, и нужно добавить предложение `LANGUAGE`, потому что PL/pgSQL — не единственный возможный язык.
- В Postgres Pro тело функции является строкой, поэтому нужно использовать кавычки или знаки доллара. Это заменяет завершающий `/` в подходе Oracle.
- Команда `show errors` не существует в Postgres Pro и не требуется, так как ошибки будут выводиться автоматически.

Вот как эта функция будет выглядеть после портирования в Postgres Pro:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
```

```
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Пример 41.10 показывает, как портировать функцию, которая создаёт другую функцию, и как обрабатывать проблемы с кавычками.

Пример 41.10. Портирование функции, создающей другую функцию, из PL/SQL в PL/pgSQL

Следующая процедура получает строки из `SELECT` и строит большую функцию, в целях эффективности возвращающую результат в операторах `IF`.

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR2,
                 v_domain IN VARCHAR2, v_url IN VARCHAR2) RETURN VARCHAR2 IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

В конечном итоге в Postgres Pro эта функция может выглядеть так:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc() AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
```

```
            || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
            || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Обратите внимание, что тело функции строится отдельно, с использованием `quote_literal` для дублирования кавычек. Эта техника необходима, потому что мы не можем безопасно использовать знаки доллара при определении новой функции: мы не знаем наверняка, какие строки будут вставлены из `referrer_key.key_string`. (Мы предполагаем, что `referrer_key.kind` всегда имеет значение из списка: `host`, `domain` или `url`, но `referrer_key.key_string` может быть чем угодно, в частности, может содержать знаки доллара.) На самом деле, в этой функций есть улучшение по сравнению с оригиналом Oracle, потому что не будет генерироваться неправильный код, когда `referrer_key.key_string` или `referrer_key.referrer_type` содержат кавычки.

Пример 41.11 показывает, как портировать функцию с выходными параметрами (`OUT`) и манипулирующую строками. В Postgres Pro нет встроенной функции `instr`, но её можно создать, используя комбинацию других функций. В Подраздел 41.13.3 приведена реализации `instr` на PL/pgSQL, которая может быть полезна вам при портировании ваших функций.

Пример 41.11. Портирование из PL/SQL в PL/pgSQL процедуры, которая манипулирует строками и содержит `OUT` параметры

Следующая процедура на языке Oracle PL/SQL разбирает URL и возвращает составляющие его элементы (сервер, путь и запрос).

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR2,
    v_host OUT VARCHAR2,  -- Возвращается как результат
    v_path OUT VARCHAR2,  -- И это тоже
    v_query OUT VARCHAR2) -- И это
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
```

```
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Вот возможная трансляция в PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- Возвращается как результат
    v_path OUT VARCHAR,  -- И это тоже
    v_query OUT VARCHAR) -- И это
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

Эту функцию можно использовать так:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Пример 41.12 показывает, как портировать процедуру, использующую большое количество специфических для Oracle возможностей.

Пример 41.12. Портирование процедуры из PL/SQL в PL/pgSQL

Версия Oracle:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- освободить блокировку
        raise_application_error(-20000,
                 'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- ничего не делать, если задание уже есть
    END;
    COMMIT;
END;
/
show errors
```

Вот как эту процедуру можно переписать на PL/pgSQL:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id integer) AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- освободить блокировку
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running'; -- 1
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN -- 2
            -- ничего не делать, если задание уже есть
    END;
    COMMIT;
```

```
END;
$$ LANGUAGE plpgsql;
```

1 Синтаксис `RAISE` существенно отличается от Oracle, хотя основной вариант `RAISE` *`имя_исключения`* работает похоже.

2 Имена исключений, поддерживаемые PL/pgSQL, отличаются от исключений в Oracle. Количество встроенных имён исключений значительно больше (см. Приложение A). В настоящее время нет способа задать пользовательское имя исключения, хотя вместо этого можно вызывать ошибку с заданным пользователем значением SQLSTATE.

41.13.2. На что ещё обратить внимание

В этом разделе рассматриваются ещё несколько вещей, на которые нужно обращать внимание при портировании функций из Oracle PL/SQL в Postgres Pro.

41.13.2.1. Неявный откат изменений после возникновения исключения

В PL/pgSQL при перехвате исключения в секции `EXCEPTION` все изменения в базе данных с начала блока автоматически откатываются. В Oracle это эквивалентно следующему:

```
BEGIN
    SAVEPOINT s1;
    ... здесь код ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... здесь код ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... здесь код ...
END;
```

При портировании процедуры Oracle, которая использует `SAVEPOINT` и `ROLLBACK TO` в таком же стиле, задача простая: достаточно убрать операторы `SAVEPOINT` и `ROLLBACK TO`. Если же `SAVEPOINT` и `ROLLBACK TO` используются по-другому, то придётся подумать.

41.13.2.2. `EXECUTE`

PL/pgSQL версия `EXECUTE` работает аналогично версии в PL/SQL, но нужно помнить об использовании `quote_literal` и `quote_ident`, как описано в Подразделе 41.5.4. Без использования этих функций конструкции типа `EXECUTE 'SELECT * FROM $1';` будут работать ненадёжно.

41.13.2.3. Оптимизация функций на PL/pgSQL

Для оптимизации исполнения Postgres Pro предоставляет два модификатора при создании функции: «изменчивость» (будет ли функция всегда возвращать тот же результат при тех же аргументах) и «строгость» (возвращает ли функция NULL, если хотя бы один из аргументов NULL). Для получения подробной информации обратитесь к справочной странице CREATE FUNCTION.

При использовании этих атрибутов оптимизации оператор `CREATE FUNCTION` может выглядеть примерно так:

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

41.13.3. Приложение

Этот раздел содержит код для совместимых с Oracle функций `instr`, которые можно использовать для упрощения портирования.

```
--
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2 [, n [, m]])
-- where [] denotes optional parameters.
--
-- Search string1, beginning at the nth character, for the mth occurrence
-- of string2.  If n is negative, search backwards, starting at the abs(n)'th
-- character from the end of string1.
-- If n is not passed, assume 1 (search starts at first character).
-- If m is not passed, assume 1 (find first occurrence).
-- Returns starting index of string2 in string1, or 0 if string2 is not found.
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
    RETURN instr($1, $2, 1);
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search_for IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
```

```
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument ''%'' is out of range', occur_index
          USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

Глава 42. PL/Tcl — процедурный язык Tcl

PL/Tcl — это загружаемый процедурный язык для СУБД Postgres Pro, позволяющий использовать *язык Tcl* для написания функций и процедур Postgres Pro.

42.1. Обзор

PL/Tcl предоставляет большинство возможностей, которые имеет разработчик функций на C, с небольшими ограничениями, и позволяет применять мощные библиотеки обработки строк, существующие для Tcl.

Одним убедительным *хорошим* ограничением является то, что весь код выполняется в контексте безопасности интерпретатора Tcl. Помимо ограниченного набора команд безопасного Tcl, разрешены только несколько команд для обращения к базе данных через SPI и вызовы `elog()` для выдачи сообщений. PL/Tcl не даёт возможности взаимодействовать с внутренним механизмом сервера баз данных или обращаться к ОС с правами серверного процесса Postgres Pro, что возможно в функциях на C. Таким образом, использование этого языка можно доверить непривилегированным пользователям; это не даст им неограниченные полномочия.

Ещё одно существенное ограничение заключается в том, что функции на Tcl нельзя использовать для создания функций ввода-вывода для новых типов данных.

Иногда возникает желание написать функцию на Tcl, которая не будут ограничена безопасным Tcl. Например, может потребоваться функция, которая будет посылать сообщения по почте. Для этих случаев есть вариация PL/Tcl, названная `PL/TclU` (название подразумевает «untrusted Tcl», недоверенный Tcl). Это тот же язык, за исключением того, что для него используется полноценный интерпретатор Tcl. *Если применяется PL/TclU, он должен быть установлен как недоверенный процедурный язык*, чтобы только суперпользователи могли создавать функции на нём. Автор функции на PL/TclU должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных.

Разделяемый объектный код для обработчиков вызова PL/Tcl и PL/TclU собирается автоматически и устанавливается в каталог библиотек Postgres Pro, если поддержка Tcl включена на этапе конфигурирования процедуры установки. Чтобы установить PL/Tcl и/или PL/TclU в конкретную базу данных, воспользуйтесь командой `CREATE EXTENSION`, например, так: `CREATE EXTENSION pltcl` или `CREATE EXTENSION pltclu`.

42.2. Функции на PL/Tcl и их аргументы

Чтобы создать функцию на языке PL/Tcl, используйте стандартный синтаксис CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (типы_аргументов) RETURNS тип_результата AS $$
    # Тело функции на PL/Tcl
$$ LANGUAGE pltcl;
```

С PL/TclU команда та же, но в качестве языка должно быть указано `pltclu`.

Тело функции содержит просто скрипт на Tcl. Когда вызывается функция, значения аргументов передаются скрипту Tcl в виде переменных с именами `1` ... *n*. Результат из кода Tcl возвращается как обычно, оператором `return`. В процедуре значение, возвращаемое из кода Tcl, игнорируется.

Например, функцию, возвращающую большее из двух целых чисел, можно определить так:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Обратите внимание на предложение `STRICT`, которое избавляет нас от необходимости думать о входящих значениях NULL: если при вызове передаётся значение NULL, функция не будет выполняться вовсе, будет сразу возвращён результат NULL.

В нестрогой функции, если фактическое значение аргумента — NULL, соответствующей переменной `$n` будет присвоена пустая строка. Чтобы определить, был ли передан NULL в определённом аргументе, используйте функцию `argisnull`. Например, предположим, что нам нужна функция `tcl_max`, которая с одним аргументом NULL и вторым аргументом не NULL должна возвращать не NULL, а второй аргумент:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

Как показано выше, чтобы вернуть значение NULL из функции PL/Tcl, нужно выполнить `return_null`. Это можно сделать и в строгой, и в нестрогой функции.

Аргументы составного типа передаются функции в виде массивов Tcl. Именами элементов массива являются имена атрибутов составного типа. Если атрибут в переданной строке имеет значение NULL, он будет отсутствовать в данном массиве. Например:

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

Функции PL/Tcl могут возвращать и результаты составного типа. Для этого код на Tcl должен вернуть список пар имя/значение столбца, соответствующий ожидаемому типу результата. Столбцы, имена которых в этом списке отсутствуют, получат значения NULL, а если в списке указано имя несуществующего столбца, возникнет ошибка. Например:

```
CREATE FUNCTION square_cube(in int, out squared int, out cubed int) AS $$
    return [list squared [expr {$1 * $1}] cubed [expr {$1 * $1 * $1}]]
$$ LANGUAGE pltcl;
```

Выходные аргументы процедур возвращаются таким же образом. Например:

```
CREATE PROCEDURE tcl_triple(INOUT a integer, INOUT b integer) AS $$
    return [list a [expr {$1 * 3}] b [expr {$2 * 3}]]
$$ LANGUAGE pltcl;

CALL tcl_triple(5, 10);
```

Подсказка

Список результатов можно создать из желаемого кортежа, представленного в виде массива, с помощью команды `array get` языка Tcl. Например:

```
CREATE FUNCTION raise_pay(employee, delta int) RETURNS employee AS $$
    set 1(salary) [expr {$1(salary) + $2}]
    return [array get 1]
$$ LANGUAGE pltcl;
```

Функции PL/Tcl могут возвращать наборы результатов. Для этого код на Tcl должен вызывать `return_next` для каждой возвращаемой строки, передавая ей соответствующее значение, когда возвращается скалярный тип, или список пар имя/значение столбца, когда возвращается составной тип. Пример с результатом скалярного типа:

```
CREATE FUNCTION sequence(int, int) RETURNS SETOF int AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next $i
    }
$$ LANGUAGE pltcl;
```

и с результатом составного:

```
CREATE FUNCTION table_of_squares(int, int) RETURNS TABLE (x int, x2 int) AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next [list x $i x2 [expr {$i * $i}]]
    }
$$ LANGUAGE pltcl;
```

42.3. Значения данных в PL/Tcl

Значения аргументов, передаваемые в код функции PL/Tcl, представляют собой просто входные аргументы, преобразованные в текстовый вид (так же, как при выводе оператором `SELECT`). И наоборот, команды `return` и `return_next` примут любую строку, соответствующую формату ввода для объявленного типа результата функции или заданного столбца в результате составного типа.

42.4. Глобальные данные в PL/Tcl

Иногда полезно иметь некоторые глобальные данные, сохраняемые между двумя вызовами функции или совместно используемые разными функциями. Это легко сделать в PL/Tcl, но есть некоторые ограничения, которые необходимо понимать.

По соображениям безопасности, PL/Tcl выполняет функции, вызываемые некоторой ролью SQL в отдельном интерпретаторе Tcl, выделенном для этой роли. Это предотвращает случайное или злонамеренное влияние одного пользователя на поведение функций PL/Tcl другого пользователя. В каждом интерпретаторе будут свои значения всех «глобальных» переменных Tcl. Таким образом, в двух функциях PL/Tcl будут общие глобальные переменные, только если они выполняются одной ролью SQL. В приложении, выполняющем код в одном сеансе с разными ролями SQL (вызывающем функции `SECURITY DEFINER`, использующем команду `SET ROLE` и т. д.) может понадобиться явно предпринять дополнительные меры, чтобы функции могли разделять свои данные. Для этого сначала установите для функций, которые должны взаимодействовать, одного владельца, а затем задайте для них свойство `SECURITY DEFINER`. Разумеется, при этом нужно позаботиться о том, чтобы эти функции не могли сделать ничего непредусмотренного.

Все функции PL/TclU, вызываемые в одном сеансе, выполняются одним интерпретатором Tcl, который, конечно, отличается от интерпретатора(ов), используемого для функций PL/Tcl. Поэтому глобальные данные функций PL/TclU автоматически становятся общими. Это не считается угрозой безопасности, так как все функции PL/TclU выполняются на одном уровне доверия, а именно уровне суперпользователя базы данных.

Чтобы защитить функции PL/Tcl от непреднамеренного влияния друг на друга, каждой из них предоставляется глобальная переменная-массив через команду `upvar`. Глобальным именем этой переменной является внутреннее имя функции, а в качестве локального выбрано `GD`. Переменную `GD` рекомендуется использовать для постоянных внутренних данных функции. Обычные глобальные переменные Tcl следует использовать только для значений, которые предназначены именно для совместного использования несколькими функциями. (Заметьте, что массивы `GD` являются глобальными только для конкретного интерпретатора, так что они не нарушают ограничения безопасности, описанные выше.)

Использование `GD` демонстрируется в примере `spi_execp`, приведённом ниже.

42.5. Обращение к базе данных из PL/Tcl

В этом разделе для обозначения необязательных элементов в описании синтаксиса используются знаки вопроса (а не квадратные скобки), как это принято в Tcl. Имеются следующие команды для доступа к базе данных из тела функции PL/Tcl:

`spi_exec ?-count` *`n`*`? ?-array` *`имя`*`?` *`команда`* `?`*`тело-цикла`*`?`

Выполняет команду SQL, заданную в виде строки. В случае ошибки в этой команде выдаётся ошибка в Tcl. В противном случае `spi_exec` возвращает число обработанных командой строк (выбранных, добавленных, изменённых или удалённых), либо ноль, если эта команда — служебный оператор. Кроме того, если команда — оператор `SELECT`, значения выбранных столбцов помещаются в переменные Tcl, как описано ниже.

Необязательный аргумент `-count` указывает функции `spi_exec` остановиться после получения *`n`* строк, как если бы запрос включал предложение `LIMIT`. Если *`n`* равно нулю, запрос выполняется до конца, так же, как без указания `-count`.

Если в качестве команды выполняется оператор `SELECT`, значения результирующих столбцов помещаются в переменные Tcl, названные по именам столбцов. Если передаётся `-array`, значения столбцов вместо этого становятся элементами названного ассоциативного массива, индексами в котором становятся имена столбцов. Кроме того, в элементе с именем «`.tupno`» сохраняется номер текущей строки в результирующем наборе (отсчитывая от нуля), если только это имя не занято одним из столбцов результата.

Если в качестве команды выполняется `SELECT` без указания скрипта *`тело-цикла`*, в переменных Tcl или элементах массива сохраняется только первая строка результатов; оставшиеся строки (если они есть), игнорируются. Если запрос не возвращает строки, не сохраняется ничего. (Этот случай можно отследить, проверив результат `spi_exec`.) Например, команда:

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

присвоит переменной `$cnt` в Tcl число строк, содержащихся в системном каталоге `pg_proc`.

Если передаётся необязательный аргумент *`тело-цикла`*, заданный в нём блок скрипта Tcl будет выполняться для каждой строки результата запроса. (Аргумент *`тело-цикла`* игнорируется, если целевая команда — не `SELECT`.) При этом значения столбцов текущей строки сохраняются в переменных Tcl или элементах массива перед каждой итерацией этого цикла. Например, код:

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

будет выводить в журнал сообщение для каждой строки `pg_class`. Это работает подобно другим конструкциям циклов в Tcl; в частности, команды `continue` и `break` в теле цикла будут действовать обычным образом.

Если в столбце результата запроса выдаётся NULL, целевая переменная для неё не устанавливается, и оказывается «неустановленной».

`spi_prepare` *`query`* *`список-типов`*

Подготавливает и сохраняет план запроса для последующего выполнения. Сохранённый план будет продолжать существование до завершения текущего сеанса.

Запрос может принимать параметры, то есть местозаполнители для значений, которые будут передаваться, когда план будет собственно выполняться. В строке запроса эти параметры обозначаются как `$1` ... `$n`. Если в запросе используются параметры, нужно задать имена типов этих параметров в виде списка Tcl. (Если параметры отсутствуют, задайте пустой *`список_типов`*.)

Функция `spi_prepare` возвращает идентификатор запроса, который может использоваться в последующих вызовах `spi_execp`. Пример приведён в описании `spi_execp`.

`spi_execp ?-count n? ?-array имя? ?-nulls строка? ид-запроса ?список-значений? ?тело-цикла?`

Выполняет запрос, ранее подготовленный функцией `spi_prepare`. В качестве *`ид_запроса`* передаётся идентификатор, возвращённый функцией `spi_prepare`. Если в запросе задействуются параметры, необходимо указать *`список-значений`*. Это должен быть принятый в Tcl список параметров. Он должен иметь ту же длину, что и список типов параметров, ранее переданный `spi_prepare`. Опустите *`список-значений`*, если у запроса нет параметров.

Необязательный аргумент `-nulls` принимает строку из пробелов и символов `'n'`, которые отмечают, в каких параметрах `spi_execp` передаются значения NULL. Если присутствует, эта строка должна иметь ту же длину, что и *`список-значений`*. В случае её отсутствия значения всех параметров считаются отличными от NULL.

Не считая отличий в способе передачи запроса и параметров, `spi_execp` работает так же, как `spi_exec`. Параметры `-count`, `-array` и *`тело-цикла`* задаются так же, и так же передаётся возвращаемое значение.

Взгляните на пример функции на PL/Tcl, использующей подготовленный план:

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # подготовить сохранённый план при первом вызове
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

Обратные косые черты внутри строки запроса, передаваемой функции `spi_prepare`, нужны для того, чтобы маркеры `$n` передавались функции `spi_prepare` как есть, а не заменялись при подстановке переменных Tcl.

`subtransaction команда`

Скрипт Tcl, который содержит *`команда`*, выполняется в подтранзакции SQL. Если этот скрипт возвращает ошибку, вся подтранзакция откатывается назад, а затем в окружающий код Tcl возвращается ошибка. За дополнительными подробностями и примером обратитесь к Разделу 42.9.

`quote строка`

Дублирует все вхождения апострофа и обратной косой черты в заданной строке. Это можно использовать для защиты строк, которые будут вставляться в команды SQL, передаваемые в `spi_exec` или `spi_prepare`. Например, представьте, что при выполнении такой команды SQL:

```
"SELECT '$val' AS ret"
```

переменная языка Tcl `val` содержит `doesn't`. Это приведёт к формированию такой окончательной строки команды:

```
SELECT 'doesn't' AS ret
```

при разборе которой в процессе `spi_exec` или `spi_prepare` возникнет ошибка. Чтобы этот запрос работал правильно, итоговая команда должна выглядеть так:

```
SELECT 'doesn''t' AS ret
```

Получить её в PL/Tcl можно так:

```
"SELECT '[ quote $val ]' AS ret"
```

Преимуществом `spi_execp` является то, что для неё заключать значения параметров в кавычки подобным образом не нужно, так как параметры никогда не разбираются в составе строки команды SQL.

`elog` *`уровень сообщение`*

Выдаёт служебное сообщение или сообщение об ошибке. Возможные уровни сообщений: `DEBUG` (ОТЛАДКА), `LOG` (СООБЩЕНИЕ), `INFO` (ИНФОРМАЦИЯ), `NOTICE` (ЗАМЕЧАНИЕ), `WARNING` (ПРЕДУПРЕЖДЕНИЕ), `ERROR` (ОШИБКА) и `FATAL` (ВАЖНО). С уровнем `ERROR` выдаётся ошибка; если она не перехватывается окружающим кодом Tcl, она распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. По сути то же самое делает команда `error` языка Tcl. Сообщение уровня `FATAL` прерывает транзакцию и приводит к завершению текущего сеанса. (Вероятно, нет обоснованной причины использовать этот уровень ошибок в функциях PL/Tcl, но он поддерживается для полноты.) При использовании других уровней происходит просто вывод сообщения с заданным уровнем важности. Будут ли сообщения определённого уровня передаваться клиенту и/или записываться в журнал, определяется конфигурационными переменными log_min_messages и client_min_messages. За дополнительными сведениями обратитесь к Главе 18 и Разделу 42.8.

42.6. Триггерные функции на PL/Tcl

На PL/Tcl можно написать триггерные функции. Postgres Pro требует, чтобы функция, которая будет вызываться как триггерная, была объявлена как функция без аргументов и возвращала тип `trigger`.

Информация от менеджера триггеров передаётся в тело функции в следующих переменных:

`$TG_name`

Имя триггера из оператора `CREATE TRIGGER`.

`$TG_relid`

Идентификатор объекта таблицы, для которой будет вызываться триггерная функция.

`$TG_table_name`

Имя таблицы, для которой будет вызываться триггерная функция.

`$TG_table_schema`

Схема таблицы, для которой будет вызываться триггерная функция.

`$TG_relatts`

Список языка Tcl, содержащий имена столбцов таблицы. В начало списка добавлен пустой элемент, поэтому при поиске в этом списке имени столбца с помощью стандартной в Tcl команды `lsearch` будет возвращён номер элемента, начиная с 1, так же, как нумеруются столбцы в Postgres Pro. (В позициях удалённых столбцов также содержатся пустые элементы, так что нумерация следующих за ними атрибутов не нарушается.)

`$TG_when`

Строка `BEFORE`, `AFTER` или `INSTEAD OF`, в зависимости от типа события триггера.

`$TG_level`

Строка `ROW` или `STATEMENT`, в зависимости от уровня события триггера.

`$TG_op`

Строка `INSERT`, `UPDATE`, `DELETE` или `TRUNCATE`, в зависимости от действия события триггера.

`$NEW`

Ассоциативный массив, содержащий значения новой строки таблицы для действий `INSERT` или `UPDATE`, либо пустой массив для `DELETE`. Индексами в массиве являются имена столбцов. Столбцы со значениями NULL в нём отсутствуют. Для триггеров уровня оператора этот массив не определяется.

`$OLD`

Ассоциативный массив, содержащий значения старой строки таблицы для действий `UPDATE` или `DELETE`, либо пустой массив для `INSERT`. Индексами в массиве являются имена столбцов. Столбцы со значениями NULL в нём отсутствуют. Для триггеров уровня оператора этот массив не определяется.

`$args`

Список на языке Tcl аргументов функции, заданных в операторе `CREATE TRIGGER`. Эти аргументы также доступны под обозначениями `$1` ... `$n` в теле функции.

Возвращаемым значением триггерной функции может быть строка `OK` или `SKIP` либо список пар имя столбца/значение. Если возвращается значение `OK`, операция (`INSERT`/`UPDATE`/`DELETE`), которая привела к срабатыванию триггера, выполняется нормально. Значение `SKIP` указывает менеджеру триггеров просто пропустить эту операцию с текущей строкой данных. Если возвращается список, через него PL/Tcl передаёт менеджеру триггеров изменённую строку; содержимое изменённой строки задаётся именами и значениями столбцов в списке. Все столбцы, не перечисленные в этом списке, получают значения NULL. Возвращать изменённую строку имеет смысл только для триггеров уровня строки с порядком `BEFORE` команд `INSERT` и `UPDATE`, в которых вместо заданной в `$NEW` будет записываться изменённая строка; либо с порядком `INSTEAD OF` команд `INSERT` и `UPDATE`, в которых возвращаемая строка служит исходными данными для предложений `INSERT RETURNING` или `UPDATE RETURNING`. В триггерах уровня строки с порядком `BEFORE` или `INSTEAD OF` команды `DELETE` возврат изменённой строки воспринимается так же, как и возврат значения `OK`, то есть операция выполняется. Для всех остальных типов триггеров возвращаемое значение игнорируется.

Подсказка

Список результатов можно создать из изменённого кортежа, представленного в виде массива, с помощью команды `array get` языка Tcl.

Следующий небольшой пример показывает триггерную функцию, которая ведёт в таблице целочисленный счётчик числа изменений, выполненных в строке. Для новых строк счётчик инициализируется нулевым значением, а затем увеличивается на единицу при каждом изменении.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
```

```
$$ LANGUAGE pltcl;

CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE FUNCTION trigfunc_modcount('modcnt');
```

Заметьте, что сама триггерная функция не знает имени столбца; оно передаётся в аргументах триггера. Это позволяет применять эту функцию для различных таблиц.

42.7. Функции событийных триггеров в PL/Tcl

На PL/Tcl можно написать функции событийных триггеров. Postgres Pro требует, чтобы функция, которая будет вызываться как событийный триггер, была объявлена как функция без аргументов и возвращала тип `event_trigger`.

Информация от менеджера триггеров передаётся в тело функции в следующих переменных:

`$TG_event`

Имя события, при котором срабатывает этот триггер.

`$TG_tag`

Тег команды, для которой срабатывает этот триггер.

Возвращаемое значение триггерной функции игнорируется.

В этом примере мини-функция событийного триггера просто выдаёт замечание (`NOTICE`) при каждом выполнении поддерживаемой команды:

```
CREATE OR REPLACE FUNCTION tclsnitch() RETURNS event_trigger AS $$
  elog NOTICE "tclsnitch: $TG_event $TG_tag"
$$ LANGUAGE pltcl;

CREATE EVENT TRIGGER tcl_a_snitch ON ddl_command_start EXECUTE FUNCTION tclsnitch();
```

42.8. Обработка ошибок в PL/Tcl

Tcl-код, содержащийся или вызываемый из функции PL/Tcl, может выдавать ошибку либо выполняя недопустимую операцию, либо генерируя ошибку с помощью команды `error` языка Tcl или команды `elog` языка PL/Tcl. Такие ошибки могут быть перехвачены в среде Tcl с помощью команды Tcl `catch`. Если ошибка не перехватывается, а распространяется выше уровня выполнения функций PL/Tcl, она передаётся в запрос, вызвавший функцию, как ошибка SQL.

И напротив, ошибки СУБД, возникающие внутри команд `spi_exec`, `spi_prepare` и `spi_execp` в среде PL/Tcl, выдаются как ошибки Tcl, так что их можно перехватить командой Tcl `catch`. (Каждая из этих команд PL/Tcl выполняет SQL-операцию в подтранзакции, которая откатывается в случае ошибки, так что для частично завершённых операций производится автоматическая очистка.) Опять же, если ошибка не перехватывается и распространяется выше верхнего уровня, она становится ошибкой SQL.

В Tcl имеется переменная `errorCode`, представляющая дополнительную информацию об ошибке в виде, удобном для обработки в программах на Tcl. Эта информация передаётся в формате списка Tcl, первое слово в котором указывает на подсистему или библиотеку, выдающую ошибку; последующее содержимое определяется в зависимости от подсистемы или библиотеки. Для ошибок СУБД, возникающих в командах PL/Tcl, первым словом будет `POSTGRES`, вторым — номер версии Postgres Pro, а дополнительные слова представляют пары имя/значения, передающие подробную информацию об ошибке. В этих парах всегда передаются поля `SQLSTATE`, `condition` и `message` (первые два представляют код ошибки и имя условия, как описано в Приложении A). Также могут передаваться поля `detail`, `hint`, `context`, `schema`, `table`, `column`, `datatype`, `constraint`, `statement`, `cursor_position`, `filename`, `lineno` и `funcname`.

С информацией в переменной `errorCode` среды PL/Tcl удобно работать, загрузив переменную в массив, чтобы имена полей стали индексами в массиве. Пример такого кода:

```
if {[catch { spi_exec $sql_command }]} {
    if {[lindex $::errorCode 0] == "POSTGRES"} {
        array set errorArray $::errorCode
        if {$errorArray(condition) == "undefined_table"} {
            # разобраться с отсутствием таблицы
        } else {
            # разобраться с другими типами ошибок SQL
        }
    }
}
```

(Двойные двоеточия явно указывают, что переменная `errorCode` является глобальной.)

42.9. Явные подтранзакции в PL/Tcl

Перехват ошибок, произошедших при обращении к базе данных, как описано в Разделе 42.8, может привести к нежелательной ситуации, когда часть операций будет успешно выполнена, прежде чем произойдёт сбой. Данные останутся в несогласованном состоянии после обработки такой ошибки. PL/Tcl предлагает решение этой проблемы в форме явных подтранзакций.

Рассмотрите функцию, реализующую перевод денег между двумя счетами:

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
    if [catch {
        spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'"
        spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'"
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

Если второй оператор `UPDATE` выдаст исключение, эта функция запишет в журнал сообщение об ошибке, но результат первого `UPDATE` будет тем не менее зафиксирован. Другими словами, денежные средства будут списаны со счёта Джо, но не поступят на счёт Мери. Это происходит потому, что каждый вызов `spi_exec` выполняется в отдельной подтранзакции, а откатывается только одна из подтранзакций.

В таких случаях вы можете обернуть несколько операций с базой данных в одну явную подтранзакцию, которая будет выполнена успешно или отменена как единое целое. Для этого в PL/Tcl есть команда `subtransaction`. С ней мы можем переписать нашу функцию так:

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
    if [catch {
        subtransaction {
            spi_exec "UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'"
            spi_exec "UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'"
        }
    } errormsg] {
        set result [format "error transferring funds: %s" $errormsg]
    } else {
        set result "funds transferred successfully"
```

```
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote $result]')"
$$ LANGUAGE pltcl;
```

Заметьте, что и в этом случае нужно использовать `catch`. В противном случае ошибка распространится на верхний уровень функции, что не даст произвести желаемое добавление записи в таблицу `operations`. Команда `subtransaction` не перехватывает ошибки, она только обеспечивает откат всех операций с базой данных в своей области действия в случае ошибки.

Откат явной подтранзакции происходит в случае любых ошибок, сгенерированных вложенным кодом Tcl, а не только ошибок, возникающих при обращении к базе данных. Таким образом, обычное исключение Tcl, возникшее внутри команды `subtransaction`, также приведёт к откату подтранзакции. Однако при выходе из вложенного кода Tcl без ошибки (например, с помощью команды `return`) откат не производится.

42.10. Управление транзакциями

В процедуре, которая вызывается в коде верхнего уровня или в анонимном блоке кода (в команде `DO`), можно управлять транзакциями. Чтобы зафиксировать текущую транзакцию, выполните команду `commit`, а чтобы откатить — `rollback`. (Заметьте, что выполнить SQL-команды `COMMIT` или `ROLLBACK` через `spi_exec` или подобную функцию нельзя. Соответствующие операции могут выполняться только данными функциями.) После завершения одной транзакции следующая начинается автоматически, отдельной функции для этого нет.

Пример:

```
CREATE PROCEDURE transaction_test1()
LANGUAGE pltcl
AS $$
for {set i 0} {$i < 10} {incr i} {
    spi_exec "INSERT INTO test1 (a) VALUES ($i)"
    if {$i % 2 == 0} {
        commit
    } else {
        rollback
    }
}
$$;

CALL transaction_test1();
```

Транзакции не могут завершаться, когда имеется открытая явная подтранзакция.

42.11. Конфигурация PL/Tcl

В этом разделе описываются параметры конфигурации, влияющие на работу PL/Tcl.

`pltcl.start_proc` (`string`)

В этом параметре, если он не пуст, задаётся имя (возможно, дополненное схемой) функции на языке PL/Tcl без параметров, которая будет выполняться, когда для PL/Tcl будет создаваться новый экземпляр Tcl. Такая функция может выполнять инициализацию в рамках сеанса, например, загружать дополнительный код Tcl. Новый интерпретатор Tcl создаётся при первом выполнении какой-либо функции PL/Tcl в сеансе базы данных или когда требуется дополнительный интерпретатор из-за того, что функция PL/Tcl была вызвана новой ролью SQL.

Указанная функция должна быть написана на языке `pltcl` и не должна иметь свойство `SECURITY DEFINER`. (Благодаря этим ограничениям эта функция будет запускаться в интерпретаторе, который она должна инициализировать.) Текущий пользователь должен иметь право и на её выполнение тоже.

Если эта функция завершится ошибкой, эта ошибка прервёт вызов функции, которой потребовался новый интерпретатор, и распространится в вызывающий запрос, приводя к прерыванию текущей транзакции или подтранзакции. Любые действия, уже произведённые в среде Tcl, отменены не будут; однако этот интерпретатор более не будет использоваться. При следующей попытке использования этого языка последует повторная попытка инициализации со свежим интерпретатором Tcl.

Изменять этот параметр разрешено только суперпользователям. Хотя изменить его можно в рамках сеанса, такие изменения не повлияют на работу интерпретаторов Tcl, созданных ранее.

`pltclu.start_proc` (`string`)

Это параметр полностью аналогичен `pltcl.start_proc`, но применяется к PL/TclU. Указанная функция должна быть написана на языке `pltclu`.

42.12. Имена процедур Tcl

В Postgres Pro одно имя функции может использоваться разными определениями функций, если они имеют разное число и типы аргументов. Tcl, однако, требует, чтобы имена всех процедур различались. PL/Tcl решает эту проблему, устанавливая такие внутренние имена процедур Tcl, чтобы они включали в свой состав OID функции из системной таблицы `pg_proc`. Таким образом, функциям Postgres Pro с одним именем и разными типами аргументов так же будут соответствовать различные процедуры Tcl. Это обычно остаётся незамеченным для программиста PL/Tcl, но может проявиться при отладке.

Глава 43. PL/Perl — процедурный язык Perl

PL/Perl — это загружаемый процедурный язык, позволяющий реализовывать функции и процедуры Postgres Pro на *языке программирования Perl*.

Основным преимуществом PL/Perl является то, что он позволяет применять в сохранённых функциях и процедурах множество функций и операторов «перемалывания строк», имеющихся в Perl. Разобрать сложные строки на языке Perl может быть гораздо проще, чем используя строковые функции и управляющие структуры в PL/pgSQL.

Чтобы установить PL/Perl в определённую базу данных, выполните команду `CREATE EXTENSION plperl`.

> **Подсказка**
>
> Если язык устанавливается в `template1`, он будет автоматически установлен во все создаваемые впоследствии базы данных.

> **Примечание**
>
> Пользователи, имеющие дело с исходным кодом, должны явно включить сборку PL/Perl в процессе установки. Пользователи двоичных пакетов могут найти PL/Perl в отдельном модуле.

43.1. Функции на PL/Perl и их аргументы

Чтобы создать функцию на языке PL/Perl, используйте стандартный синтаксис CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (типы-аргументов)
RETURNS тип-результата
-- здесь описываются атрибуты функции
AS $$
    # Тело функции на PL/Perl
$$ LANGUAGE plperl;
```

Тело функции содержит обычный код Perl. Фактически, код обвязки PL/Perl помещает этот код в подпрограмму Perl. Функция PL/Perl вызывается в скалярном контексте, так что она не может вернуть список. Не скалярные значения (массивы, записи и множества) можно вернуть по ссылке, как описывается ниже.

В процедуре PL/Perl возвращаемое из кода Perl значение игнорируется.

PL/Perl также поддерживает анонимные блоки кода, которые выполняются оператором DO:

```
DO $$
    # Код PL/Perl
$$ LANGUAGE plperl;
```

Анонимный блок кода не принимает аргументы, а любое значение, которое он мог бы вернуть, отбрасывается. В остальном он работает подобно коду функции.

> **Примечание**
>
> Использовать вложенные именованные подпрограммы в Perl опасно, особенно если они обращаются к лексическим переменным в окружающей области. Так как функция PL/Perl оборачивается в подпрограмму, любая именованная функция внутри неё будет вложенной. Вообще гораздо безопаснее создавать анонимные подпрограммы и вызывать их по ссылке на код. Дополнительную информацию вы можете получить на странице руководства man perldiag, в описании ошибок `Variable "%s" will not stay shared` (Переменная "%s" не останется

разделяемой) и `Variable "%s" is not available` (Переменная "%s" недоступна), либо найти в Интернете по ключевым словам «perl nested named subroutine» (perl вложенная именованная подпрограмма).

Синтаксис команды `CREATE FUNCTION` требует, чтобы тело функции было записано как строковая константа. Обычно для этого удобнее всего заключать строковую константу в доллары (см. Подраздел 4.1.2.4). Если вы решите применять синтаксис спецпоследовательностей `E''`, вам придётся дублировать апострофы (`'`) и обратную косую черту (`\`) в теле функции (см. Подраздел 4.1.2.1).

Аргументы и результат обрабатываются как и в любой другой подпрограмме на Perl: аргументы передаются в `@_`, а результирующим значением будет указанное в `return` или полученное в последнем выражении, вычисленном в функции.

Например, функцию, возвращающую большее из двух целых чисел, можно определить так:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

Примечание

Аргументы будут преобразованы из кодировки базы данных в UTF-8 для использования в PL/Perl, а при выходе снова будут преобразованы из UTF-8 в кодировку базы данных.

Если функции передаётся NULL-значение SQL, значением аргумента в Perl станет «undefined». Показанное выше определение функции будет не очень хорошо обрабатывать значения NULL (в действительности они будут восприняты как нули). Мы могли бы добавить указание `STRICT` в это определение, чтобы Postgres Pro поступал немного разумнее: при передаче значения NULL функция вовсе не будет вызываться, будет сразу возвращён результат NULL. С другой стороны, мы могли бы проверить значения undefined в теле функции. Например, предположим, что нам нужна функция `perl_max`, которая с одним аргументом NULL и вторым аргументом не NULL должна возвращать не NULL, а второй аргумент:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
    return $y;
$$ LANGUAGE plperl;
```

Как показано выше, чтобы выдать значение SQL NULL, нужно вернуть значение undefined. Это можно сделать и в строгой, и в нестрогой функции.

Всё в аргументах функции, что не является ссылкой, является строкой, то есть стандартным для Postgres Pro внешним текстовым представлением соответствующего типа данных. В случае с обычными числовыми или текстовыми типами, Perl просто воспринимает их должным образом, и программист, как правило, может об этом не думать. Однако в более сложных случаях может потребоваться преобразовать аргумент в форму, подходящую для использования в Perl. Например, для преобразования типа `bytea` в двоичное значение можно использовать функцию `decode_bytea`.

Аналогично, значения, передаваемые в Postgres Pro, должны быть в формате внешнего текстового представления. Например, для подготовки двоичных данных к возврату в значении `bytea` можно воспользоваться функцией `encode_bytea`.

Особого внимания заслуживает поведение с логическими значениями. Как только что было сказано, по умолчанию значения `bool` передаются в Perl в текстовом виде, то есть как `'t'` или `'f'`. И здесь возникает проблема, так как Perl не будет воспринимать `'f'` как false! Улучшить ситуацию можно, воспользовавшись «трансформацией» (см. CREATE TRANSFORM). Нужные трансформации реализованы в расширении `bool_plperl`. Чтобы применить их, установите расширение:

```
CREATE EXTENSION bool_plperl;  -- или bool_plperlu для PL/PerlU
```

Затем используйте атрибут `TRANSFORM` для функции на PL/Perl, которая принимает или выдаёт `bool`, например:

```
CREATE FUNCTION perl_and(bool, bool) RETURNS bool
TRANSFORM FOR TYPE bool
AS $$
  my ($a, $b) = @_;
  return $a && $b;
$$ LANGUAGE plperl;
```

Когда будет применяться эта трансформация, Perl будет получать аргументы `bool` как `1` или пустое значение, что для Perl будет выглядеть как true или false. Если функция возвращает результат типа `bool`, будет выдаваться значение true или false, в зависимости от того, считается ли в Perl результат истинным или нет. Подобные трансформации также выполняются для аргументов и результатов SPI-запросов, выполняемых внутри функции (Подраздел 43.3.1).

Perl может возвращать массивы PostgreSQL как ссылки на массивы Perl. Например, так:

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

Perl передаёт массивы PostgreSQL как объект, сопоставленный с `PostgreSQL::InServer::ARRAY`. С этим объектом можно работать как со ссылкой на массив или строкой, что допускает обратную совместимость с кодом Perl, написанным для PostgreSQL версии до 9.1. Например:

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
    my $arg = shift;
    my $result = "";
    return undef if (!defined $arg);

    # в качестве ссылки на массив
    for (@$arg) {
        $result .= $_;
    }

    # также работает со строкой
    $result .= $arg;

    return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

Примечание

Многомерные массивы представляются как ссылки на массивы меньшей размерности со ссылками — этот способ хорошо знаком каждому программисту на Perl.

Аргументы составного типа передаются функции как ссылки на хеши. Ключами хеша являются имена атрибутов составного типа. Например:

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

Функция на PL/Perl может вернуть результат составного типа, применяя тот же подход: возвратить ссылку на хеш с требуемыми атрибутами. Например, так:

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

Столбцы объявленного типа результата, отсутствующие в хеше, будут возвращены как значения NULL.

Подобным образом в виде ссылки на хеш могут быть возвращены выходные аргументы процедуры:

```
CREATE PROCEDURE perl_triple(INOUT a integer, INOUT b integer) AS $$
    my ($a, $b) = @_;
    return {a => $a * 3, b => $b * 3};
$$ LANGUAGE plperl;

CALL perl_triple(5, 10);
```

Функции на PL/Perl могут также возвращать множества со скалярными или составными типами. Обычно желательно возвращать результат по одной строке, чтобы сократить время подготовки с одной стороны, и чтобы не потребовалось накапливать весь набор данных в памяти, с другой. Это можно реализовать с помощью функции `return_next`, как показано ниже. Заметьте, что после последнего вызова `return_next`, нужно поместить `return` или (что лучше) `return undef`.

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
```

```
$$ LANGUAGE plperl;
```

Для небольших наборов данных можно также вернуть ссылку на массив, содержащий скаляры, ссылки на массивы, либо ссылки на хеши для простых типов, типов массивов и составных типов, соответственно. Ниже приведена пара простых примеров, показывающих, как возвратить весь набор данных в виде ссылки на массив:

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

Если вы хотите использовать в своём коде `strict`, у вас есть несколько вариантов. Для временного глобального использования вы можете задать для `plperl.use_strict` значение true командой `SET`. Это повлияет на компилируемые впоследствии функции PL/Perl, но не на функции, уже скомпилированные в текущем сеансе. Для постоянного глобального использования вы можете присвоить параметру `plperl.use_strict` значение true в файле `postgresql.conf`.

Для постоянного использования strict в определённых функциях вы можете просто написать:

```
use strict;
```

в начале тела этих функций.

Вы также можете использовать указания `feature` в `use`, если используете Perl версии 5.10.0 или новее.

43.2. Значения в PL/Perl

Значения аргументов, передаваемые в код функции PL/Perl, представляют собой просто входные аргументы, преобразованные в текстовый вид (так же, как при выводе оператором `SELECT`). И наоборот, команды `return` и `return_next` могут принять любую строку, соответствующую формату ввода для объявленного типа результата функции.

Если это поведение в каких-то случаях не устраивает, его можно улучшить, воспользовавшись трансформацией, что было проиллюстрировано выше на примере значений `bool`. Несколько примеров модулей трансформации включены в состав дистрибутива PostgreSQL.

43.3. Встроенные функции

43.3.1. Обращение к базе данных из PL/Perl

Обращаться к самой базе данных из кода Perl можно, используя следующие функции:

`spi_exec_query(`*`query`*` [, `*`предел`*`])`

Функция `spi_exec_query` выполняет команду SQL и возвращает весь набор строк в виде ссылки на массив хешей. Если *предел* задан и больше нуля, то `spi_exec_query` получает число строк, не превышающее *предел*, как если бы запрос включал предложение `LIMIT`. Без указания *предела* или когда он равен нулю, ограничение на количество строк снимается.

Эту функцию следует использовать, только если вы знаете, что набор будет относительно небольшим. Так выглядит пример запроса (`SELECT`) с дополнительно заданным максимальным числом строк:

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

Этот запрос возвращает не больше 5 строк из таблицы `my_table`. Если в `my_table` есть столбец `my_column`, получить его значение из строки `$i` результата можно следующим образом:

```
$foo = $rv->{rows}[$i]->{my_column};
```

Общее число строк, возвращённых запросом `SELECT`, можно получить так:

```
$nrows = $rv->{processed}
```

Так можно выполнить команду другого типа:

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

Затем можно получить статус команды (например, `SPI_OK_INSERT`) следующим образом:

```
$res = $rv->{status};
```

Чтобы получить число затронутых строк, выполните:

```
$nrows = $rv->{processed};
```

Полный пример:

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

`spi_query(`*команда*`)`
`spi_fetchrow(`*cursor*`)`
`spi_cursor_close(`*cursor*`)`

Функции `spi_query` и `spi_fetchrow` применяются в паре, когда набор строк может быть очень большим или когда нужно возвращать строки по мере их поступления. Функция `spi_fetchrow` работает *только* с `spi_query`. Следующий пример показывает, как использовать их вместе:

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # здесь мы обращаемся к файлу!
        or elog(ERROR, "cannot open $file for reading: $!");
    my @words = <$fh>;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
            the_num => $row->{a},
            the_text => md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

Обычно вызов `spi_fetchrow` нужно повторять, пока не будет получен результат `undef`, показывающий, что все строки уже прочитаны. Курсор, возвращаемый функцией `spi_query`, автоматически освобождается, когда `spi_fetchrow` возвращает `undef`. Если вы не хотите читать все строки, освободите курсор, выполнив `spi_cursor_close`, чтобы не допустить утечки памяти.

`spi_prepare(`*`команда`*`, `*`типы аргументов`*`)`
`spi_query_prepared(`*`план`*`, `*`аргументы`*`)`
`spi_exec_prepared(`*`план`*` [, `*`атрибуты`*`], `*`аргументы`*`)`
`spi_freeplan(`*`план`*`)`

Функции `spi_prepare`, `spi_query_prepared`, `spi_exec_prepared` и `spi_freeplan` реализуют ту же функциональность, но для подготовленных запросов. Функция `spi_prepare` принимает строку запроса с нумерованными местозаполнителями аргументов ($1, $2 и т. д.) и список строк с типами аргументов:

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                                                    'INTEGER', 'TEXT');
```

План запроса, подготовленный вызовом `spi_prepare`, можно использовать вместо строки запроса либо в `spi_exec_prepared`, возвращающей тот же результат, что и `spi_exec_query`, либо в `spi_query_prepared`, возвращающей курсор так же, как `spi_query`, который затем можно передать в `spi_fetchrow`. В необязательном втором параметре `spi_exec_prepared` можно передать хеш с атрибутами; в настоящее время поддерживается только атрибут `limit`, задающий максимальное число строк, которое может вернуть запрос. Без указания `limit` или при указании его равным нулю ограничение на количество строк снимается.

Подготовленные запросы хороши тем, что позволяют использовать единожды подготовленный план для неоднократного выполнения запроса. Когда план оказывается не нужен, его можно освободить, вызвав `spi_freeplan`:

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                        'INTERVAL');
```

```
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

Заметьте, что параметры для `spi_prepare` обозначаются как $1, $2, $3 и т. д., так что по возможности не записывайте строки запросов в двойных кавычках, чтобы не спровоцировать трудноуловимые ошибки.

Ещё один пример, иллюстрирующий использование необязательного параметра `spi_exec_prepared`:

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                      FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                      WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

`spi_commit()`
`spi_rollback()`

Эти функции фиксируют или откатывают текущую транзакцию. Они могут вызываться только в процедурах или в анонимных блоках кода (в команде `DO`), вызываемых из кода верхнего уровня. (Заметьте, что выполнить SQL-команды `COMMIT` или `ROLLBACK` через `spi_exec_query` или подобную функцию нельзя. Соответствующие операции могут выполняться только данными функциями.) После завершения одной транзакции следующая начинается автоматически, отдельной функции для этого нет.

Пример:

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plperl
AS $$
foreach my $i (0..9) {
    spi_exec_query("INSERT INTO test1 (a) VALUES ($i)");
    if ($i % 2 == 0) {
        spi_commit();
    } else {
        spi_rollback();
    }
}
$$;

CALL transaction_test1();
```

43.3.2. Вспомогательные функции в PL/Perl

`elog(`*уровень*`,` *сообщение*`)`

Выдаёт служебное сообщение или сообщение об ошибке. Возможные уровни сообщений: `DEBUG` (ОТЛАДКА), `LOG` (СООБЩЕНИЕ), `INFO` (ИНФОРМАЦИЯ), `NOTICE` (ЗАМЕЧАНИЕ), `WARNING` (ПРЕДУПРЕЖДЕНИЕ) и `ERROR` (ОШИБКА). С уровнем `ERROR` выдаётся ошибка; если она не перехватывается окружающим кодом Perl, она распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. По сути то же самое делает команда `die` языка Perl. При использовании других уровней происходит просто вывод сообщения с заданным уровнем важности. Будут ли сообщения определённого уровня передаваться клиенту и/или записываться в журнал, определяется конфигурационными параметрами log_min_messages и client_min_messages. За дополнительными сведениями обратитесь к Главе 18.

`quote_literal(`*строка*`)`

Оформляет переданную строку для использования в качестве текстовой строки в SQL-операторе. Включённые в неё апострофы и обратная косая черта при этом дублируются. Заметьте, что `quote_literal` возвращает undef, когда получает аргумент undef; если такие аргументы возможны, часто лучше использовать `quote_nullable`.

`quote_nullable(`*строка*`)`

Оформляет переданную строку для использования в качестве текстовой строки в SQL-операторе; либо, если поступает аргумент undef, возвращает строку «NULL» (без кавычек). Символы апостроф и обратная косая черта дублируются должным образом.

`quote_ident(`*строка*`)`

Оформляет переданную строку для использования в качестве идентификатора в SQL-операторе. При необходимости идентификатор заключается в кавычки (например, если он содержит символы, недопустимые в открытом виде, или буквы в разном регистре). Если переданная строка содержит кавычки, они дублируются.

`decode_bytea(`*строка*`)`

Возвращает неформатированные двоичные данные, представленные содержимым заданной строки, которая должна быть закодирована как `bytea`.

`encode_bytea(`*строка*`)`

Возвращает закодированные в виде `bytea` двоичные данные, содержащиеся в переданной строке.

`encode_array_literal(`*массив*`)`
`encode_array_literal(`*массив*`, `*разделитель*`)`

Возвращает содержимое указанного массива в виде строки в формате массива (см. Подраздел 8.15.2). Возвращает значение аргумента неизменённым, если это не ссылка не массив. Разделитель элементов в строке массива по умолчанию — «, » (если разделитель не определён или undef).

`encode_typed_literal(`*значение*`, `*имя_типа*`)`

Преобразует переменную Perl в значение типа данных, указанного во втором аргументе, и возвращает строковое представление этого значения. Корректно обрабатывает вложенные массивы и значения составных типов.

`encode_array_constructor(`*массив*`)`

Возвращает содержимое переданного массива в виде строки в формате конструктора массива (см. Подраздел 4.2.12). Отдельные значения заключаются в кавычки функцией `quote_nullable`. Возвращает значение аргумента, заключённое в кавычки функцией `quote_nullable`, если аргумент — не ссылка на массив.

`looks_like_number(`*строка*`)`

Возвращает значение true, если содержимое переданной строки похоже на число, по правилам Perl, и false в обратном случае. Возвращает undef для аргумента undef. Ведущие и замыкающие пробелы игнорируются. Строки `Inf` и `Infinity` считаются представляющими число (бесконечность).

`is_array_ref(`*аргумент*`)`

Возвращает значение true, если переданный аргумент можно воспринять как ссылку на массив, то есть это ссылка на `ARRAY` или `PostgreSQL::InServer::ARRAY`. В противном случае возвращает false.

43.4. Глобальные значения в PL/Perl

Вы можете использовать для хранения данных, включая ссылки на код, глобальный хеш `%_SHARED`. Эти данные будут сохраняться между вызовами функции на протяжении всего текущего сеанса.

Простой пример работы с разделяемыми данными:

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "cannot set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;
```

```
SELECT set_var('sample', 'Hello, PL/Perl!  How''s tricks?');
SELECT get_var('sample');
```

Это чуть более сложный пример, в котором используется ссылка на код:

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* инициализация функции */

/* Определение функции, использующей функцию заключения в кавычки */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

(Код выше можно было бы упростить до однострочной команды `return $_SHARED{myquote}->($_[0]);` в ущерб читаемости.)

По соображениям безопасности, PL/Perl выполняет функции, вызываемые некоторой ролью SQL, в отдельном интерпретаторе Perl, выделенном для этой роли. Это предотвращает случайное или злонамеренное влияние одного пользователя на поведение функций PL/Perl другого пользователя. В каждом интерпретаторе будет своё значение переменной `%_SHARED` и собственное глобальное состояние. Таким образом, две функции PL/Perl будут разделять одно значение `%_SHARED`, только если они выполняются одной ролью SQL. В приложении, выполняющем код в одном сеансе с разными ролями SQL (вызывающем функции `SECURITY DEFINER`, использующем команду `SET ROLE` и т. д.) может понадобиться явно предпринять дополнительные меры, чтобы функции на PL/Perl могли разделять данные через `%_SHARED`. Для этого сначала установите для функций, которые должны взаимодействовать, одного владельца, а затем задайте для них свойство `SECURITY DEFINER`. Разумеется, при этом нужно позаботиться о том, чтобы эти функции не могли сделать ничего непредусмотренного.

43.5. Доверенный и недоверенный PL/Perl

Обычно PL/Perl устанавливается в базу данных как «доверенный» язык программирования с именем `plperl`. При этом в целях безопасности определённые операции в Perl запрещаются. Вообще говоря, запрещаются все операции, взаимодействующие с окружением. В том числе, это операции с файлами, `require` и `use` (для внешних модулей). Поэтому функции на PL/Perl, в отличие от функций на C, никаким образом не могут взаимодействовать с внутренними механизмами сервера баз данных или обращаться к операционной системе с правами серверного процесса. Вследствие этого, использовать этот язык можно разрешить любому непривилегированному пользователю баз данных.

Предупреждение

Доверенный PL/Perl использует модуль Perl `Opcode` в целях безопасности. Однако в *документации* Perl указано, что этот модуль не эффективен в работе с доверенным PL/Perl. Если требуется повышенная безопасность, будет целесообразно выполнить команду `REVOKE USAGE ON LANGUAGE plperl FROM PUBLIC`.

В следующем примере показана функция, которая не будет работать, потому что операции с файловой системы запрещены по соображениям безопасности:

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

Создать эту функцию не удастся, так как при проверке её правильности будет обнаружено использование запрещённого оператора.

Иногда возникает желание написать на Perl код, функциональность которого не будет ограничиваться. Например, может потребоваться функция на Perl, которая будет посылать почту. Для таких потребностей PL/Perl также можно установить как «недоверенный» язык (обычно его называют PL/PerlU). В этом случае будут доступны все возможности языка Perl. Устанавливая язык, укажите имя `plperlu`, чтобы выбрать недоверенную вариацию PL/Perl.

Автор функции на PL/PerlU должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных. Заметьте, что СУБД позволяет создавать функции на недоверенных языках только суперпользователям базы данных.

Если показанная выше функция будет создана суперпользователем, и при этом будет выбран язык `plperlu`, она выполнится успешно.

Таким же образом, в анонимном блоке кода на Perl разрешены абсолютно любые операции, если в качестве языка вместо `plperl` выбирается `plperlu`, но выполнять этот код должен суперпользователь.

Примечание

Тогда как функции на PL/Perl исполняются отдельными интерпретаторами Perl для каждой роли SQL, все функции на PL/PerlU, вызываемые в рамках сеанса, исполняются в одном интерпретаторе Perl (отличном от тех, что исполняют функции PL/Perl). Благодаря этому, функции PL/PerlU могут свободно разделять общие данные, но между функциями PL/Perl и PL/PerlU взаимодействие невозможно.

Примечание

Perl поддерживает работу нескольких интерпретаторов в одном процессе, только если он был собран с нужными флагами, а именно, с флагом `usemultiplicity` или с флагом `useithreads`. (В отсутствие веских причин использовать потоки предпочтительным является вариант `usemultiplicity`. Дополнительную информацию вы можете получить на странице man perlembed.) При использовании PL/Perl с версией Perl, собранной без этих флагов, в рамках сеанса можно будет запустить только один интерпретатор Perl, так что в сеансе будет возможно выполнять либо функции PL/PerlU, либо функции PL/Perl (и вызывать их должна одна роль SQL).

43.6. Триггеры на PL/Perl

PL/Perl можно использовать для написания триггерных функций. В триггерной функции хеш-массив `$_TD` содержит информацию о произошедшем событии триггера. `$_TD` — глобальная переменная, которая получает нужное локальное значение при каждом вызове триггера. Хеш-массив `$_TD` содержит следующие поля:

`$_TD->{new}{foo}`

Новое значение столбца `foo`

`$_TD->{old}{foo}`

Старое значение столбца `foo`

`$_TD->{name}`

Имя вызываемого триггера

`$_TD->{event}`

Событие триггера: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` или `UNKNOWN`

`$_TD->{when}`

Когда вызывается триггер: `BEFORE` (ДО), `AFTER` (ПОСЛЕ), `INSTEAD OF` (ВМЕСТО) или `UNKNOWN` (НЕИЗВЕСТНО)

`$_TD->{level}`

Уровень триггера: `ROW` (СТРОКА), `STATEMENT` (ОПЕРАТОР) или `UNKNOWN` (НЕИЗВЕСТНЫЙ)

`$_TD->{relid}`

OID таблицы, для которой сработал триггер

`$_TD->{table_name}`

Имя таблицы, для которой сработал триггер

`$_TD->{relname}`

Имя таблицы, для которой сработал триггер. Это обращение устарело и может быть ликвидировано в будущем выпуске. Используйте вместо него $_TD->{table_name}.

`$_TD->{table_schema}`

Имя схемы, содержащей таблицу, для которой сработал триггер

`$_TD->{argc}`

Число аргументов в триггерной функции

`@{$_TD->{args}}`

Аргументы триггерной функции. Не определено, если `$_TD->{argc}` равно 0.

В триггерах уровня строки возможны следующие варианты возврата:

`return;`

Выполнить операцию

`"SKIP"`

Не выполнять операцию

`"MODIFY"`

Указывает, что строка `NEW` была изменена триггерной функцией

Следующий пример триггерной функции иллюстрирует описанные выше варианты:

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    # пропустить команду INSERT/UPDATE
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  # изменить строку и выполнить команду INSERT/UPDATE
    } else {
        return;           # выполнить команду INSERT/UPDATE
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE FUNCTION valid_id();
```

43.7. Событийные триггеры на PL/Perl

PL/Perl можно использовать для написания функций событийных триггеров. В функции событийного триггера хеш-массив `$_TD` содержит информацию о произошедшем событии триггера. `$_TD` — глобальная переменная, которая получает нужное локальное значение при каждом вызове триггера. Хеш-массив `$_TD` содержит следующие поля:

`$_TD->{event}`

Имя события, при котором срабатывает этот триггер.

`$_TD->{tag}`

Тег команды, для которой срабатывает этот триггер.

Возвращаемое значение триггерной функции игнорируется.

Следующий пример функции событийного триггера иллюстрирует описанное выше:

```
CREATE OR REPLACE FUNCTION perlsnitch() RETURNS event_trigger AS $$
  elog(NOTICE, "perlsnitch: " . $_TD->{event} . " " . $_TD->{tag} . " ");
$$ LANGUAGE plperl;

CREATE EVENT TRIGGER perl_a_snitch
    ON ddl_command_start
    EXECUTE FUNCTION perlsnitch();
```

43.8. Внутренние особенности PL/Perl

43.8.1. Конфигурирование

В этом разделе описываются параметры конфигурации, влияющие на работу PL/Perl.

`plperl.on_init` (`string`)

Задаёт код Perl, который будет выполняться при первой инициализации интерпретатора Perl, до того, как он получает специализацию `plperl` или `plperlu`. Когда этот код выполняется, функции SPI ещё не доступны. Если выполнение кода завершается ошибкой, инициализация интерпретатора прерывается и ошибка распространяется в вызывающий запрос, в результате чего текущая транзакция или подтранзакция прерывается.

Размер этого кода ограничивается одной строкой. Более объёмный код можно поместить в модуль и загрузить этот модуль в строке `on_init`. Например:

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

Любые модули, загруженные в `plperl.on_init`, явно или неявно, будут доступны для использования в коде на языке `plperl`. Это может создать угрозу безопасности. Чтобы определить, какие модули были загружены, можно выполнить:

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

Если библиотека `plperl` включена в shared_preload_libraries, инициализация произойдёт в главном процессе (postmaster) и в этом случае необходимо очень серьёзно оценить риск нарушения работоспособности этого процесса. Основной смысл использовать эту возможность в том, чтобы модули Perl, подключаемые в `plperl.on_init`, загружались только при запуске главного процесса, и это исключало бы издержки загрузки для отдельных сеансов. Однако имейте в виду, что эти издержки исключаются только при загрузке в сеансе первого интерпретатора Perl — будь то PL/PerlU или PL/Perl для первой SQL-роли, вызывающей функцию на PL/Perl. Любые дополнительные интерпретаторы Perl, создаваемые в сеансе базы данных, должны будут выполнять `plperl.on_init` заново. Также учтите, что в Windows предварительная загрузка не даёт никакого выигрыша, так как интерпретатор Perl, созданный в главном процессе, не передаётся дочерним процессам.

Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

`plperl.on_plperl_init` (`string`)
`plperl.on_plperlu_init` (`string`)

В этих параметрах задаётся код Perl, который будет выполняться в момент, когда интерпретатор Perl получает специализацию `plperl` или `plperlu`, соответственно. Это произойдёт, когда в рамках сеанса будет первый раз вызвана функция на PL/Perl или PL/PerlU, либо когда потребуется дополнительный интерпретатор при использовании другого языка или при вызове функции PL/Perl новой SQL-ролью. Этот код выполняется после инициализации, произведённой в `plperl.on_init`. Однако функции SPI в момент исполнения этого кода ещё не доступны. Код в `plperl.on_plperl_init` запускается после того, как интерпретатор «помещается под замок», так что в нём разрешаются только доверенные операции.

Если этот код завершается ошибкой, инициализация прерывается и ошибка распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. При этом любые действия, уже произведённые в Perl, не будут отменены; однако использоваться этот интерпретатор больше не будет. При следующей попытке использовать этот язык система попытается заново инициализировать свежий интерпретатор Perl.

Изменять эти параметры разрешено только суперпользователям. Хотя изменить их можно в рамках сеанса, такие изменения не повлияют на работу интерпретаторов Perl, задействованных для выполнения функций ранее.

`plperl.use_strict` (`boolean`)

При значении, равном true, последующая компиляция функций PL/Perl будет выполняться с включённым указанием `strict`. Этот параметр не влияет на функции, уже скомпилированные в текущем сеансе.

43.8.2. Ограничения и недостающие возможности

Следующие возможности в настоящее время в PL/Perl отсутствуют, но их реализация будет желанной доработкой.

- Функции на PL/Perl не могут напрямую вызывать друг друга.

- SPI ещё не полностью реализован.
- Если вы выбираете очень большие наборы данных, используя `spi_exec_query`, вы должны понимать, что все эти данные загружаются в память. Вы можете избежать этого, используя пару функций `spi_query`/`spi_fetchrow`, как показано ранее.

  Похожая проблема возникает, если функция, возвращающая множество, передаёт в Postgres Pro большое число строк, выполняя `return`. Этой проблемы так же можно избежать, выполняя для каждой возвращаемой строки `return_next`, как показано ранее.
- Когда сеанс завершается штатно, не по причине критической ошибки, в Perl выполняются все блоки `END`, которые были определены. Никакие другие действия в настоящее время не выполняются. В частности, буферы файлов автоматически не сбрасываются и объекты автоматически не уничтожаются.

Глава 44. PL/Python — процедурный язык Python

Процедурный язык PL/Python позволяет писать функции и процедуры Postgres Pro на *языке Python*.

Чтобы установить PL/Python в определённую базу данных, выполните команду `CREATE EXTENSION plpython3u`.

Подсказка

Если язык устанавливается в `template1`, он будет автоматически установлен во все создаваемые впоследствии базы данных.

PL/Python представлен только в виде «недоверенного» языка, что означает, что он никаким способом не ограничивает действия пользователей, и поэтому он называется `plpython3u`. Доверенная вариация `plpython` может появиться в будущем, если в Python будет разработан безопасный механизм выполнения. Автор функции на недоверенном языке PL/Python должен позаботиться о том, чтобы эту функцию нельзя было использовать не по назначению, так как она может делать всё, что может пользователь с правами администратора баз данных. Создавать функции на недоверенных языках, таких как `plpython3u`, разрешено только суперпользователям.

Примечание

Пользователи, имеющие дело с исходным кодом, должны явно включить сборку PL/Python в процессе установки. (За дополнительными сведениями обратитесь к инструкциям по установке.) Пользователи двоичных пакетов могут найти PL/Python в отдельном модуле.

44.1. Функции на PL/Python

Функции на PL/Python объявляются стандартным образом с помощью команды CREATE FUNCTION:

```
CREATE FUNCTION имя_функции (аргументы)
  RETURNS возвращаемый_тип
AS $$
  # Тело функции на PL/Python
$$ LANGUAGE plpython3u;
```

Тело функции содержит просто скрипт на языке Python. Когда вызывается функция, её аргументы передаются в виде элементов списка `args`; именованные аргументы также передаются скрипту Python как обычные переменные. С именованными аргументами скрипт обычно лучше читается. Результат из кода Python возвращается обычным способом: командой `return` или `yield` (в случае функции, возвращающей множество). Если возвращаемое значение не определено, Python возвращает `None`. Исполнитель PL/Python преобразует `None` языка Python в значение NULL языка SQL. В процедуре код Python должен возвращать `None` (обычно для этого процедура завершается без оператора `return` или используется оператор `return` без аргумента); в противном случае выдаётся ошибка.

Например, функцию, возвращающее большее из двух целых чисел, можно определить так:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpython3u;
```

Код на Python, заданный в качестве тела объявляемой функции, становится телом функции Python. Например, для показанного выше объявления получается функция:

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

Здесь 23456 — это OID, который Postgres Pro присвоил данной функции.

Значения аргументов задаются в глобальных переменных. Согласно правилам видимости в Python, тонким следствием этого является то, что переменной аргумента нельзя присвоить внутри функции выражение, включающее имя самой этой переменной, если только эта переменная не объявлена глобальной в текущем блоке. Например, следующий код не будет работать:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # ошибка
  return x
$$ LANGUAGE plpython3u;
```

так как присваивание `x` значения делает `x` локальной переменной для всего блока, и при этом `x` в правой части присваивания оказывается ещё не определённой локальной переменной `x`, а не параметром функции PL/Python. Добавив оператор `global`, это можно исправить:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # теперь всё в порядке
  return x
$$ LANGUAGE plpython3u;
```

Однако рекомендуется не полагаться на такие особенности реализации PL/Python, а принять, что параметры функции предназначены только для чтения.

44.2. Значения данных

Вообще говоря, цель исполнителя PL/Python — обеспечить «естественное» соответствие между мирами PostgreSQL и Python. Этим объясняется выбор правил сопоставления данных, описанных ниже.

44.2.1. Сопоставление типов данных

Когда вызывается функция PL/Python, её аргументы преобразуются из типа PostgreSQL в соответствующий тип Python по таким правилам:

- Тип PostgreSQL `boolean` преобразуется в тип `bool` языка Python.
- Типы PostgreSQL `smallint`, `int`, `bigint` и `oid` преобразуются в тип `int` языка Python.
- Типы PostgreSQL `real` и `double` преобразуются в тип `float` языка Python.
- Тип PostgreSQL `numeric` преобразуется в тип `Decimal` среды Python. Этот тип импортируется из пакета `cdecimal`, при его наличии. В противном случае используется `decimal.Decimal` из стандартной библиотеки. Тип `cdecimal` работает значительно быстрее, чем `decimal`. Однако в Python версии 3.3 и выше тип `cdecimal` включается в стандартную библиотеку под именем `decimal`, так что теперь этого различия нет.
- Тип PostgreSQL `bytea` преобразуется в тип `bytes` языка Python.
- Все другие типы данных, включая типы символьных строк PostgreSQL, преобразуются в тип `str` языка Python (строка в Unicode, как и все строки Python).
- Информация о нескалярных типах данных приведена ниже.

При завершении функции PL/Python её значение результата преобразуется в тип данных, объявленный как тип результата в PostgreSQL, следующим образом:

- Когда тип результата функции в PostgreSQL — `boolean`, возвращаемое значение приводится к логическому типу по правилам, принятым в *Python*. То есть false будет возвращено для 0 и пустой строки, но, обратите внимание, для `'f'` будет возвращено true.
- Когда тип результата функции PostgreSQL — `bytea`, возвращаемое значение будет преобразовано в набор байт, используя встроенные средства Python, а затем будет приведено к типу `bytea`.
- Для всех других типов результата PostgreSQL возвращаемое значение преобразуется в строку с помощью встроенной в Python функции `str`, и полученная строка передаётся функции ввода типа данных PostgreSQL. (Если значение в Python имеет тип `float`, оно преобразуется встроенной функцией `repr`, а не `str`, для недопущения потери точности.)

  При передаче в PostgreSQL строки автоматически преобразуются в кодировку сервера PostgreSQL.
- Информация о нескалярных типах данных приведена ниже.

Заметьте, что логические несоответствия между объявленным в PostgreSQL типом результата и типом фактически возвращаемого объекта Python игнорируются — значение преобразуется в любом случае.

44.2.2. Null, None

Если функции передаётся значение SQL NULL, в Python значением этого аргумента будет `None`. Например, функция `pymax`, определённая как показано в Раздел 44.1, возвратит неверный ответ, получив аргументы NULL. Мы могли бы добавить указание `STRICT` в определение функции, чтобы Postgres Pro поступал немного разумнее: при передаче значения NULL функция вовсе не будет вызываться, будет сразу возвращён результат NULL. С другой стороны, мы могли бы проверить аргументы на NULL в теле функции:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpython3u;
```

Как показано выше, чтобы выдать из функции PL/Python значение SQL NULL, нужно вернуть значение `None`. Это можно сделать и в строгой, и в нестрогой функции.

44.2.3. Массивы, списки

Значения массивов SQL передаются в PL/Python в виде списка Python. Чтобы вернуть значение массива SQL из функции PL/Python, возвратите список Python:

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return [1, 2, 3, 4, 5]
$$ LANGUAGE plpython3u;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

Многомерные массивы передаются в PL/Python в виде вложенных списков Python. Например, двухмерный массив представляется как список списков. При передаче многомерного массива SQL из функции PL/Python необходимо, чтобы все внутренние списки на каждом уровне имели одинаковый размер. Например:

```
CREATE FUNCTION test_type_conversion_array_int4(x int4[]) RETURNS int4[] AS $$
plpy.info(x, type(x))
return x
$$ LANGUAGE plpython3u;

SELECT * FROM test_type_conversion_array_int4(ARRAY[[1,2,3],[4,5,6]]);
INFO:  ([[1, 2, 3], [4, 5, 6]], <type 'list'>)
 test_type_conversion_array_int4
---------------------------------
 {{1,2,3},{4,5,6}}
(1 row)
```

Другие последовательности Python, например кортежи, тоже принимаются для обратной совместимости с PostgreSQL версии 9.6 и ниже (где многомерные массивы не поддерживались). Однако они всегда воспринимаются как одномерные массивы, чтобы не возникало неоднозначности с составными типами. По этой же причине когда в многомерном массиве используется составной тип, он должен представляться как кортеж, а не список.

Учтите, что в Python и строки являются последовательностями, что может давать неожиданные эффекты, хорошо знакомые тем, кто программирует на Python:

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpython3u;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

44.2.4. Составные типы

Аргументы составного типа передаются функции в виде сопоставлений Python. Именами элементов сопоставления являются атрибуты составного типа. Если атрибут в переданной строке имеет значение NULL, он передаётся в сопоставлении значением `None`. Пример работы с составным типом:

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpython3u;
```

Возвратить составной тип или строку таблицы из функции Python можно несколькими способами. В следующих примерах предполагается, что у нас объявлен тип:

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

Результат этого типа можно вернуть как:

Последовательность (кортеж или список, но не множество, так как оно не индексируется)

В возвращаемых объектах последовательностей должно быть столько элементов, сколько полей в составном типе результата. Элемент с индексом 0 присваивается первому полю составного типа, с индексом 1 — второму и т. д. Например:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return ( name, value )
  # или альтернативный вариант в виде списка: return [ name, value ]
$$ LANGUAGE plpython3u;
```

Чтобы выдать SQL NULL для какого-нибудь столбца, вставьте в соответствующую позицию `None`.

Когда возвращается массив составных значений, его нельзя представить в виде списка, так как невозможно однозначно определить, представляет ли список Python составной тип или ещё одну размерность массива.

Сопоставление (словарь)

Значение столбца результата получается из сопоставления, в котором ключом является имя столбца. Например:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpython3u;
```

Любые дополнительные пары ключ/значение в словаре игнорируются, а отсутствие нужных ключей считается ошибкой. Чтобы выдать SQL NULL для какого-нибудь столбца, вставьте `None` с именем соответствующего столбца в качестве ключа.

Объект (любой объект с методом `__getattr__`)

Объект передаётся аналогично сопоставлению. Пример:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # or simply
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpython3u;
```

Также поддерживаются функции с параметрами `OUT` (выходными). Например:

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpython3u;

SELECT * FROM multiout_simple();
```

Выходные параметры процедуры выдаются таким же образом. Например:

```
CREATE PROCEDURE python_triple(INOUT a integer, INOUT b integer) AS $$
return (a * 3, b * 3)
$$ LANGUAGE plpython3u;

CALL python_triple(5, 10);
```

44.2.5. Функции, возвращающие множества

Функция PL/Python также может возвращать множества, содержащие скалярные и составные типы. Это можно осуществить разными способами, так как возвращаемый объект внутри превращается в итератор. В следующих примерах предполагается, что у нас есть составной тип:

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

Множество в качестве результата можно возвратить, применив:

Последовательность (кортеж, список, множество)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # возвращает кортеж, содержащий списки в качестве составных типов
  # также будут работать и остальные комбинации
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpython3u;
```

Итератор (любой объект, реализующий методы `__iter__` и `__next__`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def __next__(self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpython3u;
```

Генератор (`yield`)

```
CREATE FUNCTION greet (how text)
```

```
    RETURNS SETOF greeting
  AS $$
    for who in [ "World", "PostgreSQL", "PL/Python" ]:
      yield ( how, who )
  $$ LANGUAGE plpython3u;
```

Также поддерживаются функции, возвращающие множества, с параметрами `OUT` (объявленные с `RETURNS SETOF record`). Например:

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer) RETURNS
 SETOF record AS $$
return [(1, 2)] * n
$$ LANGUAGE plpython3u;

SELECT * FROM multiout_simple_setof(3);
```

44.3. Совместное использование данных

Для сохранения внутренних данных при повторных вызовах одной и той же функции предусмотрен глобальный словарь `SD`. Для размещения публичных данных предназначен глобальный словарь `GD`, доступный всем функциям на Python в сеансе; используйте его с осторожностью.

Каждая функция получает собственную среду выполнения в интерпретаторе Python, так что глобальные данные и аргументы функции, например `myfunc`, не будут доступны в `myfunc2`. Исключение составляют данные в словаре `GD`, как сказано выше.

44.4. Анонимные блоки кода

PL/Python также поддерживает анонимные блоки кода, которые выполняются оператором DO:

```
DO $$
    # Код на PL/Python
$$ LANGUAGE plpython3u;
```

Анонимный блок кода не принимает аргументы, а любое значение, которое он мог бы вернуть, отбрасывается. В остальном он работает подобно коду функции.

44.5. Триггерные функции

Когда функция используется как триггер, словарь `TD` содержит значения, связанные с работой триггера:

`TD["event"]`

содержит название события в виде строки: `INSERT`, `UPDATE`, `DELETE` или `TRUNCATE`.

`TD["when"]`

содержит одну из строк: `BEFORE`, `AFTER` или `INSTEAD OF`.

`TD["level"]`

содержит `ROW` или `STATEMENT`.

`TD["new"]`
`TD["old"]`

Для триггера уровня строки одно или оба этих поля содержат соответствующие строки триггера, в зависимости от события триггера.

`TD["name"]`

содержит имя триггера.

`TD["table_name"]`

содержит имя таблицы, для которой сработал триггер.

`TD["table_schema"]`

содержит схему таблицы, для которой сработал триггер.

`TD["relid"]`

содержит OID таблицы, для которой сработал триггер.

`TD["args"]`

Если в команде `CREATE TRIGGER` задавались аргументы, их можно получить как элементы массива с `TD["args"][0]` по `TD["args"][`*n*`-1]`.

Если в `TD["when"]` передано `BEFORE` или `INSTEAD OF`, а в `TD["level"]` — `ROW`, вы можете вернуть значение `None` или `"OK"` из функции Python, чтобы показать, что строка не была изменена, значение `"SKIP"`, чтобы прервать событие, либо, если в `TD["event"]` передана команда `INSERT` или `UPDATE`, вы можете вернуть `"MODIFY"`, чтобы показать, что новая строка была изменена. Во всех других случаях возвращаемое значение игнорируется.

44.6. Обращение к базе данных

Исполнитель языка PL/Python автоматически импортирует модуль Python с именем `plpy`. Вы в своём коде можете использовать функции и константы, объявленные в этом модуле, обращаясь к ним по именам вида `plpy.`*имя*.

44.6.1. Функции обращения к базе данных

Модуль `plpy` содержит различные функции для выполнения команд в базе данных:

`plpy.execute(`*query* `[, ` *предел*`])`

При вызове `plpy.execute` со строкой запроса и необязательным аргументом, ограничивающим число строк, выполняется заданный запрос, а то, что он выдаёт, возвращается в виде объекта результата.

Если *предел* задан и больше нуля, то `plpy.execute` получает число строк, не превышающее *предел*, как если бы запрос включал предложение `LIMIT`. Без указания *предела* или когда он равен нулю ограничение на количество строк снимается.

Объект результата имитирует список или словарь. Получить из него данные можно по номеру строки и имени столбца. Например, команда:

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

вернёт не более 5 строк из отношения `my_table`. Если в `my_table` есть столбец `my_column`, к нему можно обратиться так:

```
foo = rv[i]["my_column"]
```

Число возвращённых в этом объекте строк можно получить, воспользовавшись встроенной функцией `len`.

Для объекта результата определены следующие дополнительные методы:

`nrows()`

Возвращает число строк, обработанных командой. Заметьте, что это число не обязательно будет равно числу возвращённых строк. Например, команда `UPDATE` устанавливает это значение, но не возвращает строк (без указания `RETURNING`).

`status()`

Значение состояния, возвращённое `SPI_execute()`.

`colnames()`
`coltypes()`
`coltypmods()`

Возвращают список имён столбцов, список OID типов столбцов и список модификаторов типа этих столбцов, соответственно.

Эти методы вызывают исключение, когда им передаётся объект, полученный от команды, не возвращающей результирующий набор, например, `UPDATE` без `RETURNING`, либо `DROP TABLE`. Но эти методы вполне можно использовать с результатом, содержащим ноль строк.

`__str__()`

Стандартный метод `__str__` определён так, чтобы можно было, например, вывести отладочное сообщение с результатами запроса, вызвав `plpy.debug(rv)`.

Объект результата может быть изменён.

Заметьте, что при вызове `plpy.execute` весь набор результатов будет прочитан в память. Эту функцию следует использовать, только если вы знаете, что набор будет относительно небольшим. Если вы хотите исключить риск переполнения памяти при выборке результатов большого объёма, используйте `plpy.cursor` вместо `plpy.execute`.

`plpy.prepare(`*`query`*` [, `*`типы_аргументов`*`])`
`plpy.execute(`*`план`*` [, `*`аргументы`*` [, `*`предел`*`]])`

Функция `plpy.prepare` подготавливает план выполнения для запроса. Она вызывается со строкой запроса и списком типов параметров (если в запросе есть параметры). Например:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1",
 ["text"])
```

Здесь `text` представляет переменную, передаваемую в качестве параметра `$1`. Второй аргумент необязателен, если запросу не нужно передавать никакие параметры.

Чтобы запустить подготовленный оператор на выполнение, используйте вариацию функции `plpy.execute`:

```
rv = plpy.execute(plan, ["name"], 5)
```

Передайте план в первом аргументе (вместо строки запроса), а список значений, которые будут подставлены в запрос, — во втором. Второй аргумент можно опустить, если запрос не принимает никакие параметры. Третий аргумент, как и раньше, задаёт необязательное ограничение максимального числа строк.

Вы также можете вызвать метод `execute` объекта плана:

```
rv = plan.execute(["name"], 5)
```

Параметры запросов и поля строк результата преобразуются между типами данных PostgreSQL и Python как описано в Разделе 44.2.

Когда вы подготавливаете план, используя модуль PL/Python, он сохраняется автоматически. Что это означает, вы можете узнать в документации SPI (Глава 45). Чтобы эффективно использовать это в нескольких вызовах функции, может потребоваться применить словарь постоянного хранения `SD` или `GD` (см. Раздел 44.3). Например:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # остальной код функции
$$ LANGUAGE plpython3u;
```

`plpy.cursor(`*`query`*`)`
`plpy.cursor(`*`план`*` [, `*`аргументы`*`])`

Функция `plpy.cursor` принимает те же аргументы, что и `plpy.execute` (кроме ограничения строк) и возвращает объект курсора, который позволяет обрабатывать объёмные наборы результатов небольшими порциями. Как и `plpy.execute`, этой функции можно передать строку запроса или объект плана со списком аргументов, а можно вызывать функцию `cursor` как метод объекта плана.

Объект курсора реализует метод `fetch`, который принимает целочисленный параметр и возвращает объект результата. При каждом следующем вызове `fetch` возвращаемый объект будет содержать следующий набор строк, в количестве, не превышающем значение параметра. Когда строки закончатся, `fetch` начнёт возвращать пустой объект результата. Объекты курсора также предоставляют *интерфейс итератора*, выдающий по строке за один раз, пока не будут выданы все строки. Данные, выбираемые таким образом, возвращаются не как объекты результата, а как словари (одной строке результата соответствует один словарь).

Следующий пример демонстрирует обработку содержимого большой таблицы двумя способами:

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpython3u;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpython3u;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integer"])
rows = list(plpy.cursor(plan, [2]))  # или: = list(plan.cursor([2]))

return len(rows)
$$ LANGUAGE plpython3u;
```

Курсоры ликвидируются автоматически. Но если вы хотите явно освободить все ресурсы, занятые курсором, вызовите метод `close`. Продолжать получать данные через курсор, который был закрыт, нельзя.

Подсказка

Не путайте объекты, создаваемые функцией `plpy.cursor`, с курсорами DB-API, определёнными в *спецификации API для работы с базами данных в Python*. Они не имеют ничего общего, кроме имени.

44.6.2. Обработка ошибок

Функции, обращающиеся к базе данных, могут сталкиваться с ошибками, в результате которых они будут прерываться и вызывать исключение. Обе функции `plpy.execute` и `plpy.prepare` могут вызывать экземпляр подкласса исключения `plpy.SPIError`, которое по умолчание прекращает выполнение функции. Эту ошибку можно обработать, как и любое другое исключение в Python, применив конструкцию `try/except`. Например:

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpython3u;
```

Фактический класс вызываемого исключения соответствует определённому условию возникновения ошибки. Список всех возможных условий приведён в Таблице A.1. В модуле `plpy.spiexceptions` определяются классы исключений для каждого условия Postgres Pro, с именами, производными от имён условий. Например, имя `division_by_zero` становится именем `DivisionByZero`, `unique_violation` — именем `UniqueViolation`, `fdw_error` — именем `FdwError` и т. д. Все эти классы исключений наследуются от `SPIError`. Такое разделение на классы упрощает обработку определённых ошибок, например:

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int",
 "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError as e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpython3u;
```

Заметьте, что так как все исключения из модуля `plpy.spiexceptions` наследуются от исключения `SPIError`, команда `except`, обрабатывающая это исключение, будет перехватывать все ошибки при обращении к базе данных.

В качестве другого варианта обработки различных условий ошибок, вы можете перехватывать исключение `SPIError` и определять конкретное условие ошибки внутри блока `except` по значению атрибута `sqlstate` объекта исключения. Этот атрибут содержит строку с кодом ошибки «SQLSTATE». Конечный результат при таком подходе примерно тот же.

44.7. Явные подтранзакции

Перехват ошибок, произошедших при обращении к базе данных, как описано в Подразделе 44.6.2, может привести к нежелательной ситуации, когда часть операций будет успешно выполнена, прежде чем произойдёт сбой. Данные останутся в несогласованном состоянии после обработки такой ошибки. PL/Python предлагает решение этой проблемы в форме явных подтранзакций.

44.7.1. Менеджеры контекста подтранзакций

Рассмотрим функцию, осуществляющую перевод средств между двумя счетами:

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'")
except plpy.SPIError as e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpython3u;
```

Если при выполнении второго оператора `UPDATE` произойдёт исключение, эта функция сообщит об ошибке, но результат первого `UPDATE` будет тем не менее зафиксирован. Другими словами, средства будут списаны со счёта Джо, но не зачислятся на счёт Мэри.

Во избежание таких проблем можно завернуть вызовы `plpy.execute` в явную подтранзакцию. Модуль `plpy` предоставляет вспомогательный объект для управления явными подтранзакциями, создаваемый функцией `plpy.subtransaction()`. Объекты, созданные этой функцией, реализуют *интерфейс менеджера контекста*. Используя явные подтранзакции, мы можем переписать нашу функцию так:

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name =
 'joe'")
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name =
 'mary'")
except plpy.SPIError as e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpython3u;
```

Заметьте, что конструкция `try`/`except` по-прежнему нужна. Без неё исключение распространится вверх по стеку Python и приведёт к прерыванию всей функции с ошибкой Postgres Pro, так что в таблицу `operations` запись не добавится. Менеджер контекста подтранзакции не перехватывает ошибки, он только гарантирует, что все операции с базой данных в его области действия будут атомарно зафиксированы или отменены. Откат блока подтранзакции происходит при исключении любого вида, а не только исключения, вызванного ошибками при обращении к базе данных. Обычное исключение Python, вызванное внутри блока явной подтранзакции, также приведёт к откату этой подтранзакции.

44.8. Управление транзакциями

В процедуре, которая вызывается в коде верхнего уровня или в анонимном блоке кода (в команде `DO`), можно управлять транзакциями. Чтобы зафиксировать текущую транзакцию, вызовите `plpy.commit()`, а чтобы откатить — `plpy.rollback()`. (Заметьте, что выполнить SQL-команды `COMMIT` или `ROLLBACK` через `plpy.execute` или подобную функцию нельзя. Соответствующие операции могут выполняться только данными функциями.) После завершения одной транзакции следующая начинается автоматически, отдельной функции для этого нет.

Пример:

```
CREATE PROCEDURE transaction_test1()
```

```
LANGUAGE plpython3u
AS $$
for i in range(0, 10):
    plpy.execute("INSERT INTO test1 (a) VALUES (%d)" % i)
    if i % 2 == 0:
        plpy.commit()
    else:
        plpy.rollback()
$$;

CALL transaction_test1();
```

Транзакцию нельзя завершить в случае существования открытой явной подтранзакции.

44.9. Вспомогательные функции

Модуль `plpy` также предоставляет функции

```
plpy.debug( msg, **kwargs )
plpy.log( msg, **kwargs )
plpy.info( msg, **kwargs )
plpy.notice( msg, **kwargs )
plpy.warning( msg, **kwargs )
plpy.error( msg, **kwargs )
plpy.fatal( msg, **kwargs )
```

Функции `plpy.error` и `plpy.fatal` на самом деле выдают исключение Python, которое, если его не перехватить, распространяется в вызывающий запрос, что приводит к прерыванию текущей транзакции или подтранзакции. Команды `raise plpy.Error(`*msg*`)` и `raise plpy.Fatal(`*msg*`)` равнозначны вызовам `plpy.error(`*msg*`)` и `plpy.fatal(`*msg*`)`, соответственно, но форма `raise` не позволяет передавать аргументы с ключами. Другие функции просто выдают сообщения разных уровней важности. Будут ли сообщения определённого уровня передаваться клиентам и/или записываться в журнал сервера, определяется конфигурационными переменными log_min_messages и client_min_messages. За дополнительными сведениями обратитесь к Главе 18.

Аргумент *msg* задаётся как позиционный. Для обратной совместимости может быть передано несколько позиционных аргументов. В этом случае сообщением для клиента становится строковое представление кортежа позиционных аргументов.

Дополнительно только по ключам принимаются следующие аргументы:

`detail`
`hint`
`sqlstate`
`schema_name`
`table_name`
`column_name`
`datatype_name`
`constraint_name`

Строковое представление объектов, передаваемых в аргументах по ключам, позволяет выдать клиенту более богатую информацию. Например:

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
           detail="some info about exception",
           hint="hint for users")
$$ LANGUAGE plpython3u;

=# SELECT raise_custom_exception();
```

```
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
  PL/Python function "raise_custom_exception", line 4, in <module>
    hint="hint for users")
PL/Python function "raise_custom_exception"
```

Ещё один набор вспомогательных функций образуют `plpy.quote_literal(`*`строка`*`)`, `plpy.quote_nullable(`*`строка`*`)` и `plpy.quote_ident(`*`строка`*`)`. Они равнозначны встроенным функциям заключения в кавычки, описанным в Разделе 9.4. Они полезны при конструировании свободно составляемых запросов. На PL/Python динамический SQL, показанный в Примере 41.1, формируется так:

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

44.10. Python 2 и Python 3

PL/Python поддерживает только Python 3. В предыдущих версиях Postgres Pro поддерживался Python 2, при этом использовались имена языков `plpythonu` и `plpython2u`.

44.11. Переменные окружения

Некоторые переменные окружения, воспринимаемые интерпретатором Python, тоже могут влиять на поведение PL/Python. При необходимости их нужно установить в среде основного серверного процесса Postgres Pro, например, в скрипте запуска. Множество доступных переменных окружения зависит от версии Python; за подробностями обратитесь к документации Python. На момент написания этой документации, на поведение PL/Python влияли следующие переменные окружения, при наличии подходящей версии Python:

- `PYTHONHOME`
- `PYTHONPATH`
- `PYTHONY2K`
- `PYTHONOPTIMIZE`
- `PYTHONDEBUG`
- `PYTHONVERBOSE`
- `PYTHONCASEOK`
- `PYTHONDONTWRITEBYTECODE`
- `PYTHONIOENCODING`
- `PYTHONUSERBASE`
- `PYTHONHASHSEED`

(Похоже, что вследствие тонкостей реализации Python, не зависящих от исполнителя PL/Python, некоторые переменные окружения, перечисленные на странице руководства man `python`, действуют только в интерпретаторе для командной строки, но не во встраиваемом интерпретаторе Python.)

Глава 45. Интерфейс программирования сервера

Интерфейс программирования сервера (SPI, Server Programming Interface) даёт разработчикам пользовательских функций на C возможность запускать команды SQL из своих функций или процедур. SPI представляет собой набор интерфейсных функций, упрощающих доступ к анализатору, планировщику и исполнителю запросов. В SPI есть также функции для управления памятью.

> **Примечание**
>
> Доступные процедурные языки предоставляют различные средства для выполнения SQL-команд из функций. Большинство этих средств основаны на SPI, так что эта документация будет полезна и тем, кто использует эти языки.

Учтите, что если команда, вызванная через SPI, прерывается ошибкой, управление не возвращается в вашу функцию на C. Вместо этого происходит откат транзакции или подтранзакции, из которой вызывалась ваша функция. (Это может показаться удивительным с учётом того, что для большинства функций SPI описаны соглашения по возврату ошибок. Однако эти соглашения применимы только к ошибкам, выявляемым в самих функциях SPI.) Получить управление после ошибки можно, только организовав собственную подтранзакцию, окружающую вызовы SPI, в которых возможна ошибка.

Функции SPI выдают неотрицательный результат в случае успеха (либо через возвращаемое целочисленное значение, либо в глобальной переменной `SPI_result`, как описано ниже). В случае ошибки выдаётся отрицательный результат или `NULL`.

Файлы исходного кода, использующие SPI, должны включать заголовочный файл `executor/spi.h`.

45.1. Интерфейсные функции

SPI_connect

SPI_connect, SPI_connect_ext — подключить функцию на C к менеджеру SPI

Синтаксис

```
int SPI_connect(void)
```

```
int SPI_connect_ext(int options)
```

Описание

`SPI_connect` устанавливает подключение вызова функции на C к менеджеру SPI. Данную функцию необходимо использовать, если вы хотите выполнять команды через SPI. Некоторые вспомогательные функции SPI могут вызываться из неподключённых функций.

`SPI_connect_ext` делает то же самое, но принимает один аргумент, через который можно передать дополнительные флаги. В настоящее время поддерживаются следующие флаги:

`SPI_OPT_NONATOMIC`

Переводит подключение SPI в *неатомарный* режим, в котором разрешаются вызовы функций управления транзакциями (`SPI_commit`, `SPI_rollback`). В обычном режиме вызов этих функций приводит к немедленной ошибке.

Вызов `SPI_connect()` равнозначен `SPI_connect_ext(0)`.

Возвращаемое значение

`SPI_OK_CONNECT`

при успехе

`SPI_ERROR_CONNECT`

при ошибке

SPI_finish

SPI_finish — отключить функцию на C от менеджера SPI

Синтаксис

```
int SPI_finish(void)
```

Описание

`SPI_finish` закрывает текущее соединение с менеджером SPI. Эту функцию необходимо вызывать после завершения операций SPI, которые должны выполняться в текущем вызове функции на C. Однако если вы прерываете транзакцию, выполняя `elog(ERROR)`, о закрытии соединения можно не беспокоиться. В этом случае SPI произведёт очистку автоматически.

Возвращаемое значение

`SPI_OK_FINISH`

если отключение выполнено корректно

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой функции на C

SPI_execute

SPI_execute — выполнить команду

Синтаксис

```
int SPI_execute(const char * command, bool read_only, long count)
```

Описание

`SPI_execute` выполняет заданную команду SQL для получения строк в количестве, ограниченном *`count`*. С параметром *`read_only`*, равным `true`, команда должна только читать данные; это несколько сокращает издержки на её выполнение.

Эту функцию можно вызывать только из подключённой функции на C.

Если *`count`* равен 0, команда выполняется для всех строк, к которым она применима. Если *`count`* больше нуля, будет получено не более чем *`count`* строк; выполнение команды остановится при достижении этого предела, практически так же, как и с предложением `LIMIT` в запросе. Например, команда:

```
SPI_execute("SELECT * FROM foo", true, 5);
```

получит из таблицы не более 5 строк. Заметьте, что это ограничение действует, только когда команда действительно возвращает строки. Например, эта команда:

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

вставляет все строки из `bar`, игнорируя параметр *`count`*. Однако команда

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

вставит не более 5 строк, так как её выполнение будет остановлено после получения пятой строки, выданной предложением `RETURNING`.

В одной строке можно передать несколько команд; `SPI_execute` возвращает результат команды, выполненной последней. Параметр *`count`* при этом будет применяться к каждой команде по отдельности (несмотря даже на то, что возвращён будет только последний результат). Это ограничение не будет распространяться на скрытые команды, генерируемые правилами.

Когда параметр *`read_only`* равен `false`, `SPI_execute` увеличивает счётчик команд и получает новый *снимок* перед выполнением каждой очередной команды в строке. Этот снимок фактически не меняется при текущем уровне изоляции транзакций `SERIALIZABLE` или `REPEATABLE READ`, но в режиме `READ COMMITTED` после обновления снимка очередная команда может видеть результаты только что зафиксированных транзакций из других сеансов. Это важно для согласованного поведения, когда команды модифицируют базу данных.

Когда параметр *`read_only`* равен `true`, `SPI_execute` не обновляет снимок и не увеличивает счётчик команд, и допускает в строке команд только `SELECT`. Заданные команды выполняются со снимком, ранее полученным для окружающего запроса. Этот режим выполнения несколько быстрее режима чтения/записи вследствие исключения издержек, связанных с отдельными командами. Он также позволяет создавать подлинно *стабильные* функции: так как последующие вызовы в транзакции будут использовать один снимок, результаты команд не изменятся.

Смешивать команды, только читающие, с командами, читающими и пишущими, в одной процедуре, использующей SPI, обычно неразумно; запросы только на чтение не увидят результатов изменений в базе данных, произведённых пишущими запросами.

Число строк, которые были фактически обработаны (последней) командой, возвращается в глобальной переменной `SPI_processed`. Если эта функция возвращает значение `SPI_OK_SELECT`, `SPI_OK_INSERT_RETURNING`, `SPI_OK_DELETE_RETURNING`, `SPI_OK_UPDATE_RETURNING` или `SPI_OK_MERGE_RETURNING`, вы можете обратиться по глобальному указателю `SPITupleTable *SPI_tuptable` и прочитать строки результата. Некоторые служебные команды (например, `EXPLAIN`) также возвращают наборы строк, и `SPI_tuptable` будет содержать их результаты и в этих случаях. Другие вспомогательные команды (`COPY`, `CREATE TABLE AS`) не возвращают набор строк, так что указатель `SPI_tuptable` равен NULL, но они так же возвращают число обработанных строк в `SPI_processed`.

Структура `SPITupleTable` определена так:

```
typedef struct SPITupleTable
{
    /* Открытые члены */
    TupleDesc   tupdesc;        /* дескриптор кортежа */
    HeapTuple  *vals;           /* массив кортежей */
    uint64      numvals;        /* число фактически представленных кортежей */

    /* Закрытые члены, не предназначенные для внешнего использования */
    uint64      alloced;        /* зарезервированное в памяти число элементов vals */
    MemoryContext tuptabcxt;    /* контекст таблицы результатов в памяти */
    slist_node  next;           /* ссылка для внутреннего обслуживания */
    SubTransactionId subid;     /* подтранзакция, создавшая структуру tuptable */
} SPITupleTable;
```

Поля `tupdesc`, `vals` и `numvals` могут использоваться кодом, вызывающим SPI, остальные поля являются внутренними. `vals` представляет собой массив указателей на кортежи. Число записей в нём указывается в `numvals` (по некоторым историческим причинам это число также возвращается в `SPI_processed`). Поле `tupdesc` содержит дескриптор кортежа, который вы сможете передать функциям SPI, работающими с кортежами.

`SPI_finish` освобождает все структуры `SPITupleTable`, размещённые в памяти для текущей функции на C. Вы можете освободить структуру конкретной результирующей таблицы, если она вам не нужна, вызвав `SPI_freetuptable`.

Аргументы

`const char * command`

строка с командой, которая должна быть выполнена

`bool read_only`

`true` для режима выполнения «только чтение»

`long count`

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

Если команда была выполнена успешно, возвращается одно из следующих (неотрицательных) значений:

`SPI_OK_SELECT`

если выполнялась команда `SELECT` (но не `SELECT INTO`)

`SPI_OK_SELINTO`

если выполнялась команда `SELECT INTO`

`SPI_OK_INSERT`

если выполнялась команда `INSERT`

`SPI_OK_DELETE`

если выполнялась команда `DELETE`

`SPI_OK_UPDATE`

если выполнялась команда `UPDATE`

`SPI_OK_MERGE`

если выполнялась команда `MERGE`

`SPI_OK_INSERT_RETURNING`

если выполнялась команда `INSERT RETURNING`

`SPI_OK_DELETE_RETURNING`

если выполнялась команда `DELETE RETURNING`

`SPI_OK_UPDATE_RETURNING`

если выполнялась команда `UPDATE RETURNING`

`SPI_OK_MERGE_RETURNING`

если выполнялась команда `MERGE RETURNING`

`SPI_OK_UTILITY`

если выполнялась служебная команда (например, `CREATE TABLE`)

`SPI_OK_REWRITTEN`

если команда была преобразована правилом в команду другого вида (например, `UPDATE` стал командой `INSERT`).

В случае ошибки возвращается одно из следующих отрицательных значений:

`SPI_ERROR_ARGUMENT`

если в качестве *`command`* передан `NULL` или *`count`* меньше 0

`SPI_ERROR_COPY`

при попытке выполнить `COPY TO stdout` или `COPY FROM stdin`

`SPI_ERROR_TRANSACTION`

при попытке выполнить команду управления транзакциями (`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED` или любую их вариацию)

`SPI_ERROR_OPUNKNOWN`

если тип команды неизвестен (такого быть не должно)

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой функции на C

Примечания

Все функции SPI, выполняющие запросы, заполняют и `SPI_processed`, и `SPI_tuptable` (только указатель, но не содержимое структуры). Сохраните эти две глобальные переменные в локальных переменных функции на C, если хотите обращаться к таблице результата `SPI_execute` или другой функции, выполняющей запрос, в нескольких вызовах процедуры.

SPI_exec

SPI_exec — выполнить команду чтения/записи

Синтаксис

```
int SPI_exec(const char * command, long count)
```

Описание

`SPI_exec` действует подобно `SPI_execute`, но ей не передаётся параметр *`read_only`* (всегда подразумевается `false`).

Аргументы

`const char *` *`command`*

строка с командой, которая должна быть выполнена

`long` *`count`*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

См. `SPI_execute`.

SPI_execute_extended

SPI_execute_extended — выполнить команду с выделенными параметрами

Синтаксис

```
int SPI_execute_extended(const char *command,
                         const SPIExecuteOptions * options)
```

Описание

`SPI_execute_extended` выполняет команду, которая может включать ссылки на параметры, передаваемые извне. В тексте команды параметры обозначаются символами `$n`; тип и значение каждого такого параметра предоставляются в объекте *`options->params`* (если он передаётся). В *`options`* также могут передаваться различные параметры выполнения.

Как правило, в объекте *`options->params`* все параметры должны иметь пометку `PARAM_FLAG_CONST`, поскольку для запроса всегда используется одноразовый план.

Если значение *`options->dest`* не NULL, то результирующие кортежи передаются в этот объект по мере их формирования исполнителем, а не накапливаются в `SPI_tuptable`. Подготовить и использовать объект `DestReceiver` особенно полезно для запросов, при выполнении которых можно получить множество кортежей, так как этот объект позволяет обрабатывать данные «на лету», не дожидаясь их накопления в памяти.

Аргументы

`const char * command`

строка команды

`const SPIExecuteOptions * options`

структура, содержащая необязательные аргументы

Структуру *`options`* всегда следует сначала полностью заполнять нулями, а затем уже задавать необходимые значения. Это гарантирует совместимость с будущими изменениями в коде: при добавлении в эту структуру любых новых полей нулевое значение будет определять старое поведение. Сейчас в *`options`* есть такие поля:

`ParamListInfo params`

структура данных, содержащая типы и значения параметров запроса; NULL, если их нет

`bool read_only`

`true` для режима выполнения «только чтение»

`bool allow_nonatomic`

`true` для неатомарного выполнения операторов CALL и DO (но это поле игнорируется, если функции `SPI_connect_ext` не был передан флаг `SPI_OPT_NONATOMIC`)

`bool must_return_tuples`

если `true`, вызывает ошибку для типов запросов, не возвращающих кортежи (за исключением случая, когда возвращаются нулевые кортежи)

`uint64 tcount`

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

`DestReceiver * dest`

объект `DestReceiver`, получающий все кортежи, которые выдаёт запрос; если NULL, результирующие кортежи накапливаются в `SPI_tuptable`, как при выполнении `SPI_execute`

`ResourceOwner owner`

Это поле присутствует для согласованности с `SPI_execute_plan_extended`, но игнорируется, поскольку используемый `SPI_execute_extended` план никогда не сохраняется.

Возвращаемое значение

Возвращаемые значения те же, что и у `SPI_execute`.

Если *`options->dest`* имеет значение NULL, то `SPI_processed` и `SPI_tuptable` устанавливаются как при выполнении `SPI_execute`. Если *`options->dest`* — не NULL, то `SPI_processed` принимает нулевое значение, а `SPI_tuptable` — NULL. Если необходимо подсчитать число кортежей, это нужно сделать в коде объекта `DestReceiver`.

SPI_execute_with_args

SPI_execute_with_args — выполнить команду с выделенными параметрами

Синтаксис

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

Описание

`SPI_execute_with_args` выполняет команду, которая может включать ссылки на параметры, передаваемые извне. В тексте команды параметры обозначаются символами `$n`, а в вызове указываются типы данных и значения для каждого такого символа. Параметры *`read_only`* и *`count`* имеют тот же смысл, что и в `SPI_execute`.

Основное преимущество этой функции по сравнению с `SPI_execute` в том, что она позволяет передавать в команду значения данных, не требуя кропотливой подготовки строк, и таким образом сокращает риск атак с SQL-инъекцией.

Подобного результата можно достичь, вызвав `SPI_prepare` и затем `SPI_execute_plan`; однако с данной функцией план запроса всегда подстраивается под переданные конкретные значения параметров. Поэтому для разового выполнения запроса рекомендуется применять эту функцию. Если же одна и та же команда должна выполняться с самыми разными параметрами, какой вариант окажется быстрее, будет зависеть от стоимости повторного планирования и выигрыша от выбора специализированных планов.

Аргументы

`const char * command`

строка команды

`int nargs`

число входных параметров (`$1`, `$2` и т. д.)

`Oid * argtypes`

массив размера *`nargs`*, содержащий OID типов параметров

`Datum * values`

массив размера *`nargs`*, содержащий фактические значения параметров

`const char * nulls`

массив размера *`nargs`*, описывающий, в каких параметрах передаётся NULL

Если в *`nulls`* передаётся `NULL`, `SPI_execute_with_args` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`bool read_only`

`true` для режима выполнения «только чтение» `long` *`count`*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

Возвращаемые значения те же, что и у `SPI_execute`.

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

SPI_prepare

SPI_prepare — подготовить оператор, но пока не выполнять его

Синтаксис

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

Описание

`SPI_prepare` создаёт и возвращает подготовленный оператор для заданной команды. Подготовленный оператор может быть затем неоднократно выполнен функцией `SPI_execute_plan`.

Когда одна и та же или похожие команды выполняются неоднократно, обычно выгоднее произвести анализ запроса только раз, а ещё выгоднее может быть повторно использовать план выполнения команды. `SPI_prepare` преобразует строку команды в подготовленный оператор, включающий в себя результаты анализа запроса. Подготовленный оператор также оставляет место для кеширования плана выполнения, если выбор специализированного плана для каждого выполнения не принесёт пользы.

Подготавливаемую команду можно сделать более общей, записав параметры (`$1`, `$2`, etc.) вместо значений, задаваемыми константами в обычной команде. Фактические значения параметров в этом случае будут задаваться при вызове `SPI_execute_plan`. Это позволяет применять подготовленную команду в более широком круге ситуаций, чем это возможно без параметров.

Оператор, возвращаемый функцией `SPI_prepare`, может использоваться только в текущем вызове функции на C, так как `SPI_finish` освобождает память, выделенную для такого оператора. Но этот оператор может быть сохранён на будущее с помощью функций `SPI_keepplan` или `SPI_saveplan`.

Аргументы

`const char * command`

строка команды

`int nargs`

число входных параметров (`$1`, `$2` и т. д.)

`Oid * argtypes`

указатель на массив, содержащий OID типов параметров

Возвращаемое значение

`SPI_prepare` возвращает ненулевой указатель на `SPIPlan`, скрытую структуру, представляющую подготовленный оператор. В случае ошибки возвращается `NULL`, а в `SPI_result` устанавливается один из кодов ошибок, определённых для `SPI_execute`, за исключением того, что код `SPI_ERROR_ARGUMENT` устанавливается, когда *`command`* — `NULL`, когда *`nargs`* меньше 0 или когда *`nargs`* больше 0, а *`argtypes`* — `NULL`.

Примечания

Если параметры не определены, при первом использовании `SPI_execute_plan` создаётся общий план, который затем будет применяться при последующих вызовах. Если же присутствуют параметры, `SPI_execute_plan` будет создавать специализированные планы для конкретных значений параметров. После достаточного количества использований полученного подготовленного оператора, функция `SPI_execute_plan` построит общий план, и если он не будет значительно дороже специализированных, она начнёт использовать его, а не будет строить план заново. Если это поведение по умолчанию не устраивает, его можно изменить, передав флаг `CURSOR_OPT_GENERIC_PLAN` или `CURSOR_OPT_CUSTOM_PLAN` в `SPI_prepare_cursor`, чтобы ограничиться использованием только общего или специализированных планов, соответственно.

Хотя основной смысл подготовленного оператора в том, чтобы избежать повторного разбора и планирования запроса, Postgres Pro всё же будет принудительно повторять разбор и планирование запроса перед его выполнением, если со времени предыдущего использования подготовленного оператора произойдут изменения определений (DDL) объектов базы, задействованных в этом запросе. Также, если перед очередным использованием было изменено значение search_path, запрос будет разобран заново с новым значением `search_path`. (Последняя особенность появилась в PostgreSQL 9.3.) Чтобы узнать о поведении подготовленных операторов больше, обратитесь к PREPARE.

Эту функцию следует вызывать только из подключённой функции на C.

`SPIPlanPtr` объявлен в `spi.h` как указатель на скрытую структуру. Пытаться обращаться к её содержимому напрямую не стоит, так как ваш код скорее всего сломается при выходе новых версий Postgres Pro.

Имя `SPIPlanPtr` объясняется отчасти историческими причинами, так как теперь эта структура может не содержать собственно план выполнения.

SPI_prepare_cursor

SPI_prepare_cursor — подготовить оператор, но пока не выполнять его

Синтаксис

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

Описание

Функция `SPI_prepare_cursor` равнозначна `SPI_prepare`, за исключением того, что ей можно передать «параметры курсора». Эти параметры задаются битовой маской со значениями, определёнными в `nodes/parsenodes.h` для поля `options` структуры `DeclareCursorStmt`. `SPI_prepare` подразумевает, что эти параметры всегда нулевые.

Эта функция устарела, используйте вместо неё `SPI_prepare_extended`.

Аргументы

`const char * command`

строка команды

`int nargs`

число входных параметров (`$1`, `$2` и т. д.)

`Oid * argtypes`

указатель на массив, содержащий OID типов параметров

`int cursorOptions`

битовая маска параметров курсора; 0 выбирает поведение по умолчанию

Возвращаемое значение

`SPI_prepare_cursor` возвращает результат по тем же соглашениям, что и `SPI_prepare`.

Примечания

К числу полезных бит, которые можно задать в *`cursorOptions`*, относятся `CURSOR_OPT_SCROLL`, `CURSOR_OPT_NO_SCROLL`, `CURSOR_OPT_FAST_PLAN`, `CURSOR_OPT_GENERIC_PLAN` и `CURSOR_OPT_CUSTOM_PLAN`. Заметьте, что параметр `CURSOR_OPT_HOLD` игнорируется.

SPI_prepare_extended

SPI_prepare_extended — подготовить оператор, но пока не выполнять его

Синтаксис

```
SPIPlanPtr SPI_prepare_extended(const char * command,
                                const SPIPrepareOptions * options)
```

Описание

`SPI_prepare_extended` создаёт и возвращает подготовленный оператор для заданной команды, но не выполняет саму команду. Эта функция равнозначна `SPI_prepare`, но позволяет вызывающему управлять разбором ссылок на внешние параметры, а также другими аспектами разбора и планирования запросов.

Аргументы

`const char * command`

строка команды

`const SPIPrepareOptions * options`

структура, содержащая необязательные аргументы

Структуру *`options`* всегда следует сначала полностью заполнять нулями, а затем уже задавать необходимые значения. Это гарантирует совместимость с будущими изменениями в коде: при добавлении в эту структуру любых новых полей нулевое значение будет определять старое поведение. Сейчас в *`options`* есть такие поля:

`ParserSetupHook parserSetup`

Функция настройки обработчиков разбора

`void * parserSetupArg`

аргумент для сквозной передачи в *`parserSetup`*

`RawParseMode parseMode`

режим для простого разбора; `RAW_PARSE_DEFAULT` (ноль) выбирает поведение по умолчанию

`int cursorOptions`

битовая маска параметров курсора; 0 выбирает поведение по умолчанию

Возвращаемое значение

`SPI_prepare_extended` возвращает результат по тем же соглашениям, что и `SPI_prepare`.

SPI_prepare_params

SPI_prepare_params — подготовить оператор, но пока не выполнять его

Синтаксис

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

Описание

`SPI_prepare_params` создаёт и возвращает подготовленный оператор для заданной команды, но не выполняет саму команду. Эта функция равнозначна `SPI_prepare_cursor`, но позволяет вызывающему дополнительно установить функции-обработчики для управления разбором ссылок на внешние параметры.

Эта функция устарела, используйте вместо неё `SPI_prepare_extended`.

Аргументы

`const char * command`

строка команды

`ParserSetupHook parserSetup`

Функция настройки обработчиков разбора

`void * parserSetupArg`

аргумент для сквозной передачи в `parserSetup`

`int cursorOptions`

битовая маска параметров курсора; 0 выбирает поведение по умолчанию

Возвращаемое значение

`SPI_prepare_params` возвращает результат по тем же соглашениям, что и `SPI_prepare`.

SPI_getargcount

SPI_getargcount — получить число аргументов, требующихся оператору, подготовленному функцией `SPI_prepare`

Синтаксис

```
int SPI_getargcount(SPIPlanPtr plan)
```

Описание

`SPI_getargcount` возвращает число аргументов, требующихся для выполнения оператора, подготовленного функцией `SPI_prepare`.

Аргументы

`SPIPlanPtr plan`

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

Возвращаемое значение

Число аргументов, которое ожидает план, заданный параметром *`plan`*. Если значение *`plan`* неверное или `NULL`, в `SPI_result` устанавливается код `SPI_ERROR_ARGUMENT`, а функция возвращает -1.

SPI_getargtypeid

SPI_getargtypeid — получить OID типа аргумента для оператора, подготовленного функцией `SPI_prepare`

Синтаксис

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

Описание

`SPI_getargtypeid` возвращает OID, представляющий тип аргумента под номером *`argIndex`* оператора, подготовленного функцией `SPI_prepare`. Первый аргумент идёт под номером ноль.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`int` *`argIndex`*

индекс аргумента, начиная с нуля

Возвращаемое значение

OID типа аргумента с заданным индексом. Если значение *`plan`* неверное или `NULL`, либо *`argIndex`* меньше 0 или не меньше числа аргументов, объявленных при подготовке плана (передаваемого в *`plan`*), в `SPI_result` устанавливается `SPI_ERROR_ARGUMENT` и возвращается `InvalidOid`.

SPI_is_cursor_plan

SPI_is_cursor_plan — выдать `true`, если оператор, подготовленный функцией `SPI_prepare`, можно использовать с `SPI_cursor_open`

Синтаксис

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

Описание

`SPI_is_cursor_plan` возвращает `true`, если оператор, подготовленный функцией `SPI_prepare`, можно передать в качестве аргумента `SPI_cursor_open`, или `false` в противном случае. Для положительного ответа в *`plan`* должна быть представлена одна команда, и эта команда должна возвращать кортежи; например, `SELECT` может быть подходящей командой, если он не содержит предложения `INTO`, а `UPDATE` подходит, только если он содержит предложение `RETURNING`.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

Возвращаемое значение

Значение `true` или `false`, показывающее, можно ли для подготовленного оператора, заданного параметром *`plan`*, получить курсор, при `SPI_result` равном нулю. Если дать ответ невозможно (например, если значение *`plan`* неверное или `NULL`, либо вызывающий не подключён к SPI), в `SPI_result` устанавливается соответствующий код ошибки и возвращается `false`.

SPI_execute_plan

SPI_execute_plan — выполнить оператор, подготовленный функцией `SPI_prepare`

Синтаксис

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

Описание

`SPI_execute_plan` выполняет оператор, подготовленный функцией `SPI_prepare` или родственными ей. Параметры *`read_only`* и *`count`* имеют тот же смысл, что и в `SPI_execute`.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`Datum *` *`values`*

Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

`const char *` *`nulls`*

Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *`nulls`* передаётся `NULL`, `SPI_execute_plan` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

`long` *`count`*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

Возвращаемые значения те же, что и у `SPI_execute`, со следующими дополнительными вариантами ошибок (отрицательных результатов):

`SPI_ERROR_ARGUMENT`

Если *`plan`* неверный или `NULL`, либо *`count`* меньше 0

`SPI_ERROR_PARAM`

Если в *`values`* передан `NULL` и *`plan`* был подготовлен с другими параметрами

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

SPI_execute_plan_extended

SPI_execute_plan_extended — выполнить оператор, подготовленный функцией `SPI_prepare`

Синтаксис

```
int SPI_execute_plan_extended(SPIPlanPtr plan,
                              const SPIExecuteOptions * options)
```

Описание

`SPI_execute_plan_extended` выполняет оператор, подготовленный функцией `SPI_prepare` или родственными ей. Данная функция равнозначна `SPI_execute_plan`, не считая того, что по-другому представляется информация о значениях параметров, передаваемых запросу. Также ей могут передаваться дополнительные параметры для управления выполнением запроса.

Значения параметров запроса передаются в структуре `ParamListInfo`; это удобно, когда имеющиеся значения уже представлены в этом формате. Через функции-обработчики, устанавливаемые в `ParamListInfo`, можно также передавать динамические наборы параметров.

Кроме того, кортежи результата могут не только накапливаться в структуре `SPI_tuptable`, но и передаваться заданному вызывающим кодом объекту `DestReceiver` по мере формирования очередного кортежа исполнителем. Особенно полезно использовать этот объект в запросах, которые могут выдать множество кортежей, поскольку данные будут обрабатываться «на лету», а не накапливаться в памяти.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`const SPIExecuteOptions *` *`options`*

структура, содержащая необязательные аргументы

Структуру *`options`* всегда следует сначала полностью заполнять нулями, а затем уже задавать необходимые значения. Это гарантирует совместимость с будущими изменениями в коде: при добавлении в эту структуру любых новых полей нулевое значение будет определять старое поведение. Сейчас в *`options`* есть такие поля:

`ParamListInfo` *`params`*

структура данных, содержащая типы и значения параметров запроса; NULL, если их нет

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

`bool` *`allow_nonatomic`*

`true` для неатомарного выполнения операторов CALL и DO (но это поле игнорируется, если функции `SPI_connect_ext` не был передан флаг `SPI_OPT_NONATOMIC`)

`bool` *`must_return_tuples`*

если `true`, вызывает ошибку для типов запросов, не возвращающих кортежи (за исключением случая, когда возвращаются нулевые кортежи)

`uint64` *`tcount`*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

`DestReceiver *` *`dest`*

объект `DestReceiver`, получающий все кортежи, которые выдаёт запрос; если NULL, результирующие кортежи накапливаются в `SPI_tuptable`, как при выполнении `SPI_execute_plan`

`ResourceOwner` *`owner`*

Владелец ресурса, который будет содержать счётчик ссылок на план во время его выполнения. Если NULL, используется текущий владелец ресурса. Параметр игнорируется для несохранённых планов, поскольку SPI не получает счётчики ссылок на них.

Возвращаемое значение

Возвращаемые значения те же, что и у `SPI_execute_plan`.

Если *`options->dest`* имеет значение NULL, `SPI_processed` и `SPI_tuptable` устанавливаются как при выполнении `SPI_execute_plan`. Если *`options->dest`* — не NULL, то `SPI_processed` принимает нулевое значение, а `SPI_tuptable` — NULL. Если необходимо подсчитать число кортежей, это нужно сделать в коде объекта `DestReceiver`.

SPI_execute_plan_with_paramlist

SPI_execute_plan_with_paramlist — выполнить оператор, подготовленный функцией `SPI_prepare`

Синтаксис

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

Описание

`SPI_execute_plan_with_paramlist` выполняет оператор, подготовленный функцией `SPI_prepare`. Данная функция равнозначна `SPI_execute_plan`, не считая того, что информация о значениях параметров, передаваемых запросу, представляется по-другому. Представление `ParamListInfo` может быть удобным для передачи значений, уже имеющих нужный формат. Эта функция также поддерживает динамические наборы параметров, которые реализуются через функции-обработчики, устанавливаемые в `ParamListInfo`.

Эта функция устарела, используйте вместо неё `SPI_execute_plan_extended`.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`ParamListInfo` *`params`*

структура данных, содержащая типы и значения параметров; NULL, если их нет

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

`long` *`count`*

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

Возвращаемые значения те же, что и у `SPI_execute_plan`.

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute_plan`, если вызов был успешным.

SPI_execp

SPI_execp — выполнить оператор в режиме чтения/записи

Синтаксис

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

Описание

`SPI_execp` действует подобно `SPI_execute_plan`, но ей не передаётся параметр *`read_only`* (всегда подразумевается `false`).

Аргументы

`SPIPlanPtr plan`

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`Datum * values`

Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

`const char * nulls`

Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *`nulls`* передаётся `NULL`, `SPI_execp` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`long count`

максимальное число строк, которое должно быть возвращено; с `0` ограничения нет

Возвращаемое значение

См. `SPI_execute_plan`.

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

SPI_cursor_open

SPI_cursor_open — открыть курсор для оператора, созданного функцией `SPI_prepare`

Синтаксис

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

Описание

`SPI_cursor_open` открывает курсор (внутри называемый порталом), через который будет выполняться оператор, подготовленный функцией `SPI_prepare`. Параметры этой функции имеют тот же смысл, что и соответствующие параметры `SPI_execute_plan`.

Применение курсора по сравнению с непосредственным выполнением оператора даёт двойную выгоду. Во-первых, строки результата можно получать в небольших количествах, без риска исчерпать всю память при выполнении запросов, возвращающих много строк. Во-вторых, портал может существовать и после завершения текущей функции на C (на самом деле он может просуществовать до конца текущей транзакции). Возвратив имя портала в код, вызывающий функцию на C, можно организовать выдачу результата в виде набора строк.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы

`const char * name`

имя портала, либо `NULL`, чтобы имя выбрала система

`SPIPlanPtr plan`

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`Datum * values`

Массив фактических значений параметров. Его размер должен равняться числу аргументов оператора.

`const char * nulls`

Массив, описывающий, в каких параметрах передаётся NULL. Должен иметь размер, равный числу аргументов оператора.

Если в *`nulls`* передаётся `NULL`, `SPI_cursor_open` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`bool read_only`

`true` для режима выполнения «только чтение»

Возвращаемое значение

Указатель на портал, содержащий курсор. Заметьте, что соглашение о возврате ошибок отсутствует; все ошибки выдаются через `elog`.

SPI_cursor_open_with_args

SPI_cursor_open_with_args — открывает курсор для запроса с параметрами

Синтаксис

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

Описание

`SPI_cursor_open_with_args` открывает курсор (внутри называемый порталом) для выполнения заданного запроса. Большинство параметров имеют тот же смысл, что и соответствующие параметры функций `SPI_prepare_cursor` и `SPI_cursor_open`.

Для разового выполнения запроса эту функцию следует предпочесть `SPI_prepare_cursor` с последующей `SPI_cursor_open`. Если же одна и та же команда должна выполняться с самыми разными параметрами, какой вариант окажется быстрее, будет зависеть от стоимости повторного планирования и выигрыша от выбора специализированных планов.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Эта функция устарела, используйте вместо неё `SPI_cursor_parse_open`, предоставляющую такую же функциональность, но использующую более современный API для обработки параметров запроса.

Аргументы

`const char * name`

имя портала, либо `NULL`, чтобы имя выбрала система

`const char * command`

строка команды

`int nargs`

число входных параметров (`$1`, `$2` и т. д.)

`Oid * argtypes`

массив размера *`nargs`*, содержащий OID типов параметров

`Datum * values`

массив размера *`nargs`*, содержащий фактические значения параметров

`const char * nulls`

массив размера *`nargs`*, описывающий, в каких параметрах передаётся NULL

Если в *`nulls`* передаётся `NULL`, `SPI_cursor_open_with_args` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль (`'\0'`) в конце не нужен.

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

`int` *`cursorOptions`*

битовая маска параметров курсора; 0 выбирает поведение по умолчанию

Возвращаемое значение

Указатель на портал, содержащий курсор. Заметьте, что соглашение о возврате ошибок отсутствует; все ошибки выдаются через `elog`.

SPI_cursor_open_with_paramlist

SPI_cursor_open_with_paramlist — открыть курсор с параметрами

Синтаксис

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

Описание

`SPI_cursor_open_with_paramlist` открывает курсор (внутри называемый порталом) для выполнения оператора, подготовленного функцией `SPI_prepare`. Эта функция равнозначна `SPI_cursor_open`, не считая того, что информация о значениях параметров, передаваемых запросу, представляется по-другому. Представление `ParamListInfo` может быть удобным для передачи значений, уже имеющих нужный формат. Эта функция также поддерживает динамические наборы параметров через функции-обработчики, устанавливаемые в `ParamListInfo`.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы

`const char *` *`name`*

имя портала, либо `NULL`, чтобы имя выбрала система

`SPIPlanPtr` *`plan`*

подготовленный оператор (возвращаемый функцией `SPI_prepare`)

`ParamListInfo` *`params`*

структура данных, содержащая типы и значения параметров; NULL, если их нет

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

Возвращаемое значение

Указатель на портал, содержащий курсор. Заметьте, что соглашение о возврате ошибок отсутствует; все ошибки выдаются через `elog`.

SPI_cursor_parse_open

SPI_cursor_parse_open — открывает курсор для строки запроса и набора параметров

Синтаксис

```
Portal SPI_cursor_parse_open(const char *name,
                             const char *command,
                             const SPIParseOpenOptions * options)
```

Описание

`SPI_cursor_parse_open` открывает курсор (внутри называемый порталом) для выполнения заданного запроса. Эта функция действует подобно функции `SPI_prepare_cursor` с последующей `SPI_cursor_open_with_paramlist`, за исключением того, что ей для обработки всех ссылок на параметры в строке запроса передаётся объект `ParamListInfo`.

Для разового выполнения запроса эту функцию следует предпочесть `SPI_prepare_cursor` с последующей `SPI_cursor_open_with_paramlist`. Если же одна и та же команда должна выполняться с самыми разными параметрами, какой вариант окажется быстрее, будет зависеть от стоимости повторного планирования и выигрыша от выбора специализированных планов.

Как правило, в объекте *`options->params`* все параметры должны иметь пометку `PARAM_FLAG_CONST`, поскольку для запроса всегда используется одноразовый план.

Переданные значения параметров копируются в портал курсора, так что их можно освободить и во время существования курсора.

Аргументы

`const char *` *`name`*

имя портала, либо `NULL`, чтобы имя выбрала система

`const char *` *`command`*

строка команды

`const SPIParseOpenOptions *` *`options`*

структура, содержащая необязательные аргументы

Структуру *`options`* всегда следует сначала полностью заполнять нулями, а затем уже задавать необходимые значения. Это гарантирует совместимость с будущими изменениями в коде: при добавлении в эту структуру любых новых полей нулевое значение будет определять старое поведение. Сейчас в *`options`* есть такие поля:

`ParamListInfo` *`params`*

структура данных, содержащая типы и значения параметров запроса; NULL, если их нет

`int` *`cursorOptions`*

битовая маска параметров курсора; 0 выбирает поведение по умолчанию

`bool` *`read_only`*

`true` для режима выполнения «только чтение»

Возвращаемое значение

Указатель на портал, содержащий курсор. Заметьте, что соглашение о возврате ошибок отсутствует; все ошибки выдаются через `elog`.

SPI_cursor_find

SPI_cursor_find — найти существующий курсор по имени

Синтаксис

```
Portal SPI_cursor_find(const char * name)
```

Описание

`SPI_cursor_find` находит существующий портал по имени. В основном это полезно для разрешения имени курсора, возвращённого в текстовом виде какой-то другой функцией.

Аргументы

`const char * name`

имя портала

Возвращаемое значение

указатель на портал с заданным именем или `NULL`, если такой портал не найден

Примечания

Помните, что эта функция может возвращать объект `Portal`, который не имеет свойств, подобных курсору, например, она может не возвращать кортежи. Если просто передать указатель `Portal` другим функциям SPI, они могут защититься от таких случаев, но при непосредственной проверке `Portal` следует проявлять осторожность.

SPI_cursor_fetch

SPI_cursor_fetch — выбрать строки через курсор

Синтаксис

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

Описание

`SPI_cursor_fetch` выбирает некоторое количество строк через курсор. Эта функция реализует подмножество возможностей SQL-команды `FETCH` (расширенную функциональность предоставляет `SPI_scroll_cursor_fetch`).

Аргументы

`Portal` *`portal`*

портал, содержащий курсор

`bool` *`forward`*

true для выборки с перемещением вперёд, false — назад

`long` *`count`*

максимальное число строк, которое нужно выбрать

Возвращаемое значение

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

Примечания

Выборка назад может не поддерживаться, если план курсора был создан без параметра `CURSOR_OPT_SCROLL`.

SPI_cursor_move

SPI_cursor_move — переместить курсор

Синтаксис

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

Описание

`SPI_cursor_move` перемещает курсор на несколько строк. Эта функция реализует подмножество возможностей SQL-команды `MOVE` (расширенную функциональность предоставляет `SPI_scroll_cursor_move`).

Аргументы

`Portal` *`portal`*

портал, содержащий курсор

`bool` *`forward`*

true для перемещения вперёд, false — назад

`long` *`count`*

максимальное число строк, на какое возможно перемещение

Примечания

Перемещение назад может не поддерживаться, если план курсора был создан без параметра `CURSOR_OPT_SCROLL`.

SPI_scroll_cursor_fetch

SPI_scroll_cursor_fetch — выбрать строки через курсор

Синтаксис

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

Описание

`SPI_scroll_cursor_fetch` выбирает некоторое количество строк через курсор. Её функциональность равнозначна `FETCH` в SQL.

Аргументы

`Portal portal`

портал, содержащий курсор

`FetchDirection direction`

один из вариантов: `FETCH_FORWARD`, `FETCH_BACKWARD`, `FETCH_ABSOLUTE` или `FETCH_RELATIVE`

`long count`

число строк, выбираемых с направлением `FETCH_FORWARD` или `FETCH_BACKWARD`; абсолютный номер выбираемой строки с вариантом `FETCH_ABSOLUTE`; либо относительный номер выбираемой строки с вариантом `FETCH_RELATIVE`

Возвращаемое значение

Переменные `SPI_processed` и `SPI_tuptable` устанавливаются как в `SPI_execute`, если вызов был успешным.

Примечания

Подробнее о параметрах *`direction`* и *`count`* рассказывается в описании SQL-команды FETCH.

Варианты направления, отличные от `FETCH_FORWARD`, могут не поддерживаться, если план курсора был создан без параметра `CURSOR_OPT_SCROLL`.

SPI_scroll_cursor_move

SPI_scroll_cursor_move — переместить курсор

Синтаксис

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                            long count)
```

Описание

`SPI_scroll_cursor_move` перемещает курсор на несколько строк. Её функциональность равнозначна `MOVE` в SQL.

Аргументы

`Portal portal`

портал, содержащий курсор

`FetchDirection direction`

один из вариантов: `FETCH_FORWARD`, `FETCH_BACKWARD`, `FETCH_ABSOLUTE` или `FETCH_RELATIVE`

`long count`

число строк, на которое сдвигается курсор, с направлением `FETCH_FORWARD` или `FETCH_BACKWARD`; абсолютный номер строки, к которой переходит курсор, с направлением `FETCH_ABSOLUTE`; либо относительный номер строки, к которой переходит курсор, с направлением `FETCH_RELATIVE`

Возвращаемое значение

В случае успеха переменная `SPI_processed` устанавливается как в `SPI_execute`. В `SPI_tuptable` оказывается `NULL`, так как эта функция не возвращает никакие строки.

Примечания

Подробнее о параметрах *`direction`* и *`count`* рассказывается в описании SQL-команды FETCH.

Варианты направления, отличные от `FETCH_FORWARD`, могут не поддерживаться, если план курсора был создан без параметра `CURSOR_OPT_SCROLL`.

SPI_cursor_close

SPI_cursor_close — закрыть курсор

Синтаксис

```
void SPI_cursor_close(Portal portal)
```

Описание

`SPI_cursor_close` закрывает ранее созданный курсор и освобождает память, занятую его порталом.

Все открытые курсоры закрываются автоматически в конце транзакции. Вызывать `SPI_cursor_close` может потребоваться, только если возникает желание освободить ресурсы скорее.

Аргументы

`Portal` *`portal`*

портал, содержащий курсор

SPI_keepplan

SPI_keepplan — сохранить подготовленный оператор

Синтаксис

```
int SPI_keepplan(SPIPlanPtr plan)
```

Описание

`SPI_keepplan` закрепляет переданный оператор (подготовленный функцией `SPI_prepare`), чтобы он не был ликвидирован функцией `SPI_finish` или диспетчером транзакций. Это даёт возможность повторно использовать подготовленные операторы при последующих вызовах вашей функции на C в текущем сеансе.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор, который нужно сохранить

Возвращаемое значение

0 в случае успеха; `SPI_ERROR_ARGUMENT`, если *`plan`* неверный или `NULL`

Примечания

Переданный оператор перемещается в постоянное хранилище путём смены указателя (копировать данные не требуется). Если позже вы захотите удалить его, выполните для него `SPI_freeplan`.

SPI_saveplan

SPI_saveplan — сохранить подготовленный оператор

Синтаксис

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

Описание

`SPI_saveplan` копирует переданный оператор (подготовленный функцией `SPI_prepare`) в память, чтобы он не был ликвидирован функцией `SPI_finish` или менеджером транзакций, и возвращает указатель на скопированный оператор. Это даёт возможность повторно использовать подготовленные операторы при последующих вызовах вашей функции на C в текущем сеансе.

Аргументы

`SPIPlanPtr` *`plan`*

подготовленный оператор, который нужно сохранить

Возвращаемое значение

Указатель на скопированный оператор, либо `NULL` в случае ошибки. При ошибке `SPI_result` принимает одно из этих значений:

`SPI_ERROR_ARGUMENT`

если *`plan`* неверный или `NULL`

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой функции на C

Примечания

Изначально переданный оператор не освобождается, поэтому вы можете выполнить `SPI_freeplan` для него, чтобы высвободить память до `SPI_finish`.

В большинстве случаев `SPI_keepplan` предпочтительнее данной функции, так как она даёт примерно тот же результат, но обходится без физического копирования структур данных подготовленного оператора.

SPI_register_relation

SPI_register_relation — сделать эфемерное именованное отношение доступным по имени в запросах SPI

Синтаксис

```
int SPI_register_relation(EphemeralNamedRelation enr)
```

Описание

`SPI_register_relation` делает эфемерное именованное отношение (со связанной информацией) доступным в запросах, планируемых и выполняемых через текущее подключение SPI.

Аргументы

`EphemeralNamedRelation` *`enr`*

запись эфемерного именованного отношения в реестре

Возвращаемое значение

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

`SPI_OK_REL_REGISTER`

если отношение было успешно зарегистрировано по имени

В случае ошибки возвращается одно из следующих отрицательных значений:

`SPI_ERROR_ARGUMENT`

если `NULL` передан в *`enr`* или в поле `name`

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой функции на C

`SPI_ERROR_REL_DUPLICATE`

если имя, заданное в поле `name` структуры *`enr`*, уже зарегистрировано для этого отношения

SPI_unregister_relation

SPI_unregister_relation — удалить эфемерное именованное отношение из реестра

Синтаксис

```
int SPI_unregister_relation(const char * name)
```

Описание

`SPI_unregister_relation` удаляет эфемерное именованное отношение из реестра для текущего подключения.

Аргументы

`const char * name`

имя записи отношения в реестре

Возвращаемое значение

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

`SPI_OK_REL_UNREGISTER`

если совокупность кортежей была успешно удалена из реестра

В случае ошибки возвращается одно из следующих отрицательных значений:

`SPI_ERROR_ARGUMENT`

если в *name* передан `NULL`

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой функции на C

`SPI_ERROR_REL_NOT_FOUND`

если *name* не находится в реестре для текущего подключения

SPI_register_trigger_data

SPI_register_trigger_data — сделать эфемерные данные триггера доступными в запросах SPI

Синтаксис

```
int SPI_register_trigger_data(TriggerData *tdata)
```

Описание

`SPI_register_trigger_data` делает эфемерные отношения, которые перехватывает триггер, доступными для запросов, планируемых и выполняемых через текущее подключение SPI. В настоящее время это переходные таблицы, перехватываемые триггером `AFTER`, определённым с предложением `REFERENCING OLD/NEW TABLE AS`. Эта функция должна вызываться функцией, реализующей триггер на языке программирования, после подключения.

Аргументы

`TriggerData *tdata`

объект `TriggerData`, передаваемый функцией, реализующей триггер, через `fcinfo->context`

Возвращаемое значение

Если команда была выполнена успешно, возвращается следующее (неотрицательное) значение:

`SPI_OK_TD_REGISTER`

если перехваченные данные триггера (при наличии) были успешно зарегистрированы

В случае ошибки возвращается одно из следующих отрицательных значений:

`SPI_ERROR_ARGUMENT`

если в *`tdata`* передан `NULL`

`SPI_ERROR_UNCONNECTED`

если вызывается из неподключённой функции на C

`SPI_ERROR_REL_DUPLICATE`

если имя в любом из переходных отношений в данных триггера уже зарегистрировано для этого подключения

45.2. Вспомогательные интерфейсные функции

Функции, описанные здесь, предоставляют возможности для извлечения информации из наборов результатов, возвращаемых `SPI_execute` и другими функциями SPI.

Все функции, описанные в этом разделе, могут использоваться и в подключённых, и в неподключённых функциях на C.

SPI_fname

SPI_fname — определить имя столбца с заданным номером

Синтаксис

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

Описание

`SPI_fname` возвращает копию имени столбца с заданным номером. (Когда эта копия имени будет не нужна, её можно освободить с помощью `pfree`.)

Аргументы

`TupleDesc` *`rowdesc`*

описание строк

`int` *`colnumber`*

номер столбца (начиная с 1)

Возвращаемое значение

Имя столбца; `NULL`, если *`colnumber`* вне допустимого диапазона. В случае ошибки в `SPI_result` устанавливается `SPI_ERROR_NOATTRIBUTE`.

SPI_fnumber

SPI_fnumber — определить номер столбца с заданным именем

Синтаксис

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

Описание

`SPI_fnumber` возвращает номер столбца, имеющего заданное имя.

Если *`colname`* ссылается на системный столбец (например, `ctid`), возвращается соответствующий отрицательный номер столбца. Вызывающий должен проверять, не была ли возвращена ошибка, сравнивая значение результата именно с `SPI_ERROR_NOATTRIBUTE`; проверка результата по условию меньше или равно нулю не будет корректной, если только системные столбцы не должны исключаться.

Аргументы

`TupleDesc` *`rowdesc`*

описание строк

`const char *` *`colname`*

имя столбца

Возвращаемое значение

Номер столбца (начиная с 1 для столбцов, создаваемых пользователем), либо `SPI_ERROR_NOATTRIBUTE`, если столбец с заданным именем не найден.

SPI_getvalue

SPI_getvalue — получить строковое значение указанного столбца

Синтаксис

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

Описание

`SPI_getvalue` возвращает строковое представление значения указанного столбца.

Результат возвращается в памяти, размещённой функцией `palloc`. (Когда он будет не нужен, эту память можно освободить с помощью `pfree`.)

Аргументы

`HeapTuple row`

строка с нужными данными

`TupleDesc rowdesc`

описание строк

`int colnumber`

номер столбца (начиная с 1)

Возвращаемое значение

Значение столбца, либо `NULL`, если столбец содержит NULL, *`colnumber`* вне допустимого диапазона (в `SPI_result` при этом устанавливается `SPI_ERROR_NOATTRIBUTE`) или если отсутствует функция вывода (в `SPI_result` устанавливается `SPI_ERROR_NOOUTFUNC`).

SPI_getbinval

SPI_getbinval — получить двоичное значение указанного столбца

Синтаксис

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

Описание

`SPI_getbinval` возвращает значение указанного столбца во внутренней форме (в структуре `Datum`).

Эта функция не выделяет новый блок памяти для данных. В случае с типом, передаваемым по ссылке, возвращаемым значением будет указатель на переданную строку данных.

Аргументы

`HeapTuple` *`row`*

строка с нужными данными

`TupleDesc` *`rowdesc`*

описание строк

`int` *`colnumber`*

номер столбца (начиная с 1)

`bool *` *`isnull`*

признак того, что столбец содержит NULL

Возвращаемое значение

Возвращается двоичное значение столбца. Если этот столбец содержит NULL, переменной, на которую указывает *`isnull`*, присваивается true; в противном случае — false.

При ошибке в `SPI_result` устанавливается `SPI_ERROR_NOATTRIBUTE`.

SPI_gettype

SPI_gettype — получить имя типа данных указанного столбца

Синтаксис

`char * SPI_gettype(TupleDesc rowdesc, int colnumber)`

Описание

`SPI_gettype` возвращает копию имени типа данных указанного столбца. (Когда эта копия имени будет не нужна, её можно освободить с помощью `pfree`.)

Аргументы

`TupleDesc rowdesc`

описание строк

`int colnumber`

номер столбца (начиная с 1)

Возвращаемое значение

Имя типа данных указанного столбца, либо `NULL` в случае ошибки. При ошибке в `SPI_result` устанавливается `SPI_ERROR_NOATTRIBUTE`.

SPI_gettypeid

SPI_gettypeid — получить OID типа данных указанного столбца

Синтаксис

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

Описание

`SPI_gettypeid` возвращает OID типа данных указанного столбца.

Аргументы

`TupleDesc` *`rowdesc`*

описание строк

`int` *`colnumber`*

номер столбца (начиная с 1)

Возвращаемое значение

OID типа данных указанного столбца, либо `InvalidOid` в случае ошибки. При ошибке в `SPI_result` устанавливается `SPI_ERROR_NOATTRIBUTE`.

SPI_getrelname

SPI_getrelname — возвращает имя указанного отношения

Синтаксис

```
char * SPI_getrelname(Relation rel)
```

Описание

`SPI_getrelname` возвращает копию имени указанного отношения. (Когда эта копия имени будет не нужна, её можно освободить с помощью `pfree`.)

Аргументы

`Relation rel`

целевое отношение

Возвращаемое значение

Имя указанного отношения.

SPI_getnspname

SPI_getnspname — возвращает пространство имён указанного отношения

Синтаксис

```
char * SPI_getnspname(Relation rel)
```

Описание

`SPI_getnspname` возвращает копию имени пространства имён, к которому принадлежит указанное отношение (`Relation`). Пространство имён по-другому называется схемой отношения. Когда значение, возвращённое этой функцией, будет не нужно, освободите его с помощью `pfree`.

Аргументы

`Relation` *`rel`*

целевое отношение

Возвращаемое значение

Имя пространства имён указанного отношения.

SPI_result_code_string

SPI_result_code_string — возвращает код ошибки в виде строки

Синтаксис

```
const char * SPI_result_code_string(int code);
```

Описание

`SPI_result_code_string` выдаёт строковое представление для кода результата, который возвращается различными функциями SPI или находится в `SPI_result`.

Аргументы

`int code`

код результата

Возвращаемое значение

Строковое представление кода результата.

45.3. Управление памятью

Postgres Pro выделяет память в *контекстах памяти* и тем самым реализует удобный способ управления выделением памяти в различных местах, с разными сроками жизни выделенной памяти. При уничтожении контекста освобождается вся выделенная в нём память. Таким образом, нет необходимости контролировать каждый отдельный объект во избежание утечек памяти; вместо этого достаточно управлять только небольшим числом контекстов. Функция `palloc` и родственные ей освобождают память из «текущего» контекста.

`SPI_connect` создаёт новый контекст памяти и делает его текущим. `SPI_finish` восстанавливает контекст, который был текущим до этого, и уничтожает контекст, созданный функцией `SPI_connect`. Эти действия обеспечивают при выходе из вашей функции на C освобождение временной памяти, выделенной внутри этой функции, во избежание утечки памяти.

Однако если ваша функция на C должна вернуть объект в выделенной памяти (как значение типа, передаваемого по ссылке), эту память нельзя выделять через `palloc`, как минимум пока установлено подключение к SPI. Если вы попытаетесь это сделать, объект будет освобождён при вызове `SPI_finish` и ваша функция не будет работать надёжно. Для решения этой проблемы выделяйте память для возвращаемого объекта, используя `SPI_palloc`. `SPI_palloc` выделяет память в «верхнем контексте исполнителя», то есть, в контексте памяти, который был текущим при вызове `SPI_connect`; именно этот контекст подходит для значения, возвращаемого из функции на C. Некоторые из вспомогательных функций, описанных в этом разделе, также возвращают объекты, созданные в верхнем контексте исполнителя.

Когда вызывается `SPI_connect`, текущим контекстом становится частный контекст функции на C, создаваемый в `SPI_connect`. Все операции выделения памяти, выполняемые функциями `palloc`, `repalloc` или служебными функциями SPI (кроме описанных в этом разделе исключений), производятся в этом контексте. Когда функция на C отключается от менеджера SPI (выполняя `SPI_finish`), текущим контекстом снова становится верхний контекст исполнителя, а вся память, выделенная в контексте этой функции, освобождается, так что использовать её дальше нельзя.

SPI_palloc

SPI_palloc — выделить память в верхнем контексте исполнителя

Синтаксис

```
void * SPI_palloc(Size size)
```

Описание

`SPI_palloc` выделяет память в верхнем контексте исполнителя.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она выдаёт ошибку.

Аргументы

`Size size`

размер выделяемой памяти, в байтах

Возвращаемое значение

указатель на выделенный блок памяти заданного размера

SPI_repalloc

SPI_repalloc — поменять блок памяти в верхнем контексте исполнителя

Синтаксис

```
void * SPI_repalloc(void * pointer, Size size)
```

Описание

`SPI_repalloc` изменяет размер блока памяти, ранее выделенного функцией `SPI_palloc`.

Эта функция теперь не отличается от простой `repalloc`. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы

`void * pointer`

указатель на существующий блок памяти, подлежащий изменению

`Size size`

размер выделяемой памяти, в байтах

Возвращаемое значение

указатель на новый блок памяти указанного размера, в который скопировано содержимое прежнего блока

SPI_pfree

SPI_pfree — освободить память в верхнем контексте исполнителя

Синтаксис

```
void SPI_pfree(void * pointer)
```

Описание

`SPI_pfree` освобождает память, ранее выделенную функцией `SPI_palloc` или `SPI_repalloc`.

Эта функция теперь не отличается от простой `pfree`. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы

`void * pointer`

указатель на существующий блок памяти, подлежащий освобождению

SPI_copytuple

SPI_copytuple — скопировать строку в верхнем контексте исполнителя

Синтаксис

```
HeapTuple SPI_copytuple(HeapTuple row)
```

Описание

`SPI_copytuple` делает копию строки в верхнем контексте исполнителя. Обычно это применяется, когда нужно вернуть изменённую строку из триггера. В функции, которая должна возвращать составной тип, нужно использовать `SPI_returntuple`.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в `SPI_result` значение `SPI_ERROR_UNCONNECTED`.

Аргументы

`HeapTuple` *`row`*

строка, подлежащая копированию

Возвращаемое значение

скопированная строка либо `NULL` в случае ошибки (`SPI_result` содержит код ошибки)

SPI_returntuple

SPI_returntuple — подготовить строку для возврата в виде Datum

Синтаксис

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

Описание

`SPI_returntuple` делает копию строки в верхнем контексте исполнителя и возвращает её в форме типа `Datum`. Чтобы выдать результат, полученный указатель остаётся только преобразовать в `Datum` функцией `PointerGetDatum`.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в `SPI_result` значение `SPI_ERROR_UNCONNECTED`.

Заметьте, что эту операцию следует применять в функциях, объявленных как возвращающие составные типы. В триггерах она не применяется; чтобы вернуть изменённую строку из триггера, используйте `SPI_copytuple`.

Аргументы

`HeapTuple` *`row`*

строка, подлежащая копированию

`TupleDesc` *`rowdesc`*

дескриптор строки (передавайте каждый раз один дескриптор для более эффективного кеширования)

Возвращаемое значение

`HeapTupleHeader`, указывающий на скопированную строку, или `NULL` в случае ошибки (`SPI_result` содержит код ошибки)

SPI_modifytuple

SPI_modifytuple — создать строку, заменяя отдельные поля в данной

Синтаксис

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

Описание

`SPI_modifytuple` создаёт новую строку, подставляя новые значения для указанных столбцов и копируя исходное содержимое остальных столбцов. Исходная строка не изменяется. Новая строка возвращается в верхнем контексте исполнителя.

Эту функцию можно использовать только когда установлено подключение к SPI. В противном случае она возвращает NULL и устанавливает в `SPI_result` значение `SPI_ERROR_UNCONNECTED`.

Аргументы

`Relation` *`rel`*

Используется только в качестве дескриптора строки. (Передача отношения вместо собственно дескриптора строки — нехорошая особенность.)

`HeapTuple` *`row`*

строка, подлежащая изменению

`int` *`ncols`*

число изменяемых столбцов

`int *` *`colnum`*

массив длины *`ncols`*, содержащий номера изменяемых столбцов (начиная с 1)

`Datum *` *`values`*

массив длины *`ncols`*, содержащий новые значения указанных столбцов

`const char *` *`nulls`*

массив длины *`ncols`*, описывающий, в каких столбцах передаётся NULL

Если в *`nulls`* передаётся `NULL`, `SPI_modifytuple` считает, что ни один из параметров не равен NULL. В противном случае элемент массива *`nulls`* должен содержать `' '`, если значение соответствующего параметра не NULL, либо `'n'`, если это значение — NULL. (В последнем случае значение, переданное в соответствующем элементе *`values`*, не учитывается.) Заметьте, что *`nulls`* — это не текстовая строка, а просто массив: ноль `'\0'` в конце не нужен.

Возвращаемое значение

новая строка с изменениями, размещённая в верхнем контексте исполнителя, или `NULL` при ошибке (`SPI_result` содержит код ошибки)

В случае ошибки в `SPI_result` устанавливается:

`SPI_ERROR_ARGUMENT`

если *`rel`* — `NULL`, либо *`row`* — `NULL`, либо *`ncols`* меньше или равно 0, либо *`colnum`* — `NULL`, либо *`values`* — `NULL`

`SPI_ERROR_NOATTRIBUTE`

> если *`colnum`* содержит недопустимый номер столбца (меньше или равен 0, либо больше числа столбцов в строке *`row`*)

`SPI_ERROR_UNCONNECTED`

> если SPI неактивен

SPI_freetuple

SPI_freetuple — освободить строку, размещённую в верхнем контексте исполнителя

Синтаксис

```
void SPI_freetuple(HeapTuple row)
```

Описание

`SPI_freetuple` освобождает строку, ранее размещённую в верхнем контексте исполнителя.

Эта функция теперь не отличается от простой `heap_freetuple`. Она сохранена только для обратной совместимости с существующим кодом.

Аргументы

`HeapTuple` *`row`*

строка, подлежащая освобождению

SPI_freetuptable

SPI_freetuptable — освободить набор строк, созданный `SPI_execute` или подобной функцией

Синтаксис

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

Описание

`SPI_freetuptable` освобождает набор строк, созданных предыдущей функцией SPI выполнения команд, например `SPI_execute`. Таким образом, при вызове этой функции в качестве аргумента часто передаётся глобальная переменная `SPI_tuptable`.

Эта функция полезна, когда функция на C, использующая SPI, должна выполнить несколько команд, но не хочет сохранять результаты предыдущих команд до завершения. Заметьте, что любые не освобождённые таким образом наборы строк будут всё равно освобождены при выполнении `SPI_finish`. Кроме того, если была запущена подтранзакция, а затем она прервалась в ходе выполнения использующей SPI функции, все наборы строк, созданные в рамках подтранзакции, будут автоматически освобождены.

Начиная с PostgreSQL версии 9.3, `SPI_freetuptable` содержит защитную логику, отфильтровывающую повторные запросы на удаление одного и того же набора строк. В предыдущих версиях повторное удаление могло приводить к сбоям.

Аргументы

`SPITupleTable * tuptable`

указатель на набор строк, который нужно освободить (если NULL, ничего не происходит)

SPI_freeplan

SPI_freeplan — освободить ранее сохранённый подготовленный оператор

Синтаксис

```
int SPI_freeplan(SPIPlanPtr plan)
```

Описание

`SPI_freeplan` освобождает подготовленный оператор, до этого выданный функцией `SPI_prepare` или сохранённый функциями `SPI_keepplan` и `SPI_saveplan`.

Аргументы

`SPIPlanPtr` *`plan`*

указатель на оператор, подлежащий освобождению

Возвращаемое значение

0 в случае успеха; `SPI_ERROR_ARGUMENT`, если *`plan`* неверный или `NULL`

45.4. Управление транзакциями

Выполнять команды управления транзакциями (в частности, `COMMIT` и `ROLLBACK`) через функции SPI, такие как `SPI_execute`, нельзя. Однако имеются отдельные интерфейсные функции, которые предназначены для управления транзакциями через SPI.

Вообще говоря, не всегда безопасно и разумно начинать и заканчивать транзакции в произвольных определяемых пользователями функциях, вызываемых из SQL, не принимая во внимание контекст их вызова. Например, завершение транзакции в середине функции, вызванной в сложном SQL-выражении внутри некоторой SQL-команды, скорее всего приведёт к странным внутренним ошибкам или сбоям. Представленные здесь интерфейсные функции прежде всего предназначены для использования реализациями процедурных языков с целью управления транзакциями в процедурах уровня SQL, вызываемых командой `CALL` (при этом учитывается её контекст). Та же логика может быть реализована в процедурах на C, использующих SPI, но подробное освещение этой темы выходит за рамки данной документации.

SPI_commit

SPI_commit, SPI_commit_and_chain — зафиксировать текущую транзакцию

Синтаксис

```
void SPI_commit(void)
```

```
void SPI_commit_and_chain(void)
```

Описание

`SPI_commit` фиксирует текущую транзакцию. Это примерно равносильно выполнению SQL-команды `COMMIT`. После того как транзакция зафиксирована, автоматически начинается новая транзакция с характеристиками по умолчанию, что позволяет сразу продолжить использование функций SPI. Если во время фиксации происходит ошибка, текущая транзакция откатывается, начинается новая, а затем уже выдаётся ошибка.

`SPI_commit_and_chain` делает то же самое, но новая транзакция получает те же характеристики, что завершённая. Эта функция подобна SQL-команде `COMMIT AND CHAIN`.

Эти функции можно выполнить, только если SPI-подключение переведено в неатомарный режим в результате вызова `SPI_connect_ext`.

SPI_rollback

SPI_rollback, SPI_rollback_and_chain — прервать текущую транзакцию

Синтаксис

```
void SPI_rollback(void)
```

```
void SPI_rollback_and_chain(void)
```

Описание

`SPI_rollback` откатывает текущую транзакцию. Это примерно равносильно выполнению SQL-команды `ROLLBACK`. После того как транзакция отменена, автоматически начинается новая транзакция с характеристиками по умолчанию, что позволяет сразу продолжить использование функций SPI.

`SPI_rollback_and_chain` делает то же самое, но новая транзакция получает те же характеристики, что завершённая. Эта функция подобна SQL-команде `ROLLBACK AND CHAIN`.

Эти функции можно выполнить, только если SPI-подключение переведено в неатомарный режим в результате вызова `SPI_connect_ext`.

SPI_start_transaction

SPI_start_transaction — устаревшая функция

Синтаксис

```
void SPI_start_transaction(void)
```

Описание

Функция `SPI_start_transaction` не делает ничего, она сохранена только для совместимости с более ранними выпусками PostgreSQL. Ранее её надо было вызывать после `SPI_commit` или `SPI_rollback`, но сейчас эти функции начинают новую транзакцию автоматически.

45.5. Видимость изменений в данных

Видимость изменений в данных, которые производятся функциями, использующими SPI, (или любыми другими функциями на C), описывается следующими правилами:

- В процессе выполнения SQL-команды любые произведённые ей изменения не видны для неё самой. Например, в команде:

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  вставляемые строки не видны в части `SELECT`.
- Изменения, произведённые командой K, видны во всех командах, запущенных после K, независимо от того, были ли эти команды запущены из K (во время выполнения K) или после завершения K.
- Команды, выполняемые через SPI внутри функции, вызванной SQL-командой (будь то обычная функция или триггер), следуют одному или другому из вышеприведённых правил в зависимости флага чтения/записи, переданного SPI. Команды, выполняемые в режиме «только чтение», следует первому правилу: они не видят изменений, произведённых вызывающей командой. Команды, выполняемые в режиме «чтение-запись», следуют второму правилу: они могут видеть все произведённые к этому времени изменения.
- Все стандартные процедурные языки устанавливают режим чтения-записи в SPI в зависимости от атрибута изменчивости функции. Команды функций `STABLE` и `IMMUTABLE` выполняются в режиме «только чтение», тогда как команды функций `VOLATILE` — в режиме «чтение-запись». Хотя авторы функций на C могут нарушить это соглашение, вряд ли это будет хорошей идеей.

В следующем разделе приводится пример, иллюстрирующий применение этих правил.

45.6. Примеры

Этот раздел содержит очень простой пример использования SPI. Функция `execq` принимает в качестве первого аргумента команду SQL, а в качестве второго — число строк, выполняет команду, вызывая `SPI_exec`, и возвращает число строк, обработанных этой командой. Более сложные примеры работы с SPI вы можете найти в модуле spi.

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(execq);

Datum
execq(PG_FUNCTION_ARGS)
{
```

```
    char *command;
    int cnt;
    int ret;
    uint64 proc;

    /* Преобразовать данный текстовый объект в строку C */
    command = text_to_cstring(PG_GETARG_TEXT_PP(0));
    cnt = PG_GETARG_INT32(1);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;

    /*
     * Если были выбраны какие-то строки, вывести их через elog(INFO).
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        SPITupleTable *tuptable = SPI_tuptable;
        TupleDesc tupdesc = tuptable->tupdesc;
        char buf[8192];
        uint64 j;

        for (j = 0; j < tuptable->numvals; j++)
        {
            HeapTuple tuple = tuptable->vals[j];
            int i;

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    PG_RETURN_INT64(proc);
}
```

Так эта функция будет объявляться после того, как она будет скомпилирована в разделяемую библиотеку (подробности в Подразделе 36.10.5):

```
CREATE FUNCTION execq(text, integer) RETURNS int8
    AS 'имя_файла'
    LANGUAGE C STRICT;
```

Демонстрация использования:

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
```

```
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- вставлено функцией execq
INFO:  EXECQ:  1    -- возвращено функцией execq и вставлено командой INSERT

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a RETURNING *', 1);
INFO:  EXECQ:  2    -- 0 + 2, затем выполнение было остановлено при достижении числа
 строк
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2

 execq
-------
     3              -- 10 — только максимальное значение, 3 — реальное число строк
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 10 FROM a', 1);
 execq
-------
     3              -- обработаны все строки; обработка не прекращается при достижении
 числа строк, потому что ничего не возвращается
(1 row)

=> SELECT * FROM a;
 x
----
  0
  1
  2
 10
 11
 12
(6 rows)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  -- 0 (нет строк в a) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
```

```
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- 1 (была одна строка в a) + 1
(2 rows)

-- Этот пример демонстрирует правило видимости изменений в данных.
-- execq вызывается дважды и каждый раз видит разное количество строк:

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1    -- результаты первого вызова execq
INFO:  EXECQ:  2
INFO:  EXECQ:  1    -- результаты второго вызова execq
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 строки * 1 (x в первой в строке)
 6                  -- 3 строки (2 + 1 только вставленная) * 2 (x во второй строке)
(4 rows)
```

Глава 46. Фоновые рабочие процессы

Postgres Pro поддерживает расширенную возможность запускать пользовательский код в отдельных процессах. Такие процессы запускаются, останавливаются и контролируются главным процессом `postgres`, который позволяет тесно связать их жизненный цикл с состоянием сервера. Эти процессы получают доступ к области разделяемой памяти Postgres Pro и могут устанавливать внутренние подключения к базам данных; они также могут последовательно запускать транзакции, как и обычные серверные процессы, обслуживающие клиентов. Кроме того, используя libpq, они могут подключаться к серверу и работать как обычные клиентские приложения.

Предупреждение

С использованием фоновых рабочих процессов сопряжены угрозы стабильности и безопасности, так как они реализуются на языке `C`, и значит имеют неограниченный доступ к данным. Администраторы, желающие использовать модули, в которых задействованы фоновые рабочие процессы, должны быть крайне осторожными. Запускать рабочие процессы можно разрешать только модулям, прошедшим всесторонний аудит.

Рабочие процессы могут инициализироваться во время запуска Postgres Pro, если имя соответствующего модуля добавлено в `shared_preload_libraries`. Модуль, желающий запустить рабочий процесс, может зарегистрировать его, вызвав `RegisterBackgroundWorker(BackgroundWorker *`*`worker`*`)` из своей функции `_PG_init()`. Рабочие процессы также могут быть запущены после запуска системы с помощью функции `RegisterDynamicBackgroundWorker(BackgroundWorker *`*`worker`*`, BackgroundWorkerHandle **`*`handle`*`)`. В отличие от `RegisterBackgroundWorker`, которую можно вызывать только из главного управляющего процесса, `RegisterDynamicBackgroundWorker` должна вызываться из обычного обслуживающего процесса или другого рабочего процесса.

Структура `BackgroundWorker` определяется так:

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    char        bgw_type[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;       /* in seconds, or BGW_NEVER_RESTART */
    char        bgw_library_name[MAXPGPATH];
    char        bgw_function_name[BGW_MAXLEN];
    Datum       bgw_main_arg;
    char        bgw_extra[BGW_EXTRALEN];
    pid_t       bgw_notify_pid;
} BackgroundWorker;
```

Поля `bgw_name` и `bgw_type` содержат строки, выводимые в отладочных сообщениях, списках процессов и подобных контекстах. Строка `bgw_type` должна быть одинаковой для всех рабочих процессов одного типа, чтобы такие процессы можно было сгруппировать, например, в списке процессов. `bgw_name`, с другой стороны, может содержать дополнительную информацию об определённом процессе. (Обычно строка `bgw_name` содержит тип в некотором виде, но строго это не требуется.)

Поле `bgw_flags` представляет битовую маску, обозначающую запрашиваемые модулем возможности. Допустимые в нём флаги:

`BGWORKER_SHMEM_ACCESS`

Запрашивает доступ к общей памяти. Этот флаг является обязательным.

`BGWORKER_BACKEND_DATABASE_CONNECTION`

> Запрашивается возможность устанавливать подключение к базе данных, через которое можно запускать транзакции и запросы. Рабочий процесс, использующий `BGWORKER_BACKEND_DATABASE_CONNECTION` для подключения к базе данных, должен также запросить доступ к разделяемой памяти, установив `BGWORKER_SHMEM_ACCESS`; в противном случае процесс не запустится.

В `bgw_start_time` определяется состояние сервера, в котором `postgres` должен запустить этот процесс; возможные варианты: `BgWorkerStart_PostmasterStart` (выполнить запуск сразу после того, как `postgres` завершит инициализацию; процессы, выбирающие такой режим, не могут подключаться к базам данных), `BgWorkerStart_ConsistentState` (выполнить запуск, когда будет достигнуто согласованное состояние горячего резерва, и когда процессы могут подключаться к базам данных и выполнять запросы на чтение), и `BgWorkerStart_RecoveryFinished` (выполнить запуск, как только система перейдёт в обычный режим чтения-записи). Заметьте, что два последних варианта различаются только для серверов горячего резерва. Заметьте также, что этот параметр указывает только, когда должны запускаться процессы; при переходе в другое состояние они не будут останавливаться.

`bgw_restart_time` задаёт паузу (в секундах), которую должен сделать `postgres`, прежде чем перезапускать процесс в случае его отказа. Это может быть любое положительное значение, либо `BGW_NEVER_RESTART`, указывающее, что процесс не нужно перезапускать в случае сбоя.

`bgw_library_name` определяет имя библиотеки, в которой следует искать точку входа для запуска рабочего процесса. Указанная библиотека будет динамически загружена рабочим процессом, а вызываемая функция будет выбрана по имени `bgw_function_name`. Для функции, вызываемой в коде ядра, в этом поле должно быть `postgres`.

`bgw_function_name` определяет имя функции, которая будет использоваться как точка входа в новый рабочий процесс. Если эта функция в динамически загружаемой библиотеке, она должна быть помечена как `PGDLLEXPORT` (а не быть помечена как `static`).

В `bgw_main_arg` задаётся аргумент `Datum`, передаваемый основной функции фонового процесса. Эта функция должна принимать один аргумент типа `Datum` и возвращать `void`. В качестве этого аргумента ей и передаётся `bgw_main_arg`. Кроме того, глобальная переменная `MyBgworkerEntry` указывает на копию структуры `BackgroundWorker`, переданной при регистрации; содержимое этой структуры может быть полезно рабочему процессу.

В Windows (и везде, где определяется `EXEC_BACKEND`) или в динамических рабочих процессах передавать `Datum` по ссылке небезопасно, возможна только передача по значению. Поэтому если функции требуется аргумент, наиболее безопасно будет передать int32 или другое небольшое значение, содержащее индекс в массиве, размещённом в разделяемой памяти. Если же попытаться передать значение `cstring` или `text`, этот указатель нельзя будет использовать в новом рабочем процессе.

Поле `bgw_extra` может содержать дополнительные данные, передаваемые фоновому рабочему процессу. В отличие от `bgw_main_arg`, эти данные не передаются в качестве аргумента основной функции рабочего процесса, но могут быть получены через `MyBgworkerEntry`, как описывалось выше.

В `bgw_notify_pid` задаётся PID обслуживающего процесса Postgres Pro, которому главный процесс должен посылать сигнал `SIGUSR1` при запуске и завершении нового рабочего процесса. Это поле должно содержать 0 для рабочих процессов, регистрируемых при запуске главного процесса, либо когда обслуживающий процесс не желает ждать окончания запуска рабочего процесса. Во всех остальных случаях в нём должно быть значение `MyProcPid`.

Запущенный процесс может подключиться к базе данных, вызвав `BackgroundWorkerInitializeConnection(`*`char *dbname`*`, `*`char *username`*`, `*`uint32 flags`*`)` или `BackgroundWorkerInitializeConnectionByOid(`*`Oid dboid`*`, `*`Oid useroid`*`, `*`uint32 flags`*`)`. Через это подключение процесс может выполнять транзакции и запросы, используя интерфейс `SPI`. Если в `dbname` передаётся NULL или `dboid` равен `InvalidOid`, сеанс не подключается ни к какой конкретной базе данных, но может обращаться к общим каталогам. Если в `username` передаётся NULL или `useroid` равен `InvalidOid`, процесс будет действовать от имени суперпользователя, созданного во время `initdb`. Значение `BGWORKER_BYPASS_ALLOWCONN` в поле `flags` позволяет подключаться к базам, не принимающим подключения пользователей. Значение `BGWORKER_BYPASS_ROLELOGINCHECK` в поле `flags` позволяет обойти проверку входа в систему для роли, используемой для подключения к базам данных. Рабочий процесс может вызвать только одну из этих двух функций и только один раз. Переключаться между базами данных он не может.

Сигналы изначально блокируются при вызове основной функции рабочего процесса и должны быть разблокированы ей: это позволяет процессу при необходимости настроить собственные обработчики событий. Новый процесс может разблокировать сигналы, вызвав `BackgroundWorkerUnblockSignals`, и заблокировать их, вызвав `BackgroundWorkerBlockSignals`.

Если `bgw_restart_time` для рабочего процесса имеет значение `BGW_NEVER_RESTART`, либо он завершается с кодом выхода 0, либо если его работа заканчивается вызовом `TerminateBackgroundWorker`, он автоматически перестаёт контролироваться управляющим процессом при выходе. В противном случае он будет перезапущен через время, заданное в `bgw_restart_time`, либо немедленно, если управляющему серверу пришлось переинициализировать кластер из-за сбоя обслуживающего процесса. Обслуживающие процессы, которым нужно только приостановить своё выполнение на время, должны переходить в состояние прерываемого ожидания, а не завершаться; для этого используется функция `WaitLatch()`. При вызове этой функции обязательно установите флаг `WL_POSTMASTER_DEATH` и проверьте код возврата, чтобы корректно выйти в экстренном случае, когда был завершён сам `postgres`.

Когда рабочий процесс регистрируется функцией `RegisterDynamicBackgroundWorker`, обслуживающий процесс, производящий эту регистрацию, может получить информацию о состоянии порождённого процесса. Обслуживающие процессы, желающие сделать это, должны передать адрес `BackgroundWorkerHandle *` во втором аргументе `RegisterDynamicBackgroundWorker`. Если рабочий процесс успешно зарегистрирован, по этому адресу будет записан указатель на скрытую структуру, который можно затем передать функции `GetBackgroundWorkerPid(`*`BackgroundWorkerHandle *, pid_t *`*`)` или `TerminateBackgroundWorker(`*`BackgroundWorkerHandle *`*`)`. Вызывая `GetBackgroundWorkerPid`, можно опрашивать состояние рабочего процесса: значение результата `BGWH_NOT_YET_STARTED` показывает, что рабочий процесс ещё не запущен управляющим; `BGWH_STOPPED` показывает, что он был запущен, но сейчас не работает; и `BGWH_STARTED` показывает, что он работает в данный момент. В последнем случае через второй аргумент также возвращается PID этого процесса. Обрабатывая вызов `TerminateBackgroundWorker`, управляющий процесс посылает `SIGTERM` рабочему процессу, если он работает, и перестаёт его контролировать сразу по его завершении.

В некоторых случаях процессу, регистрирующему рабочий процесс, может потребоваться дождаться завершения запуска этого процесса. Это можно реализовать, записав в `bgw_notify_pid` значение `MyProcPid`, а затем передав указатель `BackgroundWorkerHandle *`, полученный во время регистрации, функции `WaitForBackgroundWorkerStartup(`*`BackgroundWorkerHandle *handle, pid_t *`*`)`. Эта функция заблокирует выполнение, пока управляющий процесс не попытается запустить рабочий процесс, либо пока сам управляющий процесс не завершится. Если рабочий процесс запущен, возвращается значение `BGWH_STARTED` и по переданному адресу записывается PID. В противном случае возвращается `BGWH_STOPPED` или `BGWH_POSTMASTER_DIED`.

Процесс также может ожидать завершения рабочего процесса, вызвав функцию `WaitForBackgroundWorkerShutdown(`*`BackgroundWorkerHandle *handle`*`)` с указателем `BackgroundWorkerHandle *`, полученным при регистрации. Эта функция заблокирует выполнение, пока не завершится рабочий процесс либо управляющий процесс. При завершении рабочего процесса эта функция возвращает `BGWH_STOPPED`, а при завершении управляющего — `BGWH_POSTMASTER_DIED`.

Рабочие процессы могут передавать асинхронные уведомления, вызывая либо команду `NOTIFY` через SPI (Server Programming Interface, Интерфейс программирования сервера), либо функцию `Async_Notify()` напрямую. Такие уведомления будут отправлены в момент фиксации транзакции.

Фоновые рабочие процессы не должны регистрироваться командой `LISTEN` для получения асинхронных уведомлений, ввиду отсутствия инфрастуктуры для получения таких уведомлений рабочим процессом.

Максимальное число рабочих процессов, которые можно зарегистрировать, ограничивается значением max_worker_processes.

Глава 47. Логическое декодирование

Postgres Pro обеспечивает инфраструктуру для потоковой передачи изменений, выполняемых через SQL, внешним потребителям. Эта функциональность может быть полезна для самых разных целей, включая аудит и реализацию репликации.

Изменения передаются в потоках, связываемых со слотами логической репликации.

Формат, в котором передаются изменения, определяет используемый модуль вывода. Пример модуля вывода включён в дистрибутив Postgres Pro. Также возможно разработать и другие модули, расширяющие выбор доступных форматов, не затрагивая код ядра самого сервера. Любой модуль вывода получает на вход отдельные строки, создаваемые командой `INSERT`, и новые версии строк, которые создаёт `UPDATE`. Доступность старых версий строк для `UPDATE` и `DELETE` зависит от выбора варианта идентификации реплики (см. описание `REPLICA IDENTITY`).

Изменения могут быть получены либо по протоколу потоковой репликации (см. Раздел 53.4 и Раздел 47.3), либо через функции, вызываемые в SQL (см. Раздел 47.4). Также возможно разработать дополнительные методы для обработки данных, поступающих через слот репликации, не модифицируя код ядра сервера (см. Раздел 47.7).

47.1. Примеры логического декодирования

Следующий пример демонстрирует управление логическим декодированием на уровне SQL.

Прежде чем вы сможете использовать логическое декодирование, вы должны установить в wal_level значение `logical`, а в max_replication_slots значение, не меньшее 1. После этого вы должны подключиться к целевой базе данных (в следующем примере, это `postgres`) как суперпользователь.

```
postgres=# -- Создать слот с именем 'regression_slot', использующий модуль вывода
 'test_decoding'
postgres=# SELECT * FROM pg_create_logical_replication_slot('regression_slot',
 'test_decoding', false, true);
    slot_name    |    lsn
-----------------+-----------
 regression_slot | 0/16B1970
(1 row)

postgres=# SELECT slot_name, plugin, slot_type, database, active, restart_lsn,
 confirmed_flush_lsn FROM pg_replication_slots;
    slot_name    |    plugin     | slot_type | database | active | restart_lsn |
 confirmed_flush_lsn
-----------------+---------------+-----------+----------+--------+-------------
+-----------------
 regression_slot | test_decoding | logical   | postgres | f      | 0/16A4408   |
 0/16A4440
(1 row)

postgres=# -- Пока никакие изменения не видны
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE

postgres=# -- DDL не реплицируется, поэтому видна только транзакция
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
```

```
    lsn    |  xid  |     data
-----------+-------+--------------
 0/BA2DA58 | 10297 | BEGIN 10297
 0/BA5A5A0 | 10297 | COMMIT 10297
(2 rows)

postgres=# -- Когда изменения прочитаны, они считаются обработанными и уже не выдаются
postgres=# -- в последующем вызове:
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('1');
postgres=*# INSERT INTO data(data) VALUES('2');
postgres=*# COMMIT;

postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+---------------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)

postgres=# INSERT INTO data(data) VALUES('3');

postgres=# -- Также можно заглянуть вперёд в потоке изменений, не считывая эти
 изменения
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+---------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- Следующий вызов pg_logical_slot_peek_changes() снова возвращает те же
 изменения
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+---------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- Модулю вывода можно передать параметры, влияющие на форматирование
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL,
 'include-timestamp', 'on');
    lsn    |  xid  |                          data
-----------+-------+---------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10 12:07:21.272494-04)
(3 rows)
```

```
postgres=# -- Не забудьте удалить слот, который вам больше не нужен, чтобы он
postgres=# -- не потреблял ресурсы сервера:
postgres=# SELECT pg_drop_replication_slot('regression_slot');
 pg_drop_replication_slot
-----------------------

(1 row)
```

Следующие примеры показывают, как можно управлять логическим декодированием средствами протокола потоковой репликации, используя программу pg_recvlogical, включённую в дистрибутив Postgres Pro. Для этого нужно, чтобы конфигурация аутентификации клиентов допускала подключения для репликации (см. Подраздел 25.2.5.1) и чтобы значение `max_wal_senders` было достаточно большим и позволило установить дополнительное подключение. Второй пример демонстрирует потоковую передачу двухфазных транзакций. Прежде чем использовать двухфазные команды, для параметра max_prepared_transactions нужно установить значение не меньше 1.

```
Пример 1:
$ pg_recvlogical -d postgres --slot=test --create-slot
$ pg_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ psql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ pg_recvlogical -d postgres --slot=test --drop-slot

Пример 2:
$ pg_recvlogical -d postgres --slot=test --create-slot --two-phase
$ pg_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ psql -d postgres -c "BEGIN;INSERT INTO data(data) VALUES('5');PREPARE TRANSACTION
 'test';"
$ fg
BEGIN 694
table public.data: INSERT: id[integer]:5 data[text]:'5'
PREPARE TRANSACTION 'test', txid 694
Control+Z
$ psql -d postgres -c "COMMIT PREPARED 'test';"
$ fg
COMMIT PREPARED 'test', txid 694
Control+C
$ pg_recvlogical -d postgres --slot=test --drop-slot
```

В следующем примере показан интерфейс SQL, который можно использовать для декодирования подготовленных транзакций. Перед тем, как использовать команды двухфазной фиксации, вы должны установить значение `max_prepared_transactions` равным как минимум 1. Вы также должны при создании слота передать в параметре *`two_phase`* функции `pg_create_logical_replication_slot` значение true. Обратите внимание, вся транзакция будет транслироваться после фиксации, если она ещё не была декодирована.

```
postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('5');
postgres=*# PREPARE TRANSACTION 'test_prepared1';

postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                          data
```

```
-----------+-----+---------------------------------------------------------
 0/1689DC0 | 529 | BEGIN 529
 0/1689DC0 | 529 | table public.data: INSERT: id[integer]:3 data[text]:'5'
 0/1689FC0 | 529 | PREPARE TRANSACTION 'test_prepared1', txid 529
(3 rows)

postgres=# COMMIT PREPARED 'test_prepared1';
postgres=# select * from pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                    data
-----------+-----+--------------------------------------------
 0/168A060 | 529 | COMMIT PREPARED 'test_prepared1', txid 529
(4 row)

postgres=#-- вы также можете откатить подготовленную транзакцию
postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('6');
postgres=*# PREPARE TRANSACTION 'test_prepared2';
postgres=# select * from pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                          data
-----------+-----+---------------------------------------------------------
 0/168A180 | 530 | BEGIN 530
 0/168A1E8 | 530 | table public.data: INSERT: id[integer]:4 data[text]:'6'
 0/168A430 | 530 | PREPARE TRANSACTION 'test_prepared2', txid 530
(3 rows)

postgres=# ROLLBACK PREPARED 'test_prepared2';
postgres=# select * from pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                     data
-----------+-----+----------------------------------------------
 0/168A4B8 | 530 | ROLLBACK PREPARED 'test_prepared2', txid 530
(1 row)
```

47.2. Концепции логического декодирования

47.2.1. Логическое декодирование

Логическое декодирование — это процедура извлечения всех постоянных изменений, происходящих в таблицах базы данных, в согласованном и понятном формате, который можно интерпретировать, не имея полного представления о внутреннем состоянии базы данных.

В Postgres Pro логическое декодирование реализуется путём перевода содержимого журнала предзаписи, описывающего изменения на уровне хранения, в специальную форму уровня приложения, например, в поток кортежей или операторов SQL.

47.2.2. Слоты репликации

В контексте логической репликации слот представляет поток изменений, которые могут быть воспроизведены клиентом в том порядке, в каком они происходили на исходном сервере. Через каждый слот передаётся последовательность изменений в одной базе данных.

> **Примечание**
>
> В Postgres Pro также есть слоты потоковой репликации (см. Подраздел 25.2.5), но они используются несколько по-другому.

Слоту репликации назначается идентификатор, уникальный для всех баз данных в кластере Postgres Pro. Слоты сохраняются независимо от подключений, использующих их, и защищены от сбоев сервера.

При обычных условиях через логический слот каждое изменение передаётся только один раз. Текущая позиция в каждом слоте сохраняется только в контрольной точке, так что в случае сбоя слот мог вернуться к предыдущему LSN, вследствие чего последние изменения могут быть переданы повторно при перезапуске сервера. За исключение нежелательных эффектов от повторной обработки одного и того же сообщения отвечают клиенты логического декодирования. Клиенты могут запоминать при декодировании, какой последний LSN они уже получали, и пропускать повторяющиеся данные или (при использовании протокола репликации) запрашивать, чтобы декодирование начиналось с этого LSN, а не с позиции, выбираемой сервером. Для этого разработан механизм отслеживания репликации, о котором можно узнать подробнее в описании источников репликации.

Для одной базы данных могут существовать несколько независимых слотов. Каждый слот имеет собственное состояние, что позволяет различным потребителям получать изменения с разных позиций в потоке изменений базы данных. Для большинства приложений каждому потребителю требуется отдельный слот.

Слот логической репликации ничего не знает о состоянии получателя(ей). Возможно даже иметь несколько различных потребителей одного слота в разные моменты времени; они просто будут получать изменения с момента, когда их перестал получать предыдущий потребитель. Но в любой определённый момент получать изменения может только один потребитель.

Логический слот репликации также можно создать на сервере горячего резерва. Чтобы запретить `VACUUM` удалять требуемые строки из системных каталогов, на резервном сервере необходимо установить параметр `hot_standby_feedback`. Несмотря на это, если какие-либо требуемые строки будут удалены, слот станет аннулирован. Настоятельно рекомендуется использовать физический слот между ведущим и резервным сервером. В противном случае `hot_standby_feedback` будет работать только пока соединение активно (например, перезапуск узла может разорвать соединение). Затем ведущий сервер может удалить строки системного каталога, которые могут потребоваться для логического декодирования на резервном сервере (поскольку ведущий сервер не знает о `catalog_xmin` на резервном сервере). Существующие логические слоты в режиме ожидания также становятся недействительными, если значение `wal_level` на основном сервере становится ниже `logical`. Это происходит, как только резервный сервер обнаруживает такое изменение в потоке WAL и означает, что для отстающих процессов walsender (если таковые имеются) некоторые записи WAL до изменения параметра `wal_level` на ведущем сервере не будут декодированы.

Для создания логического слота необходима информация обо всех текущих транзакциях. На ведущем сервере эта информация доступна напрямую, а на резервном её необходимо получать от ведущего. Таким образом, при создании слота может потребоваться ожидание каких-либо действий на ведущем сервере. Если ведущий сервер простаивает, создание логического слота на резервном может занять много времени. Это время можно сократить, вызвав функцию `pg_log_standby_snapshot` на ведущем сервере.

Внимание

Слоты репликации сохраняются при сбоях сервера и ничего не знают о состоянии их потребителя. Они не дают удалять требуемые ресурсы, даже когда не используются никаким подключением. На это уходит место в хранилище, так как ни сегменты WAL, ни требуемые строки из системных каталогов нельзя будет удалить в результате `VACUUM`, пока они нужны этому слоту репликации. В особых случаях это может привести к отключению базы для предотвращения зацикливания идентификаторов транзакций (см. Подраздел 23.1.5). Поэтому, если слот больше не требуется, его следует ликвидировать.

47.2.3. Синхронизация слотов репликации

Слоты логической репликации на главном сервере можно синхронизировать с горячим резервом следующим образом: сначала нужно создать слот, либо вызвав функцию `pg_create_logical_replication_slot` с параметром `failover`, либо указав параметр `failover` в команде `CREATE SUBSCRIPTION`. Кроме того, необходимо вызвать на резервном сервере функцию `sync_replication_slots`. Если задать на резервном сервере `sync_replication_slots`, слоты отработки отказа будут периодически синхронизироваться рабочим процессом slotsync. Чтобы синхронизация работала корректно, необходим слот физической репликации между главным и резервным сервером (например, `primary_slot_name` необходимо настроить на резервном), а также необходимо включить на резервном `hot_standby_feedback`. Также необходимо указать действительное значение `dbname` в `primary_conninfo`. Крайне рекомендуется внести слот физической репликации в список `synchronized_standby_slots` на главном сервере, чтобы подписки не получали изменения раньше горячего резерва. Даже с правильной настройкой будет заметна некоторая задержка при отправке изменений логическим подписчикам из-за ожидания слотов, перечисленных в списке `synchronized_standby_slots`. При использовании `synchronized_standby_slots` главный сервер не отключится полностью, пока резервные серверы, связанные с перечисленными там слотами, не подтвердят получение последних обновлений WAL.

Примечание

Хотя `sync_replication_slots` включает автоматическую периодическую синхронизацию слотов аварийного переключения, их также можно синхронизировать вручную на резервном сервере с помощью функции `pg_sync_replication_slots`. Однако данная функциональность в первую очередь разработана в целях тестирования и отладки, а потому должна применяться с осторожностью. В отличие от автоматической синхронизации, она не включает в себя циклические повторные попытки, что с большей вероятностью приведёт к ошибкам, в особенности во время начальной синхронизации, при которой необходимые файлы WAL или строки каталога для слота уже могли быть перемещены или вскоре будут перемещены с резервного сервера. И, напротив, автоматическая синхронизация через `sync_replication_slots` позволяет постоянно обновлять слоты, что обеспечивает бесшовное переключение и отказоустойчивость, а потому является предпочтительным способом синхронизации слотов.

При рекомендованной конфигурации синхронизации слотов, когда первоначальная синхронизация выполняется автоматически или вручную через `pg_sync_replication_slot`, на резервном сервере синхронизированный слот будет поддерживаться только при следующих условиях: слот логической репликации на ведущем сервере должен сохранять WAL и строки системного каталога, которые всё ещё доступны на резервном. Это обеспечит целостность данных и бесперебойную работу логической репликации после переключения. Если с резервного сервера уже были удалены необходимые WAL или строки системного каталога, слот не будет поддерживаться во избежание потери данных. В таких случаях в журнале появится следующее сообщение:

```
LOG:  could not synchronize replication slot "failover_slot" (не удалось
 синхронизировать слот репликации "failover_slot")
DETAIL:  Synchronization could lead to data loss, because the remote slot needs WAL
 at LSN 0/3003F28 and catalog xmin 754, but the standby has LSN 0/3003F28 and catalog
 xmin 756 (Синхронизация может привести к потере данных, поскольку удалённый слот
 запрашивает запись WAL от LSN 0/3003F28 и каталог xmin 754, но у резервного сервера
 уже есть запись WAL от LSN 0/3003F28 и каталог xmin 754).
```

Если слот логической репликации активно используется потребителем, ручное вмешательство не требуется, поскольку слот автоматически продвинется дальше, а синхронизация возобновится в ходе следующего цикла. Однако при отсутствии настроенного потребителя рекомендуется вручную продвинуть слот на ведущий сервер через `pg_logical_slot_get_changes` или `pg_logical_slot_get_binary_changes`, что продолжит синхронизацию.

Возможность возобновить процесс логической репликации после отказа зависит от значения pg_replication_slots.`synced` для слотов, синхронизированных на резервном сервере на момент отработки отказа. Для логической репликации после отказа могут использоваться только постоянные слоты, которые были синхронизированы на резервном сервере до отказа. Временные синхронизированные слоты для этого использовать нельзя, то есть возобновить логическую репликацию для таких слотов невозможно. Например, если синхронизированный слот не может стать постоянным на резервном из-за того, что подписка отключена, то подписка не будет возобновлена после отказа, даже если её включить.

Чтобы возобновить логическую репликацию из синхронизированных логических слотов после отработки отказа, необходимо в строке conninfo подписки указать главный сервер. Это можно сделать командой `ALTER SUBSCRIPTION ... CONNECTION`. Рекомендуется отключить подписки до повышения резерва и включить после изменения строки подключения.

> **Внимание**
>
> Всегда есть вероятность, что старый главный сервер снова включится во время повышения, и если подписки включены, то подписчики логической репликации могут продолжить получать данные с прошлого главного сервера после повышения резерва и вплоть до момента, пока не изменится строка подключения. Это может вызвать конфликты в данных, и подписчики не смогут продолжить репликацию с нового главного сервера.

47.2.4. Модули вывода

Модули вывода переводят данные из внутреннего представления в журнале предзаписи в формат, устраивающий потребителя слота репликации.

47.2.5. Экспортированные снимки

Когда новый слот репликации создаётся через интерфейс потоковой репликации, экспортируется снимок (см. CREATE_REPLICATION_SLOT), который будет показывать ровно то состояние базы данных, изменения после которого будут включаться в поток изменений. Используя его, можно создать новую реплику, воспользовавшись командой `SET TRANSACTION SNAPSHOT`, чтобы получить состояние базы в момент создания слота. После этого данную транзакцию можно использовать для выгрузки состояния базы на момент экспорта снимка, а затем изменять это состояние, применяя содержимое слота, так что никакие изменения не будут потеряны.

Приложения, которым не требуется экспорт снимка, могут подавить его, воспользовавшись указанием `SNAPSHOT 'nothing'`.

47.3. Интерфейс протокола потоковой репликации

Команды

- `CREATE_REPLICATION_SLOT` *`имя_слота`* `LOGICAL` *`модуль_вывода`*
- `DROP_REPLICATION_SLOT` *`имя_слота`* [`WAIT`]
- `START_REPLICATION SLOT` *`имя_слота`* `LOGICAL ...`

применяются для создания, удаления и передачи изменений из слота репликации, соответственно. Эти команды доступны только для соединения репликации; их нельзя использовать в обычном SQL. Подробнее они описаны в Разделе 53.4.

Для управления логическим декодированием по соединению потоковой репликации можно применять программу pg_recvlogical. (Внутри неё используются эти команды.)

47.4. Интерфейс логического декодирования на уровне SQL

Подробнее API уровня SQL для взаимодействия с механизмом логическим декодированием описан в Подразделе 9.28.6.

Синхронная репликация (см. Подраздел 25.2.8) поддерживается только для слотов репликации, которые используются через интерфейс потоковой репликации. Интерфейс функций и дополнительные, не системные интерфейсы не поддерживают синхронную репликацию.

47.5. Системные каталоги, связанные с логическим декодированием

Информацию о текущем состоянии слотов репликации и соединений потоковой репликации отображают представления `pg_replication_slots` и `pg_stat_replication`, соответственно. Эти представления относятся и к физической, и к логической репликации. Представление `pg_stat_replication_slots` показывает статистическую информацию о логических слотах репликации.

47.6. Модули вывода логического декодирования

Пример модуля вывода можно найти в подкаталоге `contrib/test_decoding` в дереве исходного кода Postgres Pro.

47.6.1. Функция инициализации

Модуль вывода загружается в результате динамической загрузки разделяемой библиотеки (при этом в качестве имени библиотеки задаётся имя модуля). Для нахождения библиотеки применяется обычный путь поиска библиотек. В этой библиотеке должна быть функция `_PG_output_plugin_init`, которая показывает, что библиотека на самом деле представляет собой модуль вывода, и устанавливает требуемые обработчики модуля вывода. Этой функции передаётся структура, в которой должны быть заполнены указатели на функции-обработчики отдельных действий.

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeTruncateCB truncate_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
    LogicalDecodeFilterPrepareCB filter_prepare_cb;
    LogicalDecodeBeginPrepareCB begin_prepare_cb;
    LogicalDecodePrepareCB prepare_cb;
    LogicalDecodeCommitPreparedCB commit_prepared_cb;
    LogicalDecodeRollbackPreparedCB rollback_prepared_cb;
    LogicalDecodeStreamStartCB stream_start_cb;
    LogicalDecodeStreamStopCB stream_stop_cb;
    LogicalDecodeStreamAbortCB stream_abort_cb;
    LogicalDecodeStreamPrepareCB stream_prepare_cb;
    LogicalDecodeStreamCommitCB stream_commit_cb;
    LogicalDecodeStreamChangeCB stream_change_cb;
    LogicalDecodeStreamMessageCB stream_message_cb;
    LogicalDecodeStreamTruncateCB stream_truncate_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct OutputPluginCallbacks *cb);
```

Обработчики `begin_cb`, `change_cb` и `commit_cb` должны устанавливаться обязательно, а `startup_cb`, `truncate_cb`, `message_cb`, `filter_by_origin_cb` и `shutdown_cb` могут отсутствовать. Если `truncate_cb` не установлен, но потребуется декодировать операцию `TRUNCATE`, она будет проигнорирована.

Модуль вывода может также определять функции для поддержки потоковой передачи больших транзакций во время их выполнения. Функции `stream_start_cb`, `stream_stop_cb`, `stream_abort_cb`, `stream_commit_cb` и `stream_change_cb` являются обязательными, а `stream_message_cb` и `stream_truncate_cb` — необязательными. Функция `stream_prepare_cb` также является обязательной, если модуль вывода поддерживает двухфазную фиксацию.

Модуль вывода также может определять функции для поддержки двухфазной фиксации, позволяющие декодировать действия в `PREPARE TRANSACTION`. Обработчики `begin_prepare_cb`, `prepare_cb`, `commit_prepared_cb` и `rollback_prepared_cb` являются обязательными, а `filter_prepare_cb` — нет. Обработчик `stream_prepare_cb` также является обязательным, если модуль вывода поддерживает потоковую передачу больших выполняющихся транзакций.

47.6.2. Возможности

Для декодирования, форматирования и вывода изменений модули вывода могут использовать практически всю обычную инфраструктуру сервера, включая вызов функций вывода типов. К отношениям разрешается доступ только на чтение, если только эти отношения были созданы программой `initdb` в схеме `pg_catalog`, либо помечены как пользовательские таблицы каталогов командами

```
ALTER TABLE user_catalog_table SET (user_catalog_table = true);
CREATE TABLE another_catalog_table(data text) WITH (user_catalog_table = true);
```

Обратите внимание, что доступ к пользовательским таблицам каталога или обычным системным таблицам каталога в подключаемых модулях вывода должен осуществляться только с использованием функций сканирования `systable_*`. При попытке использовать функции сканирования `heap_*` произойдёт ошибка. Кроме того, любые действия, которые требуют присваивания идентификатора транзакции, запрещаются. В частности, к этим действиям относятся операции записи в таблицы, изменения DDL и вызов `pg_current_xact_id()`.

47.6.3. Режимы вывода

Обработчики в модуле вывода могут передавать данные потребителю в практически любых форматах. Для некоторых вариантов использования, например, просмотра изменений через SQL, вывод информации в типах, которые могут содержать произвольные данные (например, `bytea`), может быть неудобоваримым. Если модуль вывода выводит только текстовые данные в кодировке сервера, он может объявить это, установив в `OutputPluginOptions.output_type` значение `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` вместо `OUTPUT_PLUGIN_BINARY_OUTPUT` в обработчике запуска. В этом случае все данные должны быть в кодировке сервера, чтобы их можно было передать в значении типа `text`. Это контролируется в сборках с включёнными проверочными утверждениями.

47.6.4. Обработчики в модуле вывода

Модуль вывода уведомляется о происходящих изменениях через различные обработчики, которые он должен установить.

Параллельные транзакции декодируются в порядке фиксации, при этом между обратными вызовами `begin` и `commit` декодируются только изменения, относящиеся к определённой транзакции. Явно или неявно отменённые транзакции никогда не декодируются. Успешные точки сохранения включаются в содержащую их транзакцию в том порядке, в котором они выполнялись в этой транзакции. Транзакция, подготовленная для двухфазной фиксации командой `PREPARE TRANSACTION`, также будет декодирована, если модуль вывода предоставил требуемые обработчики для декодирования. Возможно, что текущая подготовленная декодируемая транзакция будет прервана параллельно выполненной командой `ROLLBACK PREPARED`. В этом случае будет прервано и логическое декодирование этой транзакции. Все изменения такой транзакции пропускаются после обнаружения прерывания и вызова обработчика `prepare_cb`. Таким образом даже при блокирующем прерывании модулю вывода предоставляется достаточно информации, чтобы он мог правильно обработать `ROLLBACK PREPARED` после декодирования.

Примечание

Декодироваться будут только те транзакции, которые уже успешно сброшены на диск. Вследствие этого, `COMMIT` может не декодироваться в следующем сразу за ним вызове `pg_logical_slot_get_changes()`, когда `synchronous_commit` имеет значение `off`.

47.6.4.1. Обработчик запуска

Необязательный обработчик `startup_cb` вызывается, когда слот репликации создаётся или через него запрашивается передача изменений, независимо от того, в каком количестве изменения готовы к передаче.

```
typedef void (*LogicalDecodeStartupCB) (struct LogicalDecodingContext *ctx,
                                        OutputPluginOptions *options,
                                        bool is_init);
```

Параметр `is_init` будет равен true, когда слот репликации создаётся, и false в противном случае. Параметр *`options`* указывает на структуру параметров, которые могут устанавливать модули вывода:

```
typedef struct OutputPluginOptions
{
    OutputPluginOutputType output_type;
    bool            receive_rewrites;
} OutputPluginOptions;
```

В поле `output_type` должно быть значение `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` или `OUTPUT_PLUGIN_BINARY_OUTPUT`. См. также Подраздел 47.6.3. Если поле `receive_rewrites` равно true, модуль вывода также будет вызываться для изменений, связанных с перезаписью кучи при определённых операциях DDL. Эти изменения представляют интерес для модулей, осуществляющих репликацию DDL, но для их обработки может потребоваться особый подход.

Обработчик запуска должен проверить параметры, представленные в `ctx->output_plugin_options`. Если модулю вывода требуется поддерживать состояние, он может сохранить его в `ctx->output_plugin_private`.

47.6.4.2. Обработчик выключения

Необязательный обработчик `shutdown_cb` вызывается, когда ранее активный слот репликации перестаёт использоваться, так что ресурсы, занятые модулем вывода, можно освободить. При этом слот не обязательно удаляется, прекращается только потоковая передача через него.

```
typedef void (*LogicalDecodeShutdownCB) (struct LogicalDecodingContext *ctx);
```

47.6.4.3. Обработчик начала транзакции

Обязательный обработчик `begin_cb` вызывается, когда декодируется начало зафиксированной транзакции. Прерванные транзакции и их содержимое никогда не декодируется.

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext *ctx,
                                      ReorderBufferTXN *txn);
```

Параметр *`txn`* содержит метаинформацию о транзакции, в частности её идентификатор и время её фиксирования.

47.6.4.4. Обработчик завершения транзакции

Обязательный обработчик `commit_cb` вызывается, когда декодируется фиксирование транзакции. Перед этим обработчиком будет вызываться обработчик `change_cb` для всех изменённых строк (если строки были изменены).

```
typedef void (*LogicalDecodeCommitCB) (struct LogicalDecodingContext *ctx,
```

```
                                              ReorderBufferTXN *txn,
                                              XLogRecPtr commit_lsn);
```

47.6.4.5. Обработчик изменения

Обязательный обработчик `change_cb` вызывается для каждого отдельного изменения строки в транзакции, производимого командами `INSERT`, `UPDATE` или `DELETE`. Даже если команда изменила несколько строк сразу, этот обработчик будет вызываться для каждой отдельной строки. Обработчик `change_cb` может обращаться к системным или пользовательским таблицам каталога, чтобы дополнить выводимую информацию об изменении строки. В случае декодирования подготовленной (но ещё не зафиксированной) транзакции или декодирования незафиксированной транзакции этот обработчик изменений также может выдать ошибку из-за одновременного отката той же самой транзакции. В этом случае логическое декодирование этой прерванной транзакции корректно останавливается.

```
typedef void (*LogicalDecodeChangeCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       Relation relation,
                                       ReorderBufferChange *change);
```

Параметры *`ctx`* и *`txn`* имеют то же содержимое, что и для обработчиков `begin_cb` и `commit_cb`; дополнительный дескриптор отношения *`relation`* указывает на отношение, к которому принадлежит строка, а структура *`change`* описывает передаваемое изменение строки.

Примечание

В процессе логического декодирования могут быть обработаны изменения только в таблицах, не являющихся нежурналируемыми (см. описание `UNLOGGED`) или временными (см. описание `TEMPORARY` **или** `TEMP`).

47.6.4.6. Обработчик опустошения

Необязательный обработчик `truncate_cb` вызывается для команды `TRUNCATE`.

```
typedef void (*LogicalDecodeTruncateCB) (struct LogicalDecodingContext *ctx,
                                         ReorderBufferTXN *txn,
                                         int nrelations,
                                         Relation relations[],
                                         ReorderBufferChange *change);
```

Он получает те же параметры, что и `change_cb`. Но так как операции `TRUNCATE` в таблицах, связанных внешними ключами, должны выполняться одновременно, данный обработчик получает на вход не одно отношение, а массив отношений. За подробностями обратитесь к описанию оператора TRUNCATE.

47.6.4.7. Обработчик фильтрации источника

Необязательный обработчик `filter_by_origin_cb` вызывается, чтобы отметить, интересуют ли модуль вывода изменения, воспроизводимые из указанного источника (*`origin_id`*).

```
typedef bool (*LogicalDecodeFilterByOriginCB) (struct LogicalDecodingContext *ctx,
                                               RepOriginId origin_id);
```

В параметре *`ctx`* передаётся та же информация, что и для других обработчиков. Чтобы отметить, что изменения, поступающие из переданного узла, не представляют интереса, модуль должен вернуть true, вследствие чего эти изменения будут фильтроваться; в противном случае он должен вернуть false. Другие обработчики для фильтруемых транзакций и изменений вызываться не будут.

Это полезно при реализации каскадной или разнонаправленной репликации. Фильтрация по источнику в таких конфигурациях позволяет предотвратить передачу взад-вперёд одних и тех же изменений. Хотя информацию об источнике можно также извлечь из транзакций и изменений, фильтрация с помощью этого обработчика гораздо более эффективна.

47.6.4.8. Обработчик произвольных сообщений

Необязательный обработчик `message_cb` вызывается при получении сообщения логического декодирования.

```
typedef void (*LogicalDecodeMessageCB) (struct LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        XLogRecPtr message_lsn,
                                        bool transactional,
                                        const char *prefix,
                                        Size message_size,
                                        const char *message);
```

Параметр *`txn`* содержит метаинформацию о транзакции, включая время её фиксации и её XID. Заметьте, однако, что в нём может передаваться NULL, когда сообщение нетранзакционное и транзакции, в которой было выдано сообщение, ещё не назначен XID. В параметре *`lsn`* отмечается позиция сообщения в WAL. Параметр *`transactional`* показывает, было ли сообщение передано как транзакционное. Подобно обработчику изменений, в случае декодирования подготовленной (но ещё не зафиксированной) транзакции или декодирования незафиксированной транзакции этот обработчик также может выдать ошибку из-за одновременного отката той же самой транзакции. В этом случае логическое декодирование этой прерванной транзакции корректно останавливается. В параметре *`prefix`* передаётся некоторый префикс (завершающийся нулём), по которому текущий модуль может выделять интересующие его сообщения. И наконец, параметр *`message`* содержит само сообщение размером *`message_size`* байт.

Необходимо дополнительно позаботиться о том, чтобы префикс, определяющий интересующие модуль вывода сообщения, был уникальным. Удачным выбором обычно будет имя расширения или самого модуля вывода.

47.6.4.9. Обработчик фильтра подготовки

Необязательный обработчик `filter_prepare_cb` вызывается, чтобы определить, следует ли декодировать данные, составляющие текущую фиксируемую двухфазную транзакцию, на этапе подготовки или позже — в виде обычной однофазной транзакции на этапе `COMMIT PREPARED`. Если декодирование следует пропустить, обработчик должен выдать `true`, а иначе — `false`. Если обработчик не определён, предполагается `false` (т. е. фильтрация отсутствует — все транзакции, использующие двухфазную фиксацию, декодируются также в две фазы).

```
typedef bool (*LogicalDecodeFilterPrepareCB) (struct LogicalDecodingContext *ctx,
                                              TransactionId xid,
                                              const char *gid);
```

Параметр *`ctx`* имеет то же содержимое, что и в других обработчиках. Параметры *`xid`* и *`gid`* предоставляют два разных способа идентификации транзакции. Последующая команда `COMMIT PREPARED` или `ROLLBACK PREPARED` передаёт оба идентификатора, предоставляя модулю вывода возможность выбора, какой использовать.

Обработчик может вызываться несколько раз в течение одной декодируемой транзакции и должен выдавать один и тот же постоянный ответ для данной пары *`xid`* и *`gid`* при каждом вызове.

47.6.4.10. Обработчик начала подготовленной транзакции

Обязательный обработчик `begin_prepare_cb` вызывается при декодировании начала подготовленной транзакции. Переданное в параметре *`txn`* поле *`gid`* позволяет обработчику определить, получал ли этот модуль уже данную команду `PREPARE`, — если да, он может либо выдать ошибку, либо пропустить оставшиеся изменения транзакции.

```
typedef void (*LogicalDecodeBeginPrepareCB) (struct LogicalDecodingContext *ctx,
                                             ReorderBufferTXN *txn);
```

47.6.4.11. Обработчик подготовки транзакции

Обязательный обработчик `prepare_cb` вызывается при декодировании транзакции, подготовленной для двухфазной фиксации. Перед этим обработчиком будет вызываться обработчик `change_cb` для всех изменённых строк (если строки были изменены). В этом обработчике может использоваться поле *gid*, являющееся частью параметра *txn*.

```
typedef void (*LogicalDecodePrepareCB) (struct LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        XLogRecPtr prepare_lsn);
```

47.6.4.12. Обработчик фиксации подготовленной транзакции

Обязательный обработчик `commit_prepared_cb` вызывается при декодировании команды `COMMIT PREPARED` транзакции. В этом обработчике может использоваться поле *gid*, являющееся частью параметра *txn*.

```
typedef void (*LogicalDecodeCommitPreparedCB) (struct LogicalDecodingContext *ctx,
                                               ReorderBufferTXN *txn,
                                               XLogRecPtr commit_lsn);
```

47.6.4.13. Обработчик отмены подготовленной транзакции

Обязательный обработчик `rollback_prepared_cb` вызывается при декодировании команды `ROLLBACK PREPARED` транзакции. В этом обработчике может использоваться поле *gid*, которое является частью параметра *txn*. Проанализировав параметры *prepare_end_lsn* и *prepare_time*, модуль декодирования может определить, получал ли он `PREPARE TRANSACTION`, и если да — выполнить отмену транзакции, а в противном случае пропустить эту операцию. Одного поля *gid* для этого недостаточно, так как на нижестоящем узле может быть подготовленная транзакция с таким же строковым идентификатором.

```
typedef void (*LogicalDecodeRollbackPreparedCB) (struct LogicalDecodingContext *ctx,
                                                 ReorderBufferTXN *txn,
                                                 XLogRecPtr prepare_end_lsn,
                                                 TimestampTz prepare_time);
```

47.6.4.14. Обработчик начала потока

Обязательный обработчик `stream_start_cb` вызывается, когда открывается блок передаваемых в потоке изменений выполняющейся транзакции.

```
typedef void (*LogicalDecodeStreamStartCB) (struct LogicalDecodingContext *ctx,
                                            ReorderBufferTXN *txn);
```

47.6.4.15. Обработчик остановки потока

Обязательный обработчик `stream_stop_cb` вызывается, когда закрывается блок передаваемых в потоке изменений выполняющейся транзакции.

```
typedef void (*LogicalDecodeStreamStopCB) (struct LogicalDecodingContext *ctx,
                                           ReorderBufferTXN *txn);
```

47.6.4.16. Обработчик прерывания потока

Обязательный обработчик `stream_abort_cb` вызывается для прерывания передаваемой в потоке транзакции.

```
typedef void (*LogicalDecodeStreamAbortCB) (struct LogicalDecodingContext *ctx,
                                            ReorderBufferTXN *txn,
                                            XLogRecPtr abort_lsn);
```

47.6.4.17. Обработчик подготовки потока

Обработчик `stream_prepare_cb` вызывается для подготовки ранее переданной в потоке транзакции как части двухфазной фиксации. Этот обработчик требуется, когда плагин вывода поддерживает как потоковую передачу больших текущих транзакций, так и двухфазные фиксации.

```
typedef void (*LogicalDecodeStreamPrepareCB) (struct LogicalDecodingContext *ctx,
                                               ReorderBufferTXN *txn,
                                               XLogRecPtr prepare_lsn);
```

47.6.4.18. Обработчик фиксации потока

Обязательный обработчик `stream_commit_cb` вызывается для фиксации ранее переданной в потоке транзакции.

```
typedef void (*LogicalDecodeStreamCommitCB) (struct LogicalDecodingContext *ctx,
                                              ReorderBufferTXN *txn,
                                              XLogRecPtr commit_lsn);
```

47.6.4.19. Обработчик изменений в потоке

Обязательный обработчик `stream_change_cb` вызывается, когда передаётся изменение в блоке изменений (границы которого обозначают вызовы `stream_start_cb` и `stream_stop_cb`). Фактические изменения не отображаются, поскольку транзакция может быть прервана позже, а изменения для прерванных транзакций не декодируются.

```
typedef void (*LogicalDecodeStreamChangeCB) (struct LogicalDecodingContext *ctx,
                                              ReorderBufferTXN *txn,
                                              Relation relation,
                                              ReorderBufferChange *change);
```

47.6.4.20. Обработчик сообщения потока

Необязательный обработчик `stream_message_cb` вызывается, когда передаётся произвольное сообщение в блоке изменений (границы которого обозначают вызовы `stream_start_cb` и `stream_stop_cb`). Содержимое транзакционных сообщений не отображается, поскольку транзакция может быть прервана позже, а изменения для прерванных транзакций не декодируются.

```
typedef void (*LogicalDecodeStreamMessageCB) (struct LogicalDecodingContext *ctx,
                                               ReorderBufferTXN *txn,
                                               XLogRecPtr message_lsn,
                                               bool transactional,
                                               const char *prefix,
                                               Size message_size,
                                               const char *message);
```

47.6.4.21. Обработчик команды опустошения в потоке

Необязательный обработчик `truncate_cb` вызывается для команды `TRUNCATE` в блоке изменений (границы которого обозначают вызовы `stream_start_cb` и `stream_stop_cb`).

```
typedef void (*LogicalDecodeStreamTruncateCB) (struct LogicalDecodingContext *ctx,
                                                ReorderBufferTXN *txn,
                                                int nrelations,
                                                Relation relations[],
                                                ReorderBufferChange *change);
```

Параметры этого обработчика аналогичны параметрам `change_cb`. Но так как операции `TRUNCATE` в таблицах, связанных внешними ключами, должны выполняться одновременно, данный обработчик получает на вход не одно отношение, а массив отношений. За подробностями обратитесь к описанию оператора TRUNCATE.

47.6.5. Функции для формирования вывода

Чтобы действительно вывести данные, модули вывода могут записывать их в буфер `StringInfo` через `ctx->out`, внутри обработчиков `begin_cb`, `commit_cb` или `change_cb`. Прежде чем записывать данные в этот буфер, необходимо вызвать `OutputPluginPrepareWrite(ctx, last_write)`, а завершив запись в буфер, нужно вызвать `OutputPluginWrite(ctx, last_write)`, чтобы собственно произвести запись. Параметр *`last_write`* указывает, была ли эта определённая операция записи последней в данном обработчике.

Следующий пример показывает, как вывести данные для потребителя модуля вывода:

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
OutputPluginWrite(ctx, true);
```

47.7. Запись вывода логического декодирования

Архитектура сервера позволяет добавлять другие методы вывода для логического декодирования. По сути необходимо реализовать три функции: одну для чтения WAL, другую для подготовки к записи и третью для записи вывода (см. Подраздел 47.6.5).

47.8. Поддержка синхронной репликации для логического декодирования

47.8.1. Обзор

Логическое декодирование может использоваться для реализации синхронной репликации с тем же внешним интерфейсом, что и синхронная репликация поверх потоковой репликации. Для этого потоковая передача данных должна происходить через интерфейс потоковой репликации (см. Раздел 47.3). Клиенты такой репликации должны посылать сообщения `Обновление состояния резервного сервера (F)` (см. Раздел 53.4), как и клиенты потоковой репликации.

> **Примечание**
>
> Синхронная реплика, получающая изменения через логическое декодирование, будет работать в рамках одной базы данных. Так как *synchronous_standby_names* в настоящее время, напротив, устанавливается на уровне сервера, это означает, что этот подход не будет работать корректно при использовании нескольких баз данных.

47.8.2. Ограничения

В схеме с логической репликацией возможна взаимоблокировка, если транзакция в исключительном режиме заблокирует таблицы каталога (в том числе пользовательские). Пользовательские таблицы каталога описаны в Подразделе 47.6.2. Это происходит, потому что в процессе логического декодирования транзакций при обращении к таблицам каталога они блокируются. Во избежание этого пользователи должны воздерживаться от исключительной блокировки таблиц каталога (в том числе пользовательских). Вызвать такую блокировку могут следующие обстоятельства:

- Установление явной блокировки (командой `LOCK`) для каталога `pg_class` в транзакции.
- Выполнение `CLUSTER` для каталога `pg_class` в транзакции.
- Выполнение `PREPARE TRANSACTION` после команды `LOCK`, блокирующей `pg_class`, при работающем логическом декодировании двухфазных транзакций.
- Выполнение `PREPARE TRANSACTION` после команды `CLUSTER` для `pg_trigger` при работающем логическом декодировании двухфазных транзакций. Это приведёт к взаимоблокировке только в том случае, если у публикуемой таблицы есть триггер.
- Выполнение `TRUNCATE` для таблицы каталога (в том числе пользовательской) в транзакции.

Заметьте, что эти команды способны вызвать взаимоблокировку не только для указанных выше таблиц системного каталога, но и для любых других таблиц каталога.

47.9. Передача больших транзакций для логического декодирования

Базовые обработчики модуля вывода (например, `begin_cb`, `change_cb`, `commit_cb` и `message_cb`) вызываются, только когда транзакция на самом деле фиксируется. Изменения в любом случае декодируются из журнала транзакций, но передаются в модуль вывода только при фиксации (и отбрасываются, если транзакция прерывается).

Это означает, что, хотя декодирование происходит последовательно и декодируемые записи могут вытесняться на диск, чтобы уменьшить нагрузку на оперативную память, все декодированные изменения должны быть переданы, когда транзакция окончательно фиксируется (или, точнее, когда фиксация декодируется из журнала транзакций). В зависимости от размера транзакции и пропускной способности сети время передачи может значительно увеличить задержку применения.

Чтобы уменьшить задержку, связанную с большими транзакциями, модуль вывода может предоставлять дополнительный обработчик для поддержки инкрементальной передачи выполняющихся транзакций. Определены несколько обязательных обработчиков передачи (`stream_start_cb`, `stream_stop_cb`, `stream_abort_cb`, `stream_commit_cb` и `stream_change_cb`) и два необязательных обработчика (`stream_message_cb`) и (`stream_truncate_cb`). Кроме того, если планируется поддерживать потоковую передачу двухфазных команд, нужны дополнительные обработчики. (За подробностями обратитесь к Разделу 47.10).

Когда выполняющаяся транзакция передаётся в потоке, её изменения (и сообщения) передаются блоками, границы которых обозначают вызовы `stream_start_cb` и `stream_stop_cb`. После того, как все декодированные изменения переданы, транзакция может быть зафиксирована обработчиком `stream_commit_cb` (или, возможно, прервана обработчиком `stream_abort_cb`). Если поддерживаются двухфазные фиксации, транзакция может быть подготовлена обработчиком `stream_prepare_cb`, зафиксирована (`COMMIT PREPARED`) обработчиком `commit_prepared_cb` или прервана обработчиком `rollback_prepared_cb`.

Например, последовательность вызовов обработчиков потока для одной транзакции может быть такой:

```
stream_start_cb(...);   <-- начало первого блока изменений
  stream_change_cb(...);
  stream_change_cb(...);
  stream_message_cb(...);
  stream_change_cb(...);
  ...
  stream_change_cb(...);
stream_stop_cb(...);    <-- конец первого блока изменений

stream_start_cb(...);   <-- начало второго блока изменений
  stream_change_cb(...);
  stream_change_cb(...);
  stream_change_cb(...);
  ...
  stream_message_cb(...);
  stream_change_cb(...);
stream_stop_cb(...);    <-- конец второго блока изменений


[a. при использовании нормальной фиксации]
stream_commit_cb(...);  <-- фиксация переданной транзакции

[b. при использовании двухфазной фиксации]
stream_prepare_cb(...);   <-- подготовка переданной транзакции
commit_prepared_cb(...);  <-- фиксация подготовленной транзакции
```

На практике же последовательность вызовов обработчика может быть более сложной. В ней могут быть блоки нескольких передаваемых в потоке транзакций, какие-то транзакции могут прерываться и т. д.

Подобно поведению вытеснения на диск, передача в потоке запускается, когда общее количество изменений, декодированных из WAL (для всех текущих транзакций), превышает предел, заданный параметром `logical_decoding_work_mem`. В этот момент выбирается самая большая транзакция (это определяется по объёму памяти, который уже занимают декодированные изменения) верхнего уровня, и она передаётся в потоке. Тем не менее, в некоторых случаях всё равно приходится переносить данные на диск, даже когда включена передача в потоке, поскольку порог памяти уже превышен, но полный кортеж ещё не декодирован, например декодировано добавление только в таблицу TOAST, но не в основную таблицу.

Даже при передаче в потоке больших транзакций изменения по-прежнему применяются в порядке фиксации, сохраняя те же гарантии, что и не в потоковом режиме.

47.10. Поддержка двухфазной фиксации для логического декодирования

С базовыми обработчиками модуля вывода (например, `begin_cb`, `change_cb`, `commit_cb` и `message_cb`) такие команды двухфазной фиксации, как `PREPARE TRANSACTION`, `COMMIT PREPARED` и `ROLLBACK PREPARED`, не декодируются. При этом `PREPARE TRANSACTION` игнорируется, `COMMIT PREPARED` декодируется как `COMMIT`, а `ROLLBACK PREPARED` декодируется как `ROLLBACK`.

Для поддержки передачи двухфазных команд модуль вывода должен предоставлять дополнительные обработчики. Определены несколько обязательных обработчиков двухфазной фиксации: (`begin_prepare_cb`, `prepare_cb`, `commit_prepared_cb`, `rollback_prepared_cb` и `stream_prepare_cb`) и необязательный (`filter_prepare_cb`).

Если предоставляются обработчики модуля вывода для декодирования команд двухфазной фиксации, то при выполнении `PREPARE TRANSACTION` декодируются изменения этой транзакции, которые передаются в модуль вывода, и вызывается обработчик `prepare_cb`. Это отличается от простой схемы декодирования, когда изменения передаются в модуль вывода только при фиксировании транзакции. Начало подготовленной транзакции обозначается вызовом `begin_prepare_cb`.

Когда подготовленная транзакция откатывается командой `ROLLBACK PREPARED`, вызывается обработчик `rollback_prepared_cb`, а когда подготовленная транзакция фиксируется командой `COMMIT PREPARED`, вызывается обработчик `commit_prepared_cb`.

Модуль вывода может определять правила фильтрации, воспользовавшись `filter_prepare_cb`, чтобы декодировать в две фазы только определённые транзакции. Это можно реализовать, сопоставляя с некоторым шаблоном *`gid`* или производя поиск по *`xid`*.

Реализуя декодирование подготовленных транзакций, следует учитывать следующие моменты:

- Если подготовленная транзакция блокирует таблицы каталога (в том числе пользовательские) в исключительном режиме, декодирование PREPARE может заблокироваться, ожидая завершения основной транзакции.
- Решение, осуществляющее логическую репликацию, которое организует распределённую двухфазную фиксацию с использованием этой функциональности, может заблокироваться, если подготовленная транзакция в исключительном режиме заблокирует таблицы каталога (в том числе пользовательские). Чтобы избежать этого, пользователи должны воздержаться от блокировок таблиц каталога (например, явной командой `LOCK`) в таких транзакциях. За подробностями обратитесь к Подразделу 47.8.2.

Глава 48. Отслеживание прогресса репликации

Инфраструктура источников репликации введена для упрощения реализации решений логической репликации на основе логического декодирования. Она помогает решить две распространённых проблемы:

- Как надёжно отслеживать прогресс репликации
- Как менять поведение репликации в зависимости от источника строки; например, для предотвращения циклов при двунаправленной репликации

Источники репликации имеют только два свойства: имя и ID. Имя, по которому к источнику следует обращаться из разных систем, задаётся значением типа `text` в произвольной форме. Его следует выбирать так, чтобы конфликты между источниками репликации, созданными различными средствами репликации были маловероятны; например, добавлять в начало обозначение средства репликации. ID используется только для того, чтобы не приходилось хранить длинное имя там, где требуется минимизировать объём. Он не может разделяться между разными системами.

Источник репликации можно создать функцией `pg_replication_origin_create()`; удалить функцией `pg_replication_origin_drop()`; и увидеть в системном каталоге `pg_replication_origin`.

Одной из нетривиальных задач при организации репликации является надёжное отслеживание прогресса воспроизведения. Например, когда применяющий изменения процесс (или весь кластер) умирает, нужно иметь возможность понять, какие данные были переданы успешно. Наивные решения этой проблемы, такие как изменение строки в некоторой таблице для каждой воспроизведённой транзакции, чреваты дополнительной нагрузкой во время выполнения и раздуванием базы данных.

С использованием инфраструктуры источников репликации сеанс может быть помечен как воспроизводящий изменения с удалённого узла (с помощью функции `pg_replication_origin_session_setup()`). В дополнение к этому для каждой транзакции из источника можно задать LSN и время фиксации, вызвав `pg_replication_origin_xact_setup()`. Если сделать всё это, прогресс репликации можно будет отслеживать надёжным образом. Прогресс воспроизведения для всех источников репликации можно увидеть в представлении `pg_replication_origin_status`. Прогресс отдельного источника, например, при возобновлении репликации, можно получить для любого источника, воспользовавшись функцией `pg_replication_origin_progress()`, или для источника, настроенного в текущем сеансе, с помощью `pg_replication_origin_session_progress()`.

В топологиях репликации более сложных, чем простая репликация с одной системы в другую, возможна ещё одна проблема — повторная репликация уже воспроизведённых строк, что может приводить к зацикливанию и снижению эффективности. В качестве механизма выявления и предотвращения повторной репликации так же могут оказаться полезны источники репликации. Если воспользоваться функциями, упомянутыми в предыдущем абзаце, во все поступающие в сеансе транзакции и изменения, передаваемые обработчикам модулей вывода (см. Раздел 47.6), добавляется пометка источника репликации для текущего сеанса. Это позволяет обрабатывать их в модуле вывода по-разному, например, игнорировать все строки, кроме имеющих локальное происхождение. Кроме того, обработчик вызова `filter_by_origin_cb` позволяет отфильтровать поток изменений логического декодирования в зависимости от источника. Фильтрация через этот обработчик не так гибка, как проверка записей внутри модуля вывода, но зато гораздо эффективнее.

Глава 49. Модули архивирования

Postgres Pro предоставляет инфраструктуру для создания пользовательских модулей непрерывного архивирования (см. Раздел 24.3). Хотя выполнять архивирование, используя команду оболочки (т. е. archive_command) намного проще, пользовательский модуль архивирования может оказаться значительно более надёжным и производительным.

Если в параметре archive_library указана библиотека пользовательского модуля, сервер Postgres Pro будет отправлять заполненные файлы WAL этому модулю и по возможности не будет повторно использовать или удалять эти файлы до тех пор, пока модуль не сообщит, что они были успешно заархивированы. В конечном счёте модуль сам решает, что делать с каждым файлом WAL, но многие рекомендации перечислены в Подразделе 24.3.1.

Модули архивирования должны включать как минимум функцию инициализации (см. Раздел 49.1) и необходимые обработчики (см. Раздел 49.2). Однако этим их возможности не ограничиваются (например, они могут объявлять параметры GUC и регистрировать фоновые рабочие процессы).

Рабочий пример, демонстрирующий некоторые полезные приёмы, можно найти в модуле `contrib/basic_archive`.

49.1. Функции инициализации

Библиотека архивирования загружается путём динамической загрузки общей библиотеки с базовым именем archive_library. Для нахождения файла библиотеки просматривается обычный путь поиска библиотек. Чтобы предоставить требуемые обработчики модуля архивирования и указать, что библиотека на самом деле является модулем архивирования, она должна содержать функцию с именем `_PG_archive_module_init`. Возвращать эта функция должна указатель на структуру типа `ArchiveModuleCallbacks`, содержащую всё, что нужно знать коду ядра, чтобы использовать этот метод доступа. Возвращаемое значение должно существовать всё время жизни сервера, что обычно достигается объявлением глобальной переменной `static const`.

```
typedef struct ArchiveModuleCallbacks
{
    ArchiveStartupCB startup_cb;
    ArchiveCheckConfiguredCB check_configured_cb;
    ArchiveFileCB archive_file_cb;
    ArchiveShutdownCB shutdown_cb;
} ArchiveModuleCallbacks;
typedef const ArchiveModuleCallbacks *(*ArchiveModuleInit) (void);
```

Обязательным является только обработчик `archive_file_cb`, любые другие могут отсутствовать.

49.2. Обработчики модулей архивирования

Обработчики архивирования определяют, как именно модуль будет выполнять архивирование. Сервер будет вызывать их по мере необходимости для обработки каждого отдельного файла WAL.

49.2.1. Обработчик запуска

Обработчик `startup_cb` вызывается вскоре после загрузки модуля. Этот обработчик можно использовать для любой необходимой дополнительной инициализации. Если есть данные о состоянии модуля архивирования, обработчик может использовать `state->private_data` для их хранения.

```
typedef void (*ArchiveStartupCB) (ArchiveModuleState *state);
```

49.2.2. Обработчик проверки

Обработчик `check_configured_cb` вызывается, чтобы проверить, полностью ли настроен модуль и готов ли он принимать файлы WAL (в частности, что для его параметров конфигурации установлены допустимые значения). Если функция `check_configured_cb` не определена, сервер всегда предполагает, что модуль готов к работе.

```
typedef bool (*ArchiveCheckConfiguredCB) (ArchiveModuleState *state);
```

Если возвращается `true`, сервер перейдёт к архивированию файла, вызвав обработчик `archive_file_cb`. Если возвращается `false`, архивирование не производится и архиватор выдаст в журнал сервера следующее сообщение:

```
ВНИМАНИЕ: включён режим archive_mode, но архивирование не настроено
```

В последнем случае сервер будет периодически вызывать эту функцию, и архивирование начнётся только тогда, когда она вернёт `true`.

> **Примечание**
>
> Может быть полезно добавить дополнительную информацию к общему тексту предупреждения, когда функция возвращает `false`. Для этого добавьте сообщение в макрос `arch_module_check_errdetail` перед возвратом `false`. Как и `errdetail()`, этот макрос принимает строку формата с необязательным списком аргументов. Строка, переданная в макрос, будет выведена как строка `DETAIL` в предупреждающем сообщении.

49.2.3. Обработчик архивирования

Обработчик `archive_file_cb` вызывается для архивирования одного файла WAL.

```
typedef bool (*ArchiveFileCB) (ArchiveModuleState *state, const char *file, const char
 *path);
```

Если возвращается `true`, сервер считает, что файл был успешно заархивирован, и может переработать или удалить исходный файл WAL. Если возвращается `false` или возникает ошибка, сервер сохраняет исходный файл WAL и повторяет попытку архивирования позже. Аргумент *файл* содержит только имя архивируемого файла WAL, а *путь* содержит полный путь к файлу WAL (включая имя файла).

> **Примечание**
>
> Обработчик `archive_file_cb` вызывается в кратковременном контексте памяти, который будет сбрасываться между вызовами. Если нужен долгоживущий контекст, создайте его в обработчике `startup_cb`.

49.2.4. Обработчик выключения

Обработчик `shutdown_cb` вызывается, когда завершается процесс архиватора (например, после ошибки) или изменяется значение archive_library. Если функция `shutdown_cb` не определена, никакие специальные действия в этих случаях не предпринимаются. Если есть данные о состоянии модуля архивирования, этот обработчик должен удалить их во избежание утечек.

```
typedef void (*ArchiveShutdownCB) (ArchiveModuleState *state);
```

Часть VI. Справочное руководство

Статьи этого справочного руководства составлены так, чтобы дать в разумном объёме авторитетную, полную и формальную сводку по соответствующим темам. Дополнительные сведения об использовании Postgres Pro в повествовательной, ознакомительной или показательной форме можно найти в других частях этой книги. Ссылки на них можно найти на страницах этого руководства.

Все эти справочные статьи также публикуются в виде традиционных страниц «man».

Команды SQL

Эта часть документации содержит справочную информацию по командам SQL, поддерживаемым Postgres Pro. Под «SQL» здесь понимается язык вообще; сведения о соответствии стандартам и совместимости всех команд приведены на соответствующих страниц справочника.

ABORT

ABORT — прервать текущую транзакцию

Синтаксис

```
ABORT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Описание

`ABORT` откатывает текущую транзакцию и приводит к отмене всех изменений, внесённых транзакцией. Эта команда ведёт себя так же, как и стандартная SQL-команда `ROLLBACK`, и существует только по историческим причинам.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

`AND CHAIN`

Если добавляется указание `AND CHAIN`, сразу после окончания текущей транзакции начинается новая с такими же характеристиками транзакции (см. `SET TRANSACTION`). В противном случае новая транзакция не начинается.

Примечания

Чтобы завершить и зафиксировать транзакцию, используйте `COMMIT`.

При выполнении команды `ABORT` вне блока транзакции выдаётся предупреждение и больше ничего не происходит.

Примеры

Чтобы прервать все операции:

```
ABORT;
```

Совместимость

Эта команда является расширением Postgres Pro и существует по историческим причинам. Ей равнозначна стандартная SQL-команда `ROLLBACK`.

См. также

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

ALTER AGGREGATE — изменить определение агрегатной функции

Синтаксис

```
ALTER AGGREGATE имя ( сигнатура_агр_функции ) RENAME TO новое_имя
ALTER AGGREGATE имя ( сигнатура_агр_функции )
                OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER AGGREGATE имя ( сигнатура_агр_функции ) SET SCHEMA новая_схема

Здесь сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`ALTER AGGREGATE` изменяет определение агрегатной функции.

Выполнить `ALTER AGGREGATE` может только владелец соответствующей агрегатной функции. Чтобы сменить схему агрегатной функции, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме агрегатной функции. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать агрегатную функцию. Однако суперпользователь может сменить владельца агрегатной функции в любом случае.)

Параметры

имя

Имя существующей агрегатной функции (возможно, дополненное схемой).

режим_аргумента

Режим аргумента: `IN` или `VARIADIC`. По умолчанию подразумевается `IN`.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле `ALTER AGGREGATE` не обращает внимание на имена аргументов, так как для однозначной идентификации агрегатной функции достаточно только типов аргументов.

тип_аргумента

Тип входных данных, с которыми работает агрегатная функция. Чтобы сослаться на агрегатную функцию без аргументов, укажите вместо списка аргументов `*`, а чтобы сослаться на сортирующую агрегатную функцию, добавьте `ORDER BY` между указаниями непосредственных и агрегируемых аргументов.

новое_имя

Новое имя агрегатной функции.

новый_владелец

Новый владелец агрегатной функции.

новая_схема

Новая схема агрегатной функции.

Примечания

Если вы хотите сослаться на сортирующую агрегатную функцию, рекомендуется добавить `ORDER BY` между непосредственными и агрегируемыми аргументами так же, как и в `CREATE AGGREGATE`. Однако команда сработает и без `ORDER BY`, если непосредственные и агрегирующие аргументы перечислены подряд в одном списке. В такой сокращённой форме, если и в списке непосредственных, и в списке агрегирующих аргументов содержится `VARIADIC "any"`, достаточно написать `VARIADIC "any"` только один раз.

Примеры

Переименование агрегатной функции `myavg` для типа `integer` в `my_average`:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

Смена владельца агрегатной функции `myavg` для типа `integer` на `joe`:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

Перемещение сортирующей агрегатной функции `mypercentile` с непосредственным аргументом типа `float8` и агрегируемым аргументом типа `integer` в схему `myschema`:

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer) SET SCHEMA myschema;
```

Это тоже будет работать:

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA myschema;
```

Совместимость

Оператор `ALTER AGGREGATE` отсутствует в стандарте SQL.

См. также

CREATE AGGREGATE, DROP AGGREGATE

ALTER COLLATION

ALTER COLLATION — изменить определение правила сортировки

Синтаксис

```
ALTER COLLATION имя REFRESH VERSION

ALTER COLLATION имя RENAME TO новое_имя
ALTER COLLATION имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER COLLATION имя SET SCHEMA новая_схема
```

Описание

`ALTER COLLATION` изменяет определение правила сортировки.

Выполнить `ALTER COLLATION` может только владелец соответствующего правила сортировки. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме правила сортировки. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать правило сортировки. Однако суперпользователь может сменить владельца правила сортировки в любом случае.)

Параметры

`имя`

Имя существующего правила сортировки (возможно, дополненное схемой).

`новое_имя`

Новое имя правила сортировки.

`новый_владелец`

Новый владелец правила сортировки.

`новая_схема`

Новая схема правила сортировки.

`REFRESH VERSION`

Обновить версию правила сортировки. Подробности в разделе Замечания ниже.

Замечания

Когда создаётся объект правила сортировки, в системный каталог записывается версия этого правила, определяемая его провайдером. Когда такое правило используется, текущая версия сравнивается с записанной и в случае несовпадения выдаётся предупреждение, например такое:

```
WARNING:  collation "xx-x-icu" has version mismatch
DETAIL:  The collation in the database was created using version 1.2.3.4, but the
 operating system provides version 2.3.4.5.
HINT:  Rebuild all objects affected by this collation and run ALTER COLLATION
 pg_catalog."xx-x-icu" REFRESH VERSION, or build Postgres Pro with the right library
 version.
```

Изменения в определениях правил сортировки могут приводить к разрушению индексов и другим проблемам, так как СУБД рассчитывает на то, что хранимые объекты отсортированы в определённом порядке. Вообще этого следует избегать, но это может иметь место в совершенно легальных обстоятельствах, например при переходе на более новую версию ОС или при обновлении с использованием `pg_upgrade` исполняемых файлов сервера, скомпонованных с более новой версией ICU. Когда возникает такая ситуация, все объекты, зависящие от данного правила сортировки, должны быть перестроены, например командой `REINDEX`. После этой операции можно обновить версию правила сортировки, выполнив команду `ALTER COLLATION ... REFRESH VERSION`. При этом системный каталог будет обновлён, в него будет записана текущая версия правила, и предупреждение уйдёт. Заметьте, что эта команда собственно не проверяет, были ли все зависимые объекты перестроены корректно.

Когда применяются правила сортировки, предоставляемые библиотекой `libc`, информация о версии записывается в системах, где используется GNU C (в большинстве ОС Linux), а также во FreeBSD и Windows. Когда применяются правила сортировки, которые предоставляет библиотека ICU, информация о версии выдаётся библиотекой ICU и доступна на всех платформах.

Примечание

Когда правила сортировки предоставляются библиотекой GNU C, версии этих правил косвенным образом определяются по версии библиотеки. Во многих дистрибутивах Linux определения правил сортировки меняются только при обновлении библиотеки C, но возможны исключения, так как авторы дистрибутива могут по своему желанию перенести обновлённые определения в старые версии библиотеки.

В Windows информация о версии правил сортировки доступна только для правил, определённых с меткой языка в формате BCP 47, например `en-US`.

Для основного правила сортировки базы данных существует аналогичная команда `ALTER DATABASE ... REFRESH COLLATION VERSION`.

Следующий запрос позволяет выбрать все правила сортировки в текущей базе данных, которые требуют обновления, и зависящие от них объекты:

```
SELECT pg_describe_object(refclassid, refobjid, refobjsubid) AS "Collation",
       pg_describe_object(classid, objid, objsubid) AS "Object"
  FROM pg_depend d JOIN pg_collation c
       ON refclassid = 'pg_collation'::regclass AND refobjid = c.oid
  WHERE c.collversion <> pg_collation_actual_version(c.oid)
  ORDER BY 1, 2;
```

Примеры

Изменение названия правила сортировки с `ru_RU` на `russian`:

```
ALTER COLLATION "ru_RU" RENAME TO russian;
```

Изменение владельца правила сортировки `en_US` на `joe`:

```
ALTER COLLATION "en_US" OWNER TO joe;
```

Совместимость

Оператор `ALTER COLLATION` отсутствует в стандарте SQL.

См. также

CREATE COLLATION, DROP COLLATION

ALTER CONVERSION

ALTER CONVERSION — изменить определение перекодировки

Синтаксис

```
ALTER CONVERSION имя RENAME TO новое_имя
ALTER CONVERSION имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER CONVERSION имя SET SCHEMA новая_схема
```

Описание

`ALTER CONVERSION` изменяет определение перекодировки.

Выполнить `ALTER CONVERSION` может только владелец соответствующей перекодировки. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме перекодировки. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать перекодировку. Однако суперпользователь может сменить владельца перекодировки в любом случае.)

Параметры

`имя`

Имя существующей перекодировки (возможно, дополненное схемой).

`новое_имя`

Новое имя перекодировки.

`новый_владелец`

Новый владелец перекодировки.

`новая_схема`

Новая схема перекодировки.

Примеры

Переименование перекодировки `iso_8859_1_to_utf8` в `latin1_to_unicode`:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

Смена владельца перекодировки `iso_8859_1_to_utf8` на `joe`:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Совместимость

Оператор `ALTER CONVERSION` отсутствует в стандарте SQL.

См. также

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

ALTER DATABASE — изменить атрибуты базы данных

Синтаксис

```
ALTER DATABASE имя [ [ WITH ] параметр [ ... ] ]

Здесь параметр:

    ALLOW_CONNECTIONS разр_подключения
    CONNECTION LIMIT предел_подключений
    IS_TEMPLATE это_шаблон

ALTER DATABASE имя RENAME TO новое_имя

ALTER DATABASE имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }

ALTER DATABASE имя SET TABLESPACE новое_табл_пространство

ALTER DATABASE имя REFRESH COLLATION VERSION

ALTER DATABASE имя SET параметр_конфигурации { TO | = } { значение | DEFAULT }
ALTER DATABASE имя SET параметр_конфигурации FROM CURRENT
ALTER DATABASE имя RESET параметр_конфигурации
ALTER DATABASE имя RESET ALL
```

Описание

`ALTER DATABASE` изменяет атрибуты базы данных.

Первая форма оператора меняет параметры на уровне базы данных. (Подробнее описано ниже.) Изменять эти параметры может только владелец базы данных или суперпользователь.

Вторая форма меняет имя базы данных. Переименовать базу данных может только владелец БД или суперпользователь (не суперпользователю требуется также право `CREATEDB`). Переименовать текущую базу данных нельзя. (Если вам нужно сделать это, сначала подключитесь к другой базе.)

Третья форма меняет владельца базы данных. Чтобы сменить владельца базы, необходимо иметь право `SET ROLE` для новой роли-владельца, и кроме того, иметь право `CREATEDB`. (Заметьте, что суперпользователи наделяются всеми этими правами автоматически.)

Четвёртая форма меняет табличное пространство по умолчанию для базы данных. Произвести это изменение может только её владелец или суперпользователь; кроме того, необходимо иметь право для создания нового табличного пространства. Эта команда физически переносит все таблицы или индексы из прежнего основного табличного пространства БД в новое. Новое табличное пространство должно быть пустым для этой базы данных и к ней никто не должен быть подключён. Таблицы и индексы, находящиеся не в основном табличном пространстве, при этом не затрагиваются.

Остальные формы меняют значение по умолчанию конфигурационных переменных времени выполнения для базы данных Postgres Pro. Когда устанавливается следующий сеанс работы с указанной базой данных, заданное этой командой значение становится значением по умолчанию. Значения переменных, заданные для базы данных, переопределяют значения, определённые в `postgresql.conf` или полученные через командную строку `postgres`. Менять сеансовые значения переменных для базы данных может только её владелец или суперпользователь. Некоторые параметры изменить таким образом нельзя, а некоторые может изменить только суперпользователь.

Параметры

имя

Имя базы данных, атрибуты которой изменяются.

разр_подключения

Если false, никто не сможет подключаться к этой базе данных.

предел_подключений

Число разрешённых одновременно подключений к этой базе данных (-1 снимает ограничение).

это_шаблон

Если true, базу данных сможет клонировать любой пользователь с правами `CREATEDB`; в противном случае клонировать эту базу смогут только суперпользователи и её владелец.

новое_имя

Новое имя базы данных.

новый_владелец

Новый владелец базы данных.

новое_табл_пространство

Новое основное табличное пространство базы данных.

Эту форму команды нельзя выполнять внутри блока транзакции.

`REFRESH COLLATION VERSION`

Обновить версию правила сортировки для базы данных. Подробнее в Замечания.

параметр_конфигурации
значение

Устанавливает сеансовое значение по умолчанию для указанного параметра конфигурации. Если указывается *значение* `DEFAULT` или равнозначный вариант, `RESET`, определение параметра на уровне базы данных удаляется, так что в новых сеансах будет действовать значение по умолчанию, определённое на уровне системы. Для очистки всех значений параметров на уровне базы данных выполните `RESET ALL`. `SET FROM CURRENT` устанавливает значение параметра на уровне базы данных из текущего значения в активном сеансе.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

Примечания

Также возможно связать параметры сеанса не с базой данных, а с определённой ролью; см. ALTER ROLE. В случае конфликта параметры на уровне роли переопределяют параметры на уровне базы данных.

Примеры

Отключение сканирования индекса по умолчанию в базе данных `test`:

```
ALTER DATABASE test SET enable_indexscan TO off;
```

Совместимость

Оператор `ALTER DATABASE` является расширением Postgres Pro.

См. также

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES — определить права доступа по умолчанию

Синтаксис

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } целевая_роль [, ...] ]
    [ IN SCHEMA имя_схемы [, ...] ]
    предложение_GRANT_или_REVOKE

Где предложение_GRANT_или_REVOKE может быть следующим:

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER |
 MAINTAIN }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTIONS | ROUTINES }
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | CREATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    TO { [ GROUP ] имя_роли | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER |
 MAINTAIN }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTIONS | ROUTINES }
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | CREATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    FROM { [ GROUP ] имя_роли | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

Описание

Команда `ALTER DEFAULT PRIVILEGES` позволяет задавать права, которые будут применяться к объектам, созданным в будущем. (Она не влияет на права, назначенные уже существующим объектам.) Права могут быть заданы глобально (т. е. для всех объектов, создаваемых в текущей базе данных) или только для объектов, создаваемых в указанных схемах.

Хотя пользователь может изменять свои права по умолчанию и права по умолчанию для ролей, членом которых он является, на права объекта при создании влияют только права по умолчанию текущей роли, и они не наследуются от любых ролей, в которые включена текущая роль.

Как объясняется в Разделе 5.8, права по умолчанию для объектов любого типа обычно дают все назначаемые разрешения владельцу объекта, а также могут давать некоторые разрешения роли `PUBLIC`. Однако это поведение можно поменять, изменив права по умолчанию командой `ALTER DEFAULT PRIVILEGES`.

В настоящее время можно изменять права только для схем, таблиц (включая представления и сторонние таблицы), последовательностей, функций и типов (включая домены). Применительно к данной команде функциями считаются также агрегатные функции и процедуры. Слова `FUNCTIONS` и `ROUTINES` для неё считаются равнозначными. (`ROUTINES` предпочтительнее в перспективе как стандартный термин, охватывающий и функции, и процедуры. В более ранних выпусках Postgres Pro допускалось только слово `FUNCTIONS`. Задать права по умолчанию для функций и процедур по отдельности нельзя.)

Заданные на уровне схемы права по умолчанию добавляются к тем, что определены глобально для конкретного типа объекта. Это означает, что вы не можете отозвать права уровня схемы, если они назначены глобально (либо по умолчанию, либо предыдущей командой `ALTER DEFAULT PRIVILEGES` без указания схемы). Команда `REVOKE` для схемы может быть полезна только для отмены действия предыдущей команды `GRANT` для этой же схемы.

Параметры

`целевая_роль`

Изменяются права по умолчанию для объектов, созданных *`целевой_ролью`* или текущей, если целевая роль не указана.

`имя_схемы`

Имя существующей схемы. Если указано, права по умолчанию меняются для объектов, которые будут созданы в этой схеме. Если `IN SCHEMA` опущено, меняются глобальные права по умолчанию. Указание `IN SCHEMA` не допускается при установлении прав для схем, так как схемы не могут быть вложенными.

`имя_роли`

Имя существующей роли, для которой даются или отзываются права. Этот и все другие параметры в *`предложении_grant_или_revoke`* действуют как описано в GRANT или REVOKE, за исключением того, что они распространяются не на один конкретный объект, а на целый класс объектов.

Примечания

Чтобы узнать текущие назначенные права по умолчанию, воспользуйтесь командой `\ddp` в psql. Как интерпретировать выводимую ей информацию, рассказывается в описании команды `\dp` в Разделе 5.8.

Если вы желаете удалить роль, права по умолчанию для которой были изменены, необходимо явно отменить изменения прав по умолчанию или воспользоваться командой `DROP OWNED BY` для избавления от назначенных для этой роли прав по умолчанию.

Примеры

Наделение всех правом SELECT для всех таблиц (и представлений), которые будут созданы в дальнейшем в схеме `myschema`, и наделение роли `webuser` правом INSERT для этих же таблиц:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

Отмена предыдущих изменений с тем, чтобы для таблиц, создаваемых в будущем, были определены только обычные права, без дополнительных:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

Лишение роли public права на выполнение (EXECUTE), которое обычно даётся для функций (для всех функций, которые будут созданы ролью `admin`):

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

Однако заметьте, что этого же эффекта *нельзя* добиться с помощью команды, ограниченной одной схемой. Эта команда будет действовать, только если ей предшествовала соответствующая команда `GRANT`:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

Это объясняется тем, что на уровне схемы можно только назначить права по умолчанию, которые добавятся к назначенным глобально, но нельзя отозвать последние.

Совместимость

Оператор `ALTER DEFAULT PRIVILEGES` отсутствует в стандарте SQL.

См. также

GRANT, REVOKE

ALTER DOMAIN

ALTER DOMAIN — изменить определение домена

Синтаксис

```
ALTER DOMAIN имя
    { SET DEFAULT выражение | DROP DEFAULT }
ALTER DOMAIN имя
    { SET | DROP } NOT NULL
ALTER DOMAIN имя
    ADD ограничение_домена [ NOT VALID ]
ALTER DOMAIN имя
    DROP CONSTRAINT [ IF EXISTS ] имя_ограничения [ RESTRICT | CASCADE ]
ALTER DOMAIN имя
     RENAME CONSTRAINT имя_ограничения TO имя_нового_ограничения
ALTER DOMAIN имя
    VALIDATE CONSTRAINT имя_ограничения
ALTER DOMAIN имя
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER DOMAIN имя
    RENAME TO новое_имя
ALTER DOMAIN имя
    SET SCHEMA новая_схема

Здесь ограничение_домена:

[ CONSTRAINT имя_ограничения ]
{ NOT NULL | CHECK (выражение) }
```

Описание

`ALTER DOMAIN` изменяет определение существующего домена. Эта команда имеет несколько разновидностей:

`SET/DROP DEFAULT`

Эти формы задают/убирают значение по умолчанию для домена. Заметьте, что эти значения по умолчанию применяются только при последующих командах `INSERT`; они не меняются в строках с данным доменом, уже добавленных в таблицу.

`SET/DROP NOT NULL`

Эти формы определяют, будет ли домен принимать значения NULL или нет. `SET NOT NULL` можно выполнить, только если столбцы с этим доменом ещё не содержат значений NULL.

`ADD` *`ограничение_домена`* `[ NOT VALID ]`

Эта форма добавляет новое ограничение для домена. Когда добавляется новое ограничение домена, все столбцы с этим доменом будут проверены на соответствие этому ограничению. Эти проверки можно подавить, добавив указание `NOT VALID`, а затем активировать позднее с помощью команды `ALTER DOMAIN ... VALIDATE CONSTRAINT`. Вновь вставленные или изменённые строки всегда проверяются по всем ограничениям, даже тем, что отмечены как `NOT VALID`. Указание `NOT VALID` допускается только для ограничений `CHECK`.

`DROP CONSTRAINT [ IF EXISTS ]`

Эта форма убирает ограничения домена. Если указано `IF EXISTS` и заданное ограничение не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

`RENAME CONSTRAINT`

Эта форма меняет название ограничения домена.

`VALIDATE CONSTRAINT`

Эта форма включает проверку ограничения, ранее добавленного как `NOT VALID`, то есть проверяет все значения в столбцах с этим типом домена на соответствие этому ограничению.

`OWNER`

Эта форма меняет владельца домена на заданного пользователя.

`RENAME`

Эта форма меняет название домена.

`SET SCHEMA`

Эта форма меняет схему домена. Все ограничения, связанные с данным доменом, так же переносятся в новую схему.

Выполнить `ALTER DOMAIN` может только владелец соответствующего домена. Чтобы сменить схему домена, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме домена. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать домен. Однако суперпользователь может сменить владельца домена в любом случае.)

Параметры

имя

Имя существующего домена (возможно, дополненное схемой), подлежащего изменению.

ограничение_домена

Новое ограничение домена.

имя_ограничения

Имя существующего ограничения, подлежащего удалению или переименованию.

`NOT VALID`

Не проверять существующие сохранённые данные на соответствие ограничению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного ограничения, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении ограничения, если существуют зависящие от него объекты. Это поведение по умолчанию.

новое_имя

Новое имя домена.

имя_нового_ограничения

Новое имя ограничения.

новый_владелец

Имя пользователя, назначаемого новым владельцем домена.

новая_схема

Новая схема домена.

Примечания

Хотя команда `ALTER DOMAIN ADD CONSTRAINT` пытается проверить, удовлетворяют ли уже существующие данные новому ограничению, эта проверка не очень надёжна, так как данная команда не «видит» строки таблицы, которые были только что добавлены или изменены, но ещё не зафиксированы. Если есть риск того, что параллельные операции могут вставить неподходящие данные, можно применить следующий подход: создать ограничение с указанием `NOT VALID`, зафиксировать эту команду, подождать окончания всех транзакций, начатых до фиксирования, а затем выполнить `ALTER DOMAIN VALIDATE CONSTRAINT` для проведения контроля данных. В этом случае проверка будет надёжной, так как после фиксирования ограничения оно будет гарантированно действовать на все новые значения типа домена, вносимые последующими транзакциями.

В настоящее время команды `ALTER DOMAIN ADD CONSTRAINT`, `ALTER DOMAIN VALIDATE CONSTRAINT` и `ALTER DOMAIN SET NOT NULL` выдают ошибку, если указанный домен или любой производный от него используется в столбце с типом-контейнером (это может быть составной, диапазонный тип или массив) в какой-либо таблице базы данных. В дальнейшем они будут доработаны, с тем чтобы новое ограничение проверялось и при такой вложенности.

Примеры

Добавление ограничения `NOT NULL` к домену:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

Удаление ограничения `NOT NULL` из домена:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

Добавление ограничения-проверки к домену:

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

Удаление ограничения-проверки из домена:

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

Переименование ограничения-проверки в домене:

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

Перемещение домена в другую схему:

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Совместимость

`ALTER DOMAIN` соответствует стандарту SQL, за исключением подвидов `OWNER`, `RENAME`, `SET SCHEMA` и `VALIDATE CONSTRAINT`, которые являются расширениями Postgres Pro. Предложение `NOT VALID` вариации `ADD CONSTRAINT` также является расширением Postgres Pro.

См. также

CREATE DOMAIN, DROP DOMAIN

ALTER EVENT TRIGGER

ALTER EVENT TRIGGER — изменить определение событийного триггера

Синтаксис

```
ALTER EVENT TRIGGER имя DISABLE
ALTER EVENT TRIGGER имя ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER EVENT TRIGGER имя RENAME TO новое_имя
```

Описание

`ALTER EVENT TRIGGER` изменяет свойства существующего событийного триггера.

Для изменения событийного триггера нужно быть суперпользователем.

Параметры

имя

Имя существующего триггера, подлежащего изменению.

новый_владелец

Имя пользователя, назначаемого новым владельцем событийного триггера.

новое_имя

Новое имя событийного триггера.

`DISABLE`/`ENABLE [ REPLICA | ALWAYS ]`

Эти формы настраивают срабатывание событийных триггеров. Отключённый триггер сохраняется в системе, но не выполняется, когда происходит его событие срабатывания. См. также session_replication_role.

Совместимость

Оператор `ALTER EVENT TRIGGER` отсутствует в стандарте SQL.

См. также

CREATE EVENT TRIGGER, DROP EVENT TRIGGER

ALTER EXTENSION

ALTER EXTENSION — изменить определение расширения

Синтаксис

```
ALTER EXTENSION имя UPDATE [ TO новая_версия ]
ALTER EXTENSION имя SET SCHEMA новая_схема
ALTER EXTENSION имя ADD элемент_объект
ALTER EXTENSION имя DROP элемент_объект

Здесь элемент_объект:

  ACCESS METHOD имя_объекта |
  AGGREGATE имя_агрегатной_функции ( сигнатура_агр_функции ) |
  CAST (исходный_тип AS целевой_тип) |
  COLLATION имя_объекта |
  CONVERSION имя_объекта |
  DOMAIN имя_объекта |
  EVENT TRIGGER имя_объекта |
  FOREIGN DATA WRAPPER имя_объекта |
  FOREIGN TABLE имя_объекта |
  FUNCTION имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  MATERIALIZED VIEW имя_объекта |
  OPERATOR имя_оператора (тип_слева, тип_справа) |
  OPERATOR CLASS имя_объекта USING индексный_метод |
  OPERATOR FAMILY имя_объекта USING индексный_метод |
  [ PROCEDURAL ] LANGUAGE имя_объекта |
  PROCEDURE имя_процедуры [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  ROUTINE имя_подпрограммы [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  SCHEMA имя_объекта |
  SEQUENCE имя_объекта |
  SERVER имя_объекта |
  TABLE имя_объекта |
  TEXT SEARCH CONFIGURATION имя_объекта |
  TEXT SEARCH DICTIONARY имя_объекта |
  TEXT SEARCH PARSER имя_объекта |
  TEXT SEARCH TEMPLATE имя_объекта |
  TRANSFORM FOR имя_типа LANGUAGE имя_языка |
  TYPE имя_объекта |
  VIEW имя_объекта

и сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`ALTER EXTENSION` изменяет определение установленного расширения. Эта команда имеет несколько подвидов:

`UPDATE`

Эта форма обновляет версию расширения. Расширение должно предоставлять подходящий скрипт обновления (или набор скриптов), который может сменить текущую установленную версию на требуемую.

`SET SCHEMA`

Эта форма переносит объекты расширения в другую схему. Чтобы эта команда выполнилась успешно, расширение должно быть *перемещаемым*.

`ADD` *`элемент_объект`*

Эта форма добавляет существующий объект в расширение. В основном это применяется в скриптах обновления расширений. Добавленный объект затем будет считаться частью расширения, и удалить его можно будет, только удалив расширение.

`DROP` *`элемент_объект`*

Эта форма удаляет из расширения включённый в него объект. В основном это применяется в скриптах обновления расширений. Сам объект при этом не уничтожается, а только отделяется от расширения.

Подробнее эти операции описаны в Разделе 36.17.

Чтобы выполнить команду `ALTER EXTENSION`, необходимо быть владельцем данного расширения. Для форм `ADD`/`DROP` требуется также быть владельцем добавляемого/удаляемого объекта.

Параметры

`имя`

Имя установленного расширения.

`новая_версия`

Запрашиваемая новая версия расширения. Её можно записать в виде идентификатора или строкового значения. Если она не указана, `ALTER EXTENSION UPDATE` пытается выполнить обновление до версии, указанной в качестве версии по умолчанию в управляющем файле расширения.

`новая_схема`

Новая схема расширения.

`имя_объекта`
`имя_агрегатной_функции`
`имя_функции`
`имя_оператора`
`имя_процедуры`
`имя_подпрограммы`

Имя объекта, добавляемого или удаляемого из расширения. Имена таблиц, агрегатных функций, доменов, сторонних таблиц, функций, операторов, классов операторов, семейств операторов, процедур, подпрограмм, последовательностей, объектов текстового поиска, типов и представлений можно дополнить именем схемы.

`исходный_тип`

Имя исходного типа данных для приведения.

`целевой_тип`

Имя целевого типа данных для приведения.

режим_аргумента

Режим аргумента функции, процедуры или агрегата: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `ALTER EXTENSION` не учитывает аргументы `OUT`, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы `IN`, `INOUT` и `VARIADIC`.

имя_аргумента

Имя аргумента функции, процедуры или агрегата. Заметьте, что на самом деле `ALTER EXTENSION` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргумента функции, процедуры или агрегата.

тип_слева
тип_справа

Тип данных аргументов оператора (возможно, дополненный именем схемы). В случае отсутствия аргумента префиксного оператора укажите вместо типа `NONE`.

`PROCEDURAL`

Это слово не несёт смысловой нагрузки.

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

Примеры

Обновление расширения `hstore` до версии 2.0:

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

Смена схемы расширения `hstore` на `utils`:

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

Добавление существующей функции в расширение `hstore`:

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

Совместимость

Оператор `ALTER EXTENSION` является расширением Postgres Pro.

См. также

CREATE EXTENSION, DROP EXTENSION

ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER — изменить определение обёртки сторонних данных

Синтаксис

```
ALTER FOREIGN DATA WRAPPER имя
    [ HANDLER функция_обработчик | NO HANDLER ]
    [ VALIDATOR функция_проверки | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER
 | SESSION_USER }
ALTER FOREIGN DATA WRAPPER имя RENAME TO новое_имя
```

Описание

`ALTER FOREIGN DATA WRAPPER` изменяет определение обёртки сторонних данных. Первая форма команды меняет вспомогательные функции или общие параметры обёртки (требуется минимум одно предложение), а вторая — владельца обёртки.

Настраивать обёртки сторонних данных могут только суперпользователи и только суперпользователи могут быть их владельцами.

Параметры

`имя`

Имя существующей обёртки сторонних данных.

`HANDLER` *`функция_обработчик`*

Задаёт новое имя функции-обработчика для обёртки сторонних данных.

`NO HANDLER`

Эти ключевые слова указывают, что обёртка сторонних данных теперь не имеет функции-обработчика.

Заметьте, что обращаться к сторонним таблицам, если их обёртка сторонних данных не имеет обработчика, нельзя.

`VALIDATOR` *`функция_проверки`*

Задаёт новое имя функции проверки для обёртки сторонних данных.

Заметьте, что возможна ситуация, что предыдущие параметры обёртки данных, зависимых серверов, сопоставлений пользователей или сторонних таблиц окажутся неприемлемыми для новой функции проверки. Postgres Pro не проверяет их, поэтому пользователь сам должен убедиться в правильности этих параметров, прежде чем использовать изменённую обёртку данных. Однако параметры, изменяемые в данной команде `ALTER FOREIGN DATA WRAPPER`, будут проверены новой функцией проверки.

`NO VALIDATOR`

Эти ключевые слова указывают, что обёртка сторонних данных теперь не имеет функции проверки.

`OPTIONS ( [ ADD | SET | DROP ]` *`параметр`* `['`*`значение`*`'] [, ... ] )`

Эта форма настраивает параметры обёртки сторонних данных. `ADD`, `SET` и `DROP` определяют, какое действие будет выполнено (добавление, установка и удаление, соответственно). Если действие не задано явно, подразумевается `ADD`. Имена параметров должны быть уникальными, они вместе со значениями проверяются функцией проверки, если она установлена.

новый_владелец

Имя пользователя, назначаемого новым владельцем обёртки сторонних данных.

новое_имя

Новое имя обёртки сторонних данных.

Примеры

Изменение параметров обёртки сторонних данных `dbi`: добавление параметра `foo`, удаление `bar`:

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP bar);
```

Установление для обёртки сторонних данных `dbi` новой функции проверки `bob.myvalidator`:

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

Совместимость

`ALTER FOREIGN DATA WRAPPER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED), за исключением предложений `HANDLER`, `VALIDATOR`, `OWNER TO` и `RENAME`, являющихся расширениями.

См. также

CREATE FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER

ALTER FOREIGN TABLE

ALTER FOREIGN TABLE — изменить определение сторонней таблицы

Синтаксис

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    действие [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
ALTER FOREIGN TABLE [ IF EXISTS ] имя
    RENAME TO новое_имя
ALTER FOREIGN TABLE [ IF EXISTS ] имя
    SET SCHEMA новая_схема

Где действие может быть следующим:

    ADD [ COLUMN ] имя_столбца тип_данных [ COLLATE правило_сортировки ]
 [ ограничение_столбца [ ... ] ]
    DROP  [ COLUMN ] [ IF EXISTS ] имя_столбца [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] имя_столбца [ SET DATA ] TYPE тип_данных
 [ COLLATE правило_сортировки ]
    ALTER [ COLUMN ] имя_столбца SET DEFAULT выражение
    ALTER [ COLUMN ] имя_столбца DROP DEFAULT
    ALTER [ COLUMN ] имя_столбца { SET | DROP } NOT NULL
    ALTER [ COLUMN ] имя_столбца SET STATISTICS integer
    ALTER [ COLUMN ] имя_столбца SET ( атрибут = значение [, ... ] )
    ALTER [ COLUMN ] имя_столбца RESET ( атрибут [, ... ] )
    ALTER [ COLUMN ] имя_столбца SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN |
 DEFAULT }
    ALTER [ COLUMN ] имя_столбца OPTIONS ( [ ADD | SET | DROP ] параметр ['значение']
 [, ... ])
    ADD ограничение_таблицы [ NOT VALID ]
    VALIDATE CONSTRAINT имя_ограничения
    DROP CONSTRAINT [ IF EXISTS ]  имя_ограничения [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ имя_триггера | ALL | USER ]
    ENABLE TRIGGER [ имя_триггера | ALL | USER ]
    ENABLE REPLICA TRIGGER имя_триггера
    ENABLE ALWAYS TRIGGER имя_триггера
    SET WITHOUT OIDS
    INHERIT таблица_родитель
    NO INHERIT таблица_родитель
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ])
```

Описание

`ALTER FOREIGN TABLE` меняет определение существующей сторонней таблицы. Эта команда имеет несколько разновидностей:

`ADD COLUMN`

Эта форма добавляет в стороннюю таблицу новый столбец, следуя тому же синтаксису, что и `CREATE FOREIGN TABLE`. В отличие от добавления столбца в обычную таблицу, при данной операции в базовом хранилище ничего не меняется; эта команда просто объявляет о доступности нового столбца через данную стороннюю таблицу.

`DROP COLUMN [ IF EXISTS ]`

Эта форма удаляет столбец из сторонней таблицы. Если что-либо зависит от этого столбца, например, представление, для успешного результата потребуется добавить `CASCADE`. Если указано `IF EXISTS` и этот столбец не существует, ошибка не происходит, вместо этого выдаётся замечание.

`SET DATA TYPE`

Эта форма меняет тип столбца сторонней таблицы. И это не влияет на нижележащее хранилище: данная операция просто меняет тип, который по мнению Postgres Pro будет иметь этот столбец.

`SET`/`DROP DEFAULT`

Эти формы задают или удаляют значение по умолчанию для столбцов. Значения по умолчанию применяются только при последующих командах `INSERT` или `UPDATE`; их изменения не отражаются в строках, уже существующих в таблице.

`SET`/`DROP NOT NULL`

Устанавливает, будет ли столбец принимать значения NULL или нет.

`SET STATISTICS`

Эта форма задаёт ориентир сбора статистики по столбцам для последующих операций `ANALYZE`. За подробностями обратитесь к описанию подобной формы `ALTER TABLE`.

`SET ( `*`атрибут`*` = `*`значение`*` [, ... ] )`
`RESET ( `*`атрибут`*` [, ... ] )`

Эта форма задаёт или сбрасывает значения атрибутов. За подробностями обратитесь к описанию подобной формы `ALTER TABLE`.

`SET STORAGE`

Эта форма задаёт режим хранения для столбца. За подробностями обратитесь к описанию подобной формы `ALTER TABLE`. Заметьте, что режим хранения не имеет значения, если обёртка сторонних данных для этой таблицы будет игнорировать его.

`ADD `*`ограничение_таблицы`*` [ NOT VALID ]`

Эта форма добавляет новое ограничение в стороннюю таблицу с применением того же синтаксиса, что и `CREATE FOREIGN TABLE`. В настоящее время поддерживаются только ограничения `CHECK`.

В отличие от ограничения, добавляемого для обычной таблицы, ограничение сторонней таблицы фактически никак не проверяется; эта команда сводится просто к заявлению о том, что все строки в сторонней таблице предположительно удовлетворяют новому условию. (Подробнее это рассматривается в описании `CREATE FOREIGN TABLE`.) Если ограничение помечено как `NOT VALID` (непроверенное), сервер не будет полагать, что оно выполняется; такая запись делается только на случай использования в будущем.

`VALIDATE CONSTRAINT`

Эта форма отмечает ограничение, которая ранее было помечено `NOT VALID`, как проверенное. Собственно для проверки этого ограничения ничего не делается, но последующие запросы будут полагать, что оно действует.

`DROP CONSTRAINT [ IF EXISTS ]`

Эта форма удаляет указанное ограничение сторонней таблицы. Если указано `IF EXISTS` и заданное ограничение не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

Эти формы управляют триггерами, принадлежащими сторонней таблице. За подробностями обратитесь к описанию подобной формы `ALTER TABLE`.

`SET WITHOUT OIDS`

Синтаксис обратной совместимости для удаления системного столбца `oid`. Так как добавить системные столбцы `oid` теперь невозможно, это указание фактически не действует.

`INHERIT` *`таблица_родитель`*

Эта форма делает целевую стороннюю таблицу потомком указанной родительской таблицы. За подробностями обратитесь к описанию подобной формы `ALTER TABLE`.

`NO INHERIT` *`таблица_родитель`*

Эта форма удаляет целевую стороннюю таблицу из списка потомков указанной родительской таблицы.

`OWNER`

Эта форма меняет владельца сторонней таблицы на заданного пользователя.

`OPTIONS ( [ ADD | SET | DROP ]` *`параметр`* `['`*`значение`*`'] [, ... ] )`

Эта форма настраивает параметры сторонней таблицы или одного из её столбцов. `ADD`, `SET` и `DROP` определяют, какое действие будет выполнено (добавление, установка и удаление, соответственно). Если действие не задано явно, подразумевается `ADD`. Имена параметров не должны повторяться (хотя параметр таблицы и параметр столбца вполне могут иметь одно имя). Имена и значения параметров также проверяются библиотекой обёртки сторонних данных.

`RENAME`

Формы `RENAME` меняют имя сторонней таблицы или имя столбца в сторонней таблице.

`SET SCHEMA`

Эта форма переносит стороннюю таблицу в другую схему.

Все действия, кроме `RENAME` и `SET SCHEMA`, можно объединить в один список изменений и выполнить одновременно. Например, можно добавить несколько столбцов и/или изменить тип столбцов одной командой.

Если команда записана в виде `ALTER FOREIGN TABLE IF EXISTS ...` и сторонняя таблица не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

Выполнить `ALTER FOREIGN TABLE` может только владелец соответствующей таблицы. Чтобы сменить схему сторонней таблицы, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме таблицы. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать таблицу. Однако суперпользователь может сменить владельца таблицы в любом случае.) Чтобы добавить столбец или изменить тип столбца, также необходимо иметь право `USAGE` для его типа данных.

Параметры

`имя`

Имя (возможно, дополненное схемой) существующей сторонней таблицы, подлежащей изменению. Если перед именем таблицы указано `ONLY`, изменяется только заданная таблица. Без `ONLY` изменяется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание `*`, чтобы явно обозначить, что изменению подлежат все дочерние таблицы.

имя_столбца

Имя нового или существующего столбца.

новое_имя_столбца

Новое имя существующего столбца.

новое_имя

Новое имя таблицы.

тип_данных

Тип данных нового столбца или новый тип данных существующего столбца.

ограничение_таблицы

Новое ограничение уровня таблицы для сторонней таблицы.

имя_ограничения

Имя существующего ограничения, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от удаляемого столбца или ограничения (например, представления, содержащие этот столбец), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении столбца или ограничения, если существуют зависящие от них объекты. Это поведение по умолчанию.

имя_триггера

Имя включаемого или отключаемого триггера.

`ALL`

Отключает или включает все триггеры, принадлежащие сторонней таблице. (Если какие-либо из триггеров являются внутрисистемными, для этого требуются права суперпользователя. Сама система не добавляет такие триггеры в сторонние таблицы, но дополнительный код может сделать это.)

`USER`

Отключает или включает все триггеры, принадлежащие сторонней таблице, кроме сгенерированных внутрисистемных.

таблица_родитель

Родительская таблица, с которой будет установлена или разорвана связь данной сторонней таблицы.

новый_владелец

Имя пользователя, назначаемого новым владельцем таблицы.

новая_схема

Имя схемы, в которую будет перемещена таблица.

Примечания

Ключевое слово `COLUMN` не несёт смысловой нагрузки и может быть опущено.

При добавлении или удалении столбцов (`ADD COLUMN`/`DROP COLUMN`), добавлении ограничений `NOT NULL` или `CHECK` или изменении типа данных (`SET DATA TYPE`) согласованность этих определений с внешним сервером не гарантируется. Ответственность за соответствие определений таблицы удалённой стороне лежит на пользователе.

За более полным описанием параметров обратитесь к `CREATE FOREIGN TABLE`.

Примеры

Установление ограничения NOT NULL для столбца:

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

Изменение параметров сторонней таблицы:

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2 'value2',
 DROP opt3);
```

Совместимость

Формы `ADD`, `DROP` и `SET DATA TYPE` соответствуют стандарту SQL. Другие формы являются собственными расширениями Postgres Pro. Кроме того, возможность указать в одной команде `ALTER FOREIGN TABLE` несколько операций так же является расширением.

`ALTER FOREIGN TABLE DROP COLUMN` позволяет удалить единственный столбец сторонней таблицы и оставить таблицу без столбцов. Это является расширением стандарта SQL, который не допускает существование сторонних таблиц с нулём столбцов.

См. также

CREATE FOREIGN TABLE, DROP FOREIGN TABLE

ALTER FUNCTION

ALTER FUNCTION — изменить определение функции

Синтаксис

```
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    действие [ ... ] [ RESTRICT ]
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    RENAME TO новое_имя
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    SET SCHEMA новая_схема
ALTER FUNCTION имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION имя_расширения

Где действие может быть следующим:

    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE
    [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST стоимость_выполнения
    ROWS строк_в_результате
    SUPPORT вспомогательная_функция
    SET параметр_конфигурации { TO | = } { значение | DEFAULT }
    SET параметр_конфигурации FROM CURRENT
    RESET параметр_конфигурации
    RESET ALL
```

Описание

`ALTER FUNCTION` изменяет определение функции.

Выполнить `ALTER FUNCTION` может только владелец соответствующей функции. Чтобы сменить схему функции, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме функции. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать функцию. Однако суперпользователь может сменить владельца функции в любом случае.)

Параметры

имя

Имя существующей функции (возможно, дополненное схемой). Если список аргументов не указан, это имя должно быть уникальным в схеме.

режим_аргумента

Режим аргумента: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `ALTER FUNCTION` не учитывает аргументы `OUT`, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы `IN`, `INOUT` и `VARIADIC`.

`имя_аргумента`

Имя аргумента. Заметьте, что на самом деле `ALTER FUNCTION` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

`тип_аргумента`

Тип данных аргументов функции (возможно, дополненный именем схемы), если таковые имеются.

`новое_имя`

Новое имя функции.

`новый_владелец`

Новый владелец функции. Заметьте, что если функция помечена как `SECURITY DEFINER`, в дальнейшем она будет выполняться от имени нового владельца.

`новая_схема`

Новая схема функции.

`DEPENDS ON EXTENSION` *`имя_расширения`*
`NO DEPENDS ON EXTENSION` *`имя_расширения`*

Эта форма делает функцию зависимой от расширения или, наоборот, удаляет эту зависимость, если указывается `NO`. Функция, помеченная как зависимая от расширения, удаляется при удалении расширения даже без указания `CASCADE`. Если функция зависит от нескольких расширений, она удаляется при удалении любого из этих расширений.

`CALLED ON NULL INPUT`
`RETURNS NULL ON NULL INPUT`
`STRICT`

`CALLED ON NULL INPUT` меняет функцию так, чтобы она вызывалась, когда некоторые или все её аргументы равны NULL. `RETURNS NULL ON NULL INPUT` или `STRICT` меняет функцию так, чтобы она не вызывалась, когда некоторые или все её аргументы равны NULL, а вместо вызова автоматически выдавался результат NULL. За подробностями обратитесь к CREATE FUNCTION.

`IMMUTABLE`
`STABLE`
`VOLATILE`

Устанавливает заданный вариант изменчивости функции. Подробнее это описано в CREATE FUNCTION.

`[EXTERNAL] SECURITY INVOKER`
`[EXTERNAL] SECURITY DEFINER`

Устанавливает, является ли функция определяющей контекст безопасности. Ключевое слово `EXTERNAL` игнорируется для соответствия стандарту SQL. Подробнее это свойство описано в CREATE FUNCTION.

`PARALLEL`

Устанавливает, будет ли функция считаться безопасной для распараллеливания. Подробнее это описано в CREATE FUNCTION.

`LEAKPROOF`

Устанавливает, является ли функция герметичной. Подробнее это свойство описано в CREATE FUNCTION.

`COST` *`стоимость_выполнения`*

Изменяет ориентировочную стоимость выполнения функции. Подробнее это описывается в CREATE FUNCTION.

`ROWS` *`строк_в_результате`*

Изменяет ориентировочное число строк в результате функции, возвращающей множество. Подробнее это описывается в CREATE FUNCTION.

`SUPPORT` *`вспомогательная_функция`*

Задаёт или меняет вспомогательную функцию для планировщика, которая будет использоваться с этой функцией. За подробностями обратитесь к Разделу 36.11. Для использования этого указания нужно быть суперпользователем.

Имя новой вспомогательной функции является обязательным, поэтому данное указание не позволяет полностью отказаться от использования вспомогательной функции. Если вам требуется это, воспользуйтесь командой `CREATE OR REPLACE FUNCTION`.

`параметр_конфигурации`
`значение`

Добавляет или изменяет установку параметра конфигурации, выполняемую при вызове функции. Если задано *`значение`* `DEFAULT` или, что равнозначно, выполняется действие `RESET`, локальное переопределение для функции удаляется и функция выполняется со значением, установленным в окружении. Для удаления всех установок параметров для данной функции укажите `RESET ALL`. `SET FROM CURRENT` устанавливает для последующих вызовов функции значение параметра, действующее в момент выполнения `ALTER PROCEDURE`.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

`RESTRICT`

Игнорируется для соответствия стандарту SQL.

Примеры

Переименование функции `sqrt` для типа `integer` в `square_root`:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

Смена владельца функции `sqrt` для типа `integer` на `joe`:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

Смена схемы функции `sqrt` для типа `integer` на `maths`:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

Обозначение функции `sqrt` для типа `integer` как зависимой от расширения `mathlib`:

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION mathlib;
```

Изменение пути поиска, который устанавливается автоматически для функции:

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

Отмена автоматического определения `search_path` для функции:

```
ALTER FUNCTION check_password(text) RESET search_path;
```

Теперь функция будет выполняться с тем путём, который задан в момент вызова.

Совместимость

Этот оператор частично совместим с оператором `ALTER FUNCTION` в стандарте SQL. Стандарт позволяет менять больше свойств функции, но не позволяет переименовывать функции, переключать контекст безопасности, связывать с функциями значения параметров конфигурации, а также менять владельца, схему и тип изменчивости функции. Также в стандарте слово `RESTRICT` считается обязательным, тогда как в Postgres Pro оно не требуется.

См. также

CREATE FUNCTION, DROP FUNCTION, ALTER PROCEDURE, ALTER ROUTINE

ALTER GROUP

ALTER GROUP — изменить имя роли или членство

Синтаксис

```
ALTER GROUP указание_роли ADD USER имя_пользователя [, ... ]
ALTER GROUP указание_роли DROP USER имя_пользователя [, ... ]

Здесь указание_роли:

    имя_роли
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER

ALTER GROUP имя_группы RENAME TO новое_имя
```

Описание

`ALTER GROUP` изменяет атрибуты группы пользователей. Эта команда считается устаревшей, хотя и поддерживается для обратной совместимости, так как группы (и пользователи) были заменены более общей концепцией ролей.

Первые две формы добавляют пользователей в группу или удаляют их из группы. (В данном случае в качестве «пользователя» или «группы» может фигурировать любая роль.) По сути они равнозначны командам, разрешающим/запрещающим членство в роли «группа»; поэтому вместо них рекомендуется использовать `GRANT` и `REVOKE`. Обратите внимание, что для этой команды нет дополнительных указаний, как для `GRANT` и `REVOKE`, например, дающих возможность выдавать или забирать право `ADMIN OPTION`, а также возможность указывать праводателя.

Третья форма меняет имя группы. Она в точности равнозначна команде `ALTER ROLE`, выполняющей переименование роли.

Параметры

`имя_группы`

Имя изменяемой группы (роли).

`имя_пользователя`

Пользователи (роли), добавляемые или исключаемые из группы. Эти пользователи должны уже существовать; `ALTER GROUP` не создаёт и не удаляет пользователей.

`новое_имя`

Новое имя группы.

Примеры

Добавление пользователей в группу:

```
ALTER GROUP staff ADD USER karl, john;
```

Удаление пользователей из группы:

```
ALTER GROUP workers DROP USER beth;
```

Совместимость

Оператор `ALTER GROUP` отсутствует в стандарте SQL.

См. также

GRANT, REVOKE, ALTER ROLE

ALTER INDEX

ALTER INDEX — изменить определение индекса

Синтаксис

```
ALTER INDEX [ IF EXISTS ] имя RENAME TO новое_имя
ALTER INDEX [ IF EXISTS ] имя SET TABLESPACE табл_пространство
ALTER INDEX имя ATTACH PARTITION имя_индекса
ALTER INDEX имя [ NO ] DEPENDS ON EXTENSION имя_расширения
ALTER INDEX [ IF EXISTS ] имя SET ( параметр_хранения [= значение] [, ... ] )
ALTER INDEX [ IF EXISTS ] имя RESET ( параметр_хранения [, ... ] )
ALTER INDEX [ IF EXISTS ] имя ALTER [ COLUMN ] номер_столбца
    SET STATISTICS integer
ALTER INDEX ALL IN TABLESPACE имя [ OWNED BY имя_роли [, ... ] ]
    SET TABLESPACE новое_табл_пространство [ NOWAIT ]
```

Описание

`ALTER INDEX` меняет определение существующего индекса. Несколько её разновидностей описаны ниже. Заметьте, что для разных разновидностей могут требоваться разные уровни блокировок. Если явно не отмечено другое, требуется блокировка `ACCESS EXCLUSIVE`. При перечислении нескольких подкоманд будет запрашиваться самая сильная блокировка из требуемых ими.

`RENAME`

Форма `RENAME` меняет имя индекса. Если этот индекс связан с ограничением таблицы (`UNIQUE`, `PRIMARY KEY` или `EXCLUDE`), это ограничение тоже переименовывается. На сохранённые данные это не влияет.

Для переименования индекса требуется блокировка `SHARE UPDATE EXCLUSIVE`.

`SET TABLESPACE`

Эта форма меняет табличное пространство индекса на заданное и переносит в него файл(ы) данных, связанные с индексом. Для изменения табличного пространства индекса нужно быть владельцем индекса и иметь право `CREATE` в новом табличном пространстве. Форма `ALL IN TABLESPACE` позволяет перенести из заданного пространства все индексы в текущей базе данных, блокируя их для перемещения и затем перемещая каждый индекс. Эта форма также поддерживает указание `OWNED BY`, с которым будут перемещены только индексы, принадлежащие заданным ролям. Если указан параметр `NOWAIT`, команда завершится ошибкой, если не сможет немедленно получить все требуемые блокировки. Заметьте, что эта команда не переместит системные каталоги; вместо неё следует использовать `ALTER DATABASE` или явные вызовы `ALTER INDEX`. См. также описание `CREATE TABLESPACE`.

`ATTACH PARTITION` *`имя_индекса`*

Эта форма присоединяет указанный индекс (возможно, дополненный схемой) к изменяемому. Указанный индекс должен относиться к секции таблицы, содержащей изменяемый индекс, и иметь такое же определение. Присоединённый индекс не может быть удалён независимо, но будет удалён автоматически при удалении родительского индекса.

`DEPENDS ON EXTENSION` *`имя_расширения`*
`NO DEPENDS ON EXTENSION` *`имя_расширения`*

Эта форма делает индекс зависимым от расширения или, наоборот, удаляет эту зависимость, если указывается `NO`. Индекс, помеченный как зависимый от расширения, автоматически удаляется при удалении расширения.

`SET ( параметр_хранения [= значение] [, ... ] )`

Эта форма настраивает один или несколько специфичных для индекса параметров хранения. Список доступных параметров приведён в описании `CREATE INDEX`. Заметьте, что эта команда не меняет содержимое индекса немедленно; для получения желаемого эффекта в зависимости от параметров может потребоваться перестроить индекс командой `REINDEX`.

`RESET ( параметр_хранения [, ... ] )`

Эта форма сбрасывает один или несколько специфичных для индекса параметров хранения к значениям по умолчанию. Как и с `SET`, для полного обновления индекса может потребоваться выполнить `REINDEX`.

`ALTER [ COLUMN ] номер_столбца SET STATISTICS integer`

Эта форма задаёт ориентир сбора статистики по столбцу для последующих операций `ANALYZE`, хотя её можно использовать только для индексируемых столбцов, заданных в виде выражений. Так как у выражений нет уникальных имён, мы обращаемся к ним по порядковым номерам столбцов в индексе. Диапазон допустимых значений ориентира: 0..10000; при -1 применяется системное значение по умолчанию (default_statistics_target). За дополнительными сведениями об использовании статистики планировщиком запросов Postgres Pro обратитесь к Разделу 14.2.

Параметры

`IF EXISTS`

Не считать ошибкой, если индекс не существует. В этом случае будет выдано замечание.

номер_столбца

Число, указывающее на номер столбца в индексе по порядку (слева направо).

имя

Имя (возможно, дополненное схемой) существующего индекса, подлежащего изменению.

новое_имя

Новое имя индекса.

табл_пространство

Табличное пространство, в которое будет перемещён индекс.

имя_расширения

Имя расширения, от которого будет зависеть индекс.

параметр_хранения

Имя специфичного для индекса параметра хранения.

значение

Новое значение специфичного для индекса параметра хранения. Это может быть число или строка, в зависимости от параметра.

Примечания

Эти операции также возможно выполнить с помощью `ALTER TABLE`. На самом деле `ALTER INDEX` — это просто синоним форм `ALTER TABLE`, работающих с индексами.

Ранее существовала форма `ALTER INDEX OWNER`, но сейчас она игнорируется (с предупреждением). Владельцем индекса может быть только владелец соответствующей таблицы. При смене владельца таблицы владелец индекса меняется автоматически.

Какие-либо изменения индексов системного каталога не допускаются.

Примеры

Переименование существующего индекса:

```
ALTER INDEX distributors RENAME TO suppliers;
```

Перемещение индекса в другое табличное пространство:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

Изменение фактора заполнения индекса (предполагается, что это поддерживает метод индекса):

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

Устанавливает ориентир сбора статистики для индекса по выражению:

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

Совместимость

`ALTER INDEX` является расширением Postgres Pro.

См. также

CREATE INDEX, REINDEX

ALTER LANGUAGE

ALTER LANGUAGE — изменить определение процедурного языка

Синтаксис

```
ALTER [ PROCEDURAL ] LANGUAGE имя RENAME TO новое_имя
ALTER [ PROCEDURAL ] LANGUAGE имя OWNER TO { новый_владелец | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
```

Описание

`ALTER LANGUAGE` изменяет определение процедурного языка. Единственное, что может это команда — переименовать язык или назначить нового владельца. Выполнить `ALTER LANGUAGE` может только суперпользователь или владелец языка.

Параметры

имя

Имя языка

новое_имя

Новое имя языка

новый_владелец

Новый владелец языка

Совместимость

Оператор `ALTER LANGUAGE` отсутствует в стандарте SQL.

См. также

CREATE LANGUAGE, DROP LANGUAGE

ALTER LARGE OBJECT

ALTER LARGE OBJECT — изменить определение большого объекта

Синтаксис

```
ALTER LARGE OBJECT oid_большого_объекта OWNER TO { новый_владелец | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
```

Описание

`ALTER LARGE OBJECT` изменяет определение большого объекта.

Выполнить `ALTER LARGE OBJECT` может только владелец большого объекта. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца. (Однако суперпользователь может изменять свойства больших объектов в любом случае.) В настоящее время единственное возможное изменение заключается в назначении нового владельца, так что всегда действуют оба ограничения.

Параметры

`oid_большого_объекта`

OID изменяемого большого объекта

`новый_владелец`

Новый владелец большого объекта

Совместимость

Оператор `ALTER LARGE OBJECT` отсутствует в стандарте SQL.

См. также

Глава 33

ALTER MATERIALIZED VIEW

ALTER MATERIALIZED VIEW — изменить определение материализованного представления

Синтаксис

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
    действие [, ... ]
ALTER MATERIALIZED VIEW имя
    [ NO ] DEPENDS ON EXTENSION имя_расширения
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
    RENAME TO новое_имя
ALTER MATERIALIZED VIEW [ IF EXISTS ] имя
    SET SCHEMA новая_схема
ALTER MATERIALIZED VIEW ALL IN TABLESPACE имя [ OWNED BY имя_роли [, ... ] ]
    SET TABLESPACE новое_табл_пространство [ NOWAIT ]

Где действие может быть следующим:

    ALTER [ COLUMN ] имя_столбца SET STATISTICS integer
    ALTER [ COLUMN ] имя_столбца SET ( атрибут = значение [, ... ] )
    ALTER [ COLUMN ] имя_столбца RESET ( атрибут [, ... ] )
    ALTER [ COLUMN ] имя_столбца SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN |
 DEFAULT }
    ALTER [ COLUMN ] имя_столбца SET COMPRESSION метод_сжатия
    CLUSTER ON имя_индекса
    SET WITHOUT CLUSTER
    SET ACCESS METHOD новый_метод_доступа
    SET TABLESPACE новое_табл_пространство
    SET ( параметр_хранения [= значение] [, ... ] )
    RESET ( параметр_хранения [, ... ] )
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

Описание

`ALTER MATERIALIZED VIEW` изменяет различные расширенные свойства существующего материализованного представления.

Выполнить `ALTER MATERIALIZED VIEW` может только владелец материализованного представления. Чтобы сменить схему материализованного представления, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме материализованного представления. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать материализованное представление. Однако суперпользователь может сменить владельца материализованного представления в любом случае.)

Подвиды и действия оператора `ALTER MATERIALIZED VIEW` являются подмножеством тех, что относятся к команде `ALTER TABLE`, и имеют то же значение применительно к материализованным представлениям. За подробностями обратитесь к описанию `ALTER TABLE`.

Параметры

имя

Имя существующего материализованного представления (возможно, дополненное схемой).

имя_столбца

Имя существующего столбца.

имя_расширения

Имя расширения, от которого будет зависеть материализованное представление (или не будет, если указано `NO`). Материализованное представление, помеченное как зависимое от расширения, автоматически удаляется при удалении расширения.

новое_имя_столбца

Новое имя существующего столбца.

новый_владелец

Имя пользователя, назначаемого новым владельцем материализованного представления.

новое_имя

Новое имя материализованного представления.

новая_схема

Новая схема материализованного представления.

Примеры

Переименование материализованного представления `foo` в `bar`:

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

Совместимость

`ALTER MATERIALIZED VIEW` является расширением Postgres Pro.

См. также

CREATE MATERIALIZED VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

ALTER OPERATOR

ALTER OPERATOR — изменить определение оператора

Синтаксис

```
ALTER OPERATOR имя ( { тип_слева | NONE } , тип_справа )
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }

ALTER OPERATOR имя ( { тип_слева | NONE } , тип_справа )
    SET SCHEMA новая_схема

ALTER OPERATOR имя ( { тип_слева | NONE } , тип_справа )
    SET ( {  RESTRICT = { процедура_ограничения | NONE }
           | JOIN = { процедура_соединения | NONE }
           | COMMUTATOR = коммут_оператор
           | NEGATOR = обратный_оператор
           | HASHES
           | MERGES
          } [, ... ] )
```

Описание

`ALTER OPERATOR` изменяет определение оператора.

Выполнить `ALTER OPERATOR` может только владелец соответствующего оператора. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме оператора. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать оператор. Однако суперпользователь может сменить владельца оператора в любом случае.)

Параметры

имя

Имя существующего оператора (возможно, дополненное схемой).

тип_слева

Тип данных левого операнда оператора; если у оператора нет левого операнда, укажите `NONE`.

тип_справа

Тип данных правого операнда оператора.

новый_владелец

Новый владелец оператора.

новая_схема

Новая схема оператора.

процедура_ограничения

Функция оценки избирательности ограничения для данного оператора; значение NONE удаляет существующую функцию оценки.

процедура_соединения

Функция оценки избирательности соединения для этого оператора; значение NONE удаляет существующую функцию оценки.

коммут_оператор

Оператор, коммутирующий для данного. Можно изменить, только если у данного оператора коммутирующий отсутствует.

обратный_оператор

Оператор, обратный к данному. Можно изменить, только если у данного оператора обратный отсутствует.

`HASHES`

Показывает, что этот оператор поддерживает соединение по хешу. Можно только включить, но не отключить.

`MERGES`

Показывает, что этот оператор поддерживает соединение слиянием. Можно только включить, но не отключить.

Примечания

Более подробно информация описана в Разделе 36.14 и Разделе 36.15.

Поскольку коммутирующие операторы связаны парами и являются коммутирующими друг для друга, команда `ALTER OPERATOR SET COMMUTATOR` также назначит данный оператор коммутирующим для оператора из параметра *коммут_оператор*. Аналогично команда `ALTER OPERATOR SET NEGATOR` назначит данный оператор обратным для оператора из параметра *обратный_оператор*. Таким образом, для выполнения команды необходимо являться владельцем и данного оператора, и назначаемого коммутирующего или обратного оператора.

Примеры

Смена владельца нестандартного оператора `a @@ b` для типа `text`:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Смена функций оценки избирательности ограничения и соединения для нестандартного оператора `a && b` для типа `int[]`:

```
ALTER OPERATOR && (int[], int[]) SET (RESTRICT = _int_contsel, JOIN =
 _int_contjoinsel);
```

Пометка оператора `&&` как коммутирующего для самого себя:

```
ALTER OPERATOR && (int[], int[]) SET (COMMUTATOR = &&);
```

Совместимость

Команда `ALTER OPERATOR` отсутствует в стандарте SQL.

См. также

CREATE OPERATOR, DROP OPERATOR

ALTER OPERATOR CLASS

ALTER OPERATOR CLASS — изменить определение класса операторов

Синтаксис

```
ALTER OPERATOR CLASS имя USING индексный_метод
    RENAME TO новое_имя

ALTER OPERATOR CLASS имя USING индексный_метод
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }

ALTER OPERATOR CLASS имя USING индексный_метод
    SET SCHEMA новая_схема
```

Описание

`ALTER OPERATOR CLASS` изменяет определение класса операторов.

Выполнить `ALTER OPERATOR CLASS` может только владелец соответствующего класса операторов. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме класса операторов. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать класс операторов. Однако суперпользователь может сменить владельца классов операторов в любом случае.)

Параметры

имя

Имя существующего класса операторов (возможно, дополненное схемой).

индексный_метод

Имя индексного метода, для которого предназначен этот класс операторов.

новое_имя

Новое имя класса операторов.

новый_владелец

Новый владелец класса операторов.

новая_схема

Новая схема класса операторов.

Совместимость

Команда `ALTER OPERATOR CLASS` отсутствует в стандарте SQL.

См. также

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY — изменить определение семейства операторов

Синтаксис

```
ALTER OPERATOR FAMILY имя USING индексный_метод ADD
  {  OPERATOR номер_стратегии имя_оператора ( тип_операнда, тип_операнда )
              [ FOR SEARCH | FOR ORDER BY семейство_сортировки ]
   | FUNCTION номер_опорной_функции [ ( тип_операнда [ , тип_операнда ] ) ]
              имя_функции [ ( тип_аргумента [, ...] ) ]
  } [, ... ]

ALTER OPERATOR FAMILY имя USING индексный_метод DROP
  {  OPERATOR номер_стратегии ( тип_операнда [ , тип_операнда ] )
   | FUNCTION номер_опорной_функции ( тип_операнда [ , тип_операнда ] )
  } [, ... ]

ALTER OPERATOR FAMILY имя USING индексный_метод
    RENAME TO новое_имя

ALTER OPERATOR FAMILY имя USING индексный_метод
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }

ALTER OPERATOR FAMILY имя USING индексный_метод
    SET SCHEMA новая_схема
```

Описание

`ALTER OPERATOR FAMILY` меняет определение семейства операторов. Она позволяет добавлять в семейство операторы и опорные функции, удалять их из семейства или менять имя и владельца семейства операторов.

Когда операторы и опорные функции добавляются в семейство с помощью `ALTER OPERATOR FAMILY`, они не становятся частью какого-либо определённого класса операторов в семействе, а просто считаются «слабосвязанными» с семейством. Это показывает, что эти операторы и функции семантически совместимы с семейством, но не требуются для корректной работы какого-либо индекса. (Операторы и функции, которые действительно требуются для этого, должны быть включены не в семейство, а в класс операторов; см. CREATE OPERATOR CLASS.) Postgres Pro позволяет удалять слабосвязанные члены из семейства в любое время, но члены класса операторов не могут быть удалены, пока не будет удалён весь класс и все зависимые от него индексы. Обычно в классы операторов включаются операторы и функции, работающие с одним типом данным (так как они нужны для поддержки индексов данных такого типа), а межтиповые функции и операторы становятся слабосвязанными членами семейства.

Выполнить `ALTER OPERATOR FAMILY` может только суперпользователь. (Это ограничение введено потому, что ошибочное определение семейства операторов может вызвать нарушения или даже сбой в работе сервера.)

`ALTER OPERATOR FAMILY` в настоящее время не проверяет, включает ли определение семейства операторов все операторы и функции, требуемые для индексного метода, и образуют ли они целостный набор. Ответственность за правильность определения семейства лежит на пользователе.

За дополнительными сведениями обратитесь к Разделу 36.16.

Параметры

имя

Имя существующего семейства операторов (возможно, дополненное схемой).

индексный_метод

Имя индексного метода, для которого предназначено это семейство операторов.

номер_стратегии

Номер стратегии индексного метода для оператора, связанного с данным семейством операторов.

имя_оператора

Имя (возможно, дополненное схемой) оператора, связанного с данным семейством операторов.

тип_операнда

В предложении `OPERATOR` указывается тип(ы) данных оператора или `NONE`, если это префиксный оператор. В отличие от похожего синтаксиса в `CREATE OPERATOR CLASS`, здесь типы операндов должны указываться всегда.

В предложении `ADD FUNCTION` это тип данных, который должна поддерживать эта функция, если он отличается от входного типа данных функции. Для функций сравнения B-деревьев и хеш-функций указывать *тип_операнда* необязательно, так как их входные типы данных всегда будут подходящими. Однако для опорных функций сортировки и функций равенства образов в B-деревьях и всех функций в классах операторов GiST, SP-GiST и GIN необходимо указать тип(ы) операндов, с которыми будут использоваться эти функции.

В предложении `DROP FUNCTION` тип операнда, который должна поддерживать эта функция.

семейство_сортировки

Имя (возможно, дополненное схемой) существующего семейства операторов `btree`, описывающего порядок сортировки, связанный с оператором сортировки.

Если не указано ни `FOR SEARCH` (для поиска), ни `FOR ORDER BY` (для сортировки), подразумевается `FOR SEARCH`.

номер_опорной_функции

Номер опорной функции индексного метода для функции, связанной с данным семейством операторов.

имя_функции

Имя (возможно, дополненное схемой) функции, которая является опорной функцией индексного метода для данного семейства операторов. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

тип_аргумента

Тип данных параметра функции.

новое_имя

Новое имя семейства операторов.

новый_владелец

Новый владелец семейства операторов.

новая_схема

Новая схема семейства операторов.

Предложения `OPERATOR` и `FUNCTION` могут указываться в любом порядке.

Примечания

Заметьте, что в синтаксисе `DROP` указывается только «слот» в семействе операторов, по номеру стратегии или опорной функции, и входные типы данных. Имя оператора или функции, занимающих этот слот, не упоминается. Также учтите, что в `DROP FUNCTION` указываются типы входных данных, которые должна поддерживать функция, но для индексов GiST, SP-GiST и GIN они могут не иметь ничего общего с типами фактических аргументов функции.

Так как механизмы индексов не проверяют права доступа к функциям прежде чем вызывать их, включение функций или операторов в семейство операторов по сути даёт всем право на выполнение их. Обычно это не проблема для таких функций, какие бывают полезны в семействе операторов.

Операторы не должны реализовываться в функциях на языке SQL. SQL-функция вероятнее всего будет встроена в вызывающий запрос, что помешает оптимизатору понять, что этот запрос соответствует индексу.

До PostgreSQL 8.4 предложение `OPERATOR` могло включать указание `RECHECK`. Теперь это не поддерживается, так как оператор индекса может быть «неточным» и это определяется на ходу в момент выполнения. Это позволяет эффективно справляться с ситуациями, когда оператор может быть или не быть неточным.

Примеры

Следующий пример добавляет опорные функции и межтиповые операторы в семейство операторов, уже содержащее классы операторов B-дерева для типов данных `int4` и `int2`.

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 и int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 и int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

Удаление этих же элементов:

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
```

```
OPERATOR 3 (int2, int4) ,
OPERATOR 4 (int2, int4) ,
OPERATOR 5 (int2, int4) ,
FUNCTION 1 (int2, int4) ;
```

Совместимость

Команда `ALTER OPERATOR FAMILY` отсутствует в стандарте SQL.

См. также

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

ALTER POLICY

ALTER POLICY — изменить определение политики защиты на уровне строк

Синтаксис

```
ALTER POLICY имя ON имя_таблицы RENAME TO новое_имя

ALTER POLICY имя ON имя_таблицы
    [ TO { имя_роли | PUBLIC | CURRENT_ROLE | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( выражение_использования ) ]
    [ WITH CHECK ( выражение_проверки ) ]
```

Описание

`ALTER POLICY` изменяет определение существующей политики на уровне строк. Заметьте, что `ALTER POLICY` позволяет изменить только набор ролей, для которых применяется политика, и выражения `USING` и `WITH CHECK`. Чтобы изменить другие свойства политики, например команду, к которой она применяется, а также характеристику разрешительная/ограничительная, политику надо удалить и создать заново.

Использовать `ALTER POLICY` может только владелец таблицы (или представления), к которой применяется эта политика.

Во второй форме `ALTER POLICY` список ролей, *`выражение_использования`* и *`выражение_проверки`* заменяются независимо, если они указаны. Когда одно из этих предложений опущено, соответствующая часть политики остаётся неизменной.

Параметры

`имя`

Имя существующей политики, подлежащей изменению.

`имя_таблицы`

Имя таблицы (возможно, дополненное схемой), к которой применяется эта политика.

`новое_имя`

Новое имя политики.

`имя_роли`

Роль (роли), на которую действует политика. В одной команде можно указать несколько ролей. Чтобы применить политику ко всем ролям, укажите `PUBLIC`.

`выражение_использования`

Выражение `USING` для политики. За подробностями обратитесь к CREATE POLICY.

`выражение_проверки`

Выражение `WITH CHECK` для политики. За подробностями обратитесь к CREATE POLICY.

Совместимость

`ALTER POLICY` является расширением Postgres Pro.

См. также

CREATE POLICY, DROP POLICY

ALTER PROCEDURE

ALTER PROCEDURE — изменить определение процедуры

Синтаксис

```
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    действие [ ... ] [ RESTRICT ]
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    RENAME TO новое_имя
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    SET SCHEMA новая_схема
ALTER PROCEDURE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION имя_расширения

Где действие может быть следующим:

    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    SET параметр_конфигурации { TO | = } { значение | DEFAULT }
    SET параметр_конфигурации FROM CURRENT
    RESET параметр_конфигурации
    RESET ALL
```

Описание

`ALTER PROCEDURE` изменяет определение процедуры.

Выполнить `ALTER PROCEDURE` может только владелец процедуры. Чтобы сменить схему процедуры, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме представления. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать процедуру. Однако суперпользователь может сменить владельца процедуры в любом случае.)

Параметры

имя

Имя существующей процедуры (возможно, дополненное схемой). Если список аргументов не указан, имя процедуры должно быть уникальным в её схеме.

режим_аргумента

Режим аргумента: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле `ALTER PROCEDURE` не обращает внимание на имена аргументов, так как для идентификации процедуры используются только их типы.

тип_аргумента

Тип данных аргументов процедуры (возможно, дополненный именем схемы), если таковые имеются. Подробнее о том, как находится нужная процедура по типам аргументов, рассказывается в описании DROP PROCEDURE.

новое_имя

Новое имя процедуры.

новый_владелец

Новый владелец процедуры. Заметьте, что если процедура помечена как `SECURITY DEFINER`, в дальнейшем она будет выполняться от имени нового владельца.

новая_схема

Новая схема процедуры.

имя_расширения

Эта форма делает процедуру зависимой от расширения или, наоборот, удаляет эту зависимость, если указывается `NO`. Процедура, помеченная как зависимая от расширения, удаляется при удалении расширения даже без указания `CASCADE`. Если процедура зависит от нескольких расширений, она удаляется при удалении любого из этих расширений.

`[EXTERNAL] SECURITY INVOKER`
`[EXTERNAL] SECURITY DEFINER`

Устанавливает, является ли процедура определяющей контекст безопасности. Ключевое слово `EXTERNAL` игнорируется для соответствия стандарту SQL. Подробнее это свойство описано в CREATE PROCEDURE.

параметр_конфигурации
значение

Добавляет или изменяет установку параметра конфигурации, выполняемую при вызове процедуры. Если задано *значение* `DEFAULT` или, что равнозначно, выполняется действие `RESET`, локальное переопределение для процедуры удаляется и процедура выполняется со значением, установленным в окружении. Для удаления всех установок параметров для данной процедуры укажите `RESET ALL`. `SET FROM CURRENT` устанавливает для последующих вызовов процедуры значение параметра, действующее в момент выполнения `ALTER PROCEDURE`.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

`RESTRICT`

Игнорируется для соответствия стандарту SQL.

Примеры

Переименование процедуры `insert_data` с двумя аргументами типа `integer` в `insert_record`:

```
ALTER PROCEDURE insert_data(integer, integer) RENAME TO insert_record;
```

Смена владельца процедуры `insert_data` с двумя аргументами типа `integer` на `joe`:

```
ALTER PROCEDURE insert_data(integer, integer) OWNER TO joe;
```

Смена схемы процедуры `insert_data` с двумя аргументами типа `integer` на `accounting`:

```
ALTER PROCEDURE insert_data(integer, integer) SET SCHEMA accounting;
```

Обозначение процедуры `insert_data(integer, integer)` как зависимой от расширения `myext`:

```
ALTER PROCEDURE insert_data(integer, integer) DEPENDS ON EXTENSION myext;
```

Изменение пути поиска, который устанавливается автоматически для процедуры:

```
ALTER PROCEDURE check_password(text) SET search_path = admin, pg_temp;
```

Отмена автоматического определения `search_path` для процедуры:

```
ALTER PROCEDURE check_password(text) RESET search_path;
```

Теперь процедура будет выполняться с тем путём, который задан в момент вызова.

Совместимость

Этот оператор частично совместим с оператором `ALTER PROCEDURE` в стандарте SQL. Стандарт позволяет изменить больше свойств процедуры, но не даёт возможности переименовать процедуру, сделать процедуру определяющей контекст безопасности, связать с процедурой значения параметров конфигурации или изменить владельца, схему или характеристику изменчивости процедуры. Также стандарт требует наличия ключевого слова `RESTRICT`, но в Postgres Pro оно необязательное.

См. также

CREATE PROCEDURE, DROP PROCEDURE, ALTER FUNCTION, ALTER ROUTINE

ALTER PROFILE

ALTER PROFILE — изменить профиль аутентификации

Синтаксис

```
ALTER PROFILE имя [ LIMIT параметр значение [ ... ] ]

Здесь параметр:

      FAILED_LOGIN_ATTEMPTS
    | PASSWORD_REUSE_TIME
    | PASSWORD_REUSE_MAX
    | PASSWORD_LIFE_TIME
    | PASSWORD_GRACE_TIME
    | USER_INACTIVE_TIME
    | FAILED_AUTH_KEEP_TIME
    | PASSWORD_MIN_UNIQUE_CHARS
    | PASSWORD_MIN_LEN
    | PASSWORD_REQUIRE_COMPLEX

ALTER PROFILE имя RENAME TO новое_имя
```

Описание

Команда `ALTER PROFILE` изменяет параметры профиля Postgres Pro. Выполнять эту команду разрешено только суперпользователям базы данных или пользователям с правами роли `pg_manage_profiles`.

Параметры

`имя`

Имя профиля, параметры которого изменяются.

`FAILED_LOGIN_ATTEMPTS` *`значение`*
`PASSWORD_REUSE_TIME` *`значение`*
`PASSWORD_REUSE_MAX` *`значение`*
`PASSWORD_LIFE_TIME` *`значение`*
`PASSWORD_GRACE_TIME` *`значение`*
`USER_INACTIVE_TIME` *`значение`*
`FAILED_AUTH_KEEP_TIME` *`значение`*
`PASSWORD_MIN_UNIQUE_CHARS` *`значение`*
`PASSWORD_MIN_LEN` *`значение`*
`PASSWORD_REQUIRE_COMPLEX` *`значение`*

Эти предложения меняют параметры, изначально установленные командой CREATE PROFILE. За дополнительными сведениями обратитесь к странице справки `CREATE PROFILE`.

`новое_имя`

Новое имя профиля.

Замечания

Для добавления нового профиля используйте команду CREATE PROFILE, а для удаления — DROP PROFILE.

Примеры

Изменение параметров `PASSWORD_LIFE_TIME` и `PASSWORD_GRACE_TIME` в профиле:

```
ALTER PROFILE admin_profile
    LIMIT PASSWORD_LIFE_TIME 90
          PASSWORD_GRACE_TIME 3;
```

См. также

CREATE PROFILE, DROP PROFILE, CREATE ROLE

ALTER PUBLICATION

ALTER PUBLICATION — изменить определение публикации

Синтаксис

```
ALTER PUBLICATION имя ADD объект_публикации [, ...]
ALTER PUBLICATION имя SET объект_публикации [, ...]
ALTER PUBLICATION имя DROP объект_публикации [, ...]
ALTER PUBLICATION имя SET ( параметр_публикации [= значение] [, ... ] )
ALTER PUBLICATION имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER PUBLICATION имя RENAME TO новое_имя

где объект_публикации может быть следующим:

    TABLE [ ONLY ] имя_таблицы [ * ] [ ( имя_столбца [, ... ] ) ] [ WHERE ( выражение
 ) ] [, ... ]
    TABLES IN SCHEMA { имя_схемы | CURRENT_SCHEMA } [, ... ]
```

Описание

Команда `ALTER PUBLICATION` может изменять атрибуты публикации.

Первые три формы управляют вхождением таблиц/схем в публикации. Предложение `SET` заменяет список таблиц/схем в публикации заданным списком; таблицы/схемы, входившие в публикацию ранее, будут удалены из неё. Предложения `ADD` и `DROP` добавляют и удаляют таблицы/схемы в публикации соответственно. Заметьте, что при добавлении таблиц/схем в публикацию, на которую уже оформлена подписка, необходимо выполнить `ALTER SUBSCRIPTION ... REFRESH PUBLICATION` на стороне подписчика, чтобы это изменение вступило в силу. Также заметьте, что команда `DROP TABLES IN SCHEMA` не удаляет относящиеся к заданной схеме таблицы, включённые в публикацию указаниями `FOR TABLE`/ `ADD TABLE`, а с предложением `DROP` нельзя использовать предложение `WHERE`.

Четвёртая форма этой команды, показанная в сводке синтаксиса, может изменять все свойства публикации, заданные в CREATE PUBLICATION. Свойства, которые не упоминаются в этой команде, сохраняют предыдущие значения.

Остальные формы команды меняют владельца и имя публикации.

Для выполнения `ALTER PUBLICATION` необходимо владеть данной публикацией. Чтобы добавить таблицу в публикацию, дополнительно нужно быть владельцем этой таблицы. Для выполнения `ADD TABLES IN SCHEMA` и `SET TABLES IN SCHEMA` требуется быть суперпользователем. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в базе данных. Кроме того, новым владельцем публикации вида `FOR ALL TABLES` или `FOR TABLES IN SCHEMA` должен быть суперпользователь. Однако суперпользователь может менять владельца публикации вне зависимости от этих ограничений.

Добавление/указание схемы, когда публикация также публикует таблицу со списком столбцов, равно как и обратная операция, не поддерживается.

Параметры

имя

Имя существующей публикации, определение которой изменяется.

имя_таблицы

Имя существующей таблицы. Если перед именем таблицы указано `ONLY`, затрагивается только заданная таблица. Без `ONLY` затрагивается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно добавить необязательное указание `*`, чтобы явно обозначить, что должны затрагиваться и все дочерние таблицы.

Дополнительно может быть указан список столбцов. За подробностями обратитесь к CREATE PUBLICATION. Заметьте, что оформить одну подписку на несколько публикаций, в которых одна таблица опубликована с разными списками столбцов, нельзя. Подробнее проблемы, возможные при изменении списка столбцов, описаны в Внимание: объединение списков столбцов из нескольких публикаций.

Если указано необязательное предложение `WHERE`, строки, для которых *выражение* выдаёт false или null, не будут публиковаться. Заметьте, что выражение должно заключаться в круглые скобки. Вычисляется *выражение* от имени роли, которая применяется для соединения репликации.

имя_схемы

Имя существующей схемы.

`SET ( параметр_публикации [= значение] [, ... ] )`

Это предложение изменяет параметры публикации, изначально установленные командой CREATE PUBLICATION. За дополнительными сведениями обратитесь к её описанию.

новый_владелец

Имя пользователя, назначаемого новым владельцем публикации.

новое_имя

Новое имя публикации.

Примеры

Изменение публикации, чтобы публиковались только удаления и изменения:

```
ALTER PUBLICATION noinsert SET (publish = 'update, delete');
```

Добавление таблиц в публикацию:

```
ALTER PUBLICATION mypublication ADD TABLE users (user_id, firstname), departments;
```

Изменение набора столбцов, публикуемых для таблицы:

```
ALTER PUBLICATION mypublication SET TABLE users (user_id, firstname, lastname), TABLE
 departments;
```

Добавление схем `marketing` и `sales` в публикацию `sales_publication`:

```
ALTER PUBLICATION sales_publication ADD TABLES IN SCHEMA marketing, sales;
```

Добавление таблиц `users`, `departments` и схемы `production` в публикацию `production_publication`:

```
ALTER PUBLICATION production_publication ADD TABLE users, departments, TABLES IN SCHEMA
 production;
```

Совместимость

`ALTER PUBLICATION` является расширением Postgres Pro.

См. также

CREATE PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

ALTER ROLE

ALTER ROLE — изменить роль в базе данных

Синтаксис

```
ALTER ROLE указание_роли [ WITH ] параметр [ ... ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль' | PASSWORD NULL
    | VALID UNTIL 'дата_время'
    | PROFILE имя_профиля
    | ACCOUNT UNLOCK | ACCOUNT LOCK

ALTER ROLE имя RENAME TO новое_имя

ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_бд ] SET параметр_конфигурации
 { TO | = } { значение | DEFAULT }
ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_бд ] SET параметр_конфигурации
 FROM CURRENT
ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_бд ] RESET параметр_конфигурации
ALTER ROLE { указание_роли | ALL } [ IN DATABASE имя_бд ] RESET ALL

Здесь указание_роли:

    имя_роли
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Описание

`ALTER ROLE` изменяет атрибуты роли Postgres Pro.

Первая форма команды в этой справке может изменить многие атрибуты роли, которые можно указать в `CREATE ROLE`. (Покрываются все возможные атрибуты, отсутствуют только возможности добавления/удаления членов роли; для этого нужно использовать `GRANT` и `REVOKE`.) Атрибуты, не упомянутые в команде, сохраняют свои предыдущие значения. Суперпользователи базы данных могут изменить любые параметры любой роли за исключением параметра `SUPERUSER` для *начального суперпользователя*. Роли с правом `CREATEROLE`, не являющиеся суперпользователями, могут также менять многие параметры ролей, для которых имеют право `ADMIN OPTION`, но только не ролей суперпользователей и репликации. Роли, не являющиеся суперпользователями, не могут менять параметр `SUPERUSER`, а могут менять `CREATEDB`, `REPLICATION` и `BYPASSRLS`, но только если они имеют соответствующие свойства. Обычные пользователи (роли) могут менять только свой пароль.

Вторая форма меняет имя роли. Суперпользователи базы данных могут переименовать любую роль, а пользователи с правом `CREATEROLE` могут переименовывать роли (не суперпользователей), для которых у них есть право `ADMIN OPTION`. Также нельзя переименовать роль текущего пользователя в активном сеансе. (Если вам нужно сделать это, подключитесь другим пользователем.) Так как в паролях с `MD5`-шифрованием имя роли используется в качестве криптосоли, при переименовании роли её пароль очищается, если он был зашифрован `MD5`.

Оставшиеся формы меняют значение по умолчанию конфигурационной переменной, которое будет распространяться на сеансы роли во всех базах данных, либо, если добавлено предложение `IN DATABASE`, только на сеансы роли в заданной базе. Если вместо имени роли указано `ALL`, это значение переменной распространяется на все роли. Использование `ALL` с `IN DATABASE` по сути равносильно использованию команды `ALTER DATABASE ... SET ...`.

Когда эта роль впоследствии установит новое подключение, указанное значение станет значением по умолчанию в сеансе, переопределяя значение, заданное в `postgresql.conf` или полученное из командной строки `postgres`. Это происходит только в момент входа; при выполнении `SET ROLE` или `SET SESSION AUTHORIZATION` новые значения не применятся. Набор параметров для всех баз данных переопределяется параметрами уровня БД, установленными для роли. Параметры для конкретной базы данных или конкретной роли переопределяют параметры для всех ролей.

Суперпользователи могут менять значения переменных по умолчанию для любых ролей, а пользователи с правом `CREATEROLE` могут менять их только для ролей (не суперпользователей), для которых у них есть право `ADMIN OPTION`. Обычные пользователи могут определять переменные только для себя. Некоторые переменные конфигурации нельзя задать таким способом, а некоторые может настроить только суперпользователь. Параметры всех ролей во всех базах данных могут настраивать только суперпользователи.

Параметры

`имя`

Имя роли, атрибуты которой изменяются.

`CURRENT_ROLE`
`CURRENT_USER`

Выбирает для изменения текущего пользователя, а не явно задаваемую роль.

`SESSION_USER`

Выбирает для изменения текущего пользователя сеанса, а не явно задаваемую роль.

`SUPERUSER`
`NOSUPERUSER`
`CREATEDB`
`NOCREATEDB`
`CREATEROLE`
`NOCREATEROLE`
`INHERIT`
`NOINHERIT`
`LOGIN`
`NOLOGIN`
`REPLICATION`
`NOREPLICATION`
`BYPASSRLS`
`NOBYPASSRLS`
`CONNECTION LIMIT` *`предел_подключений`*
[`ENCRYPTED`] `PASSWORD` '*`пароль`*'
`PASSWORD NULL`
`VALID UNTIL` '*`дата_время`*'
`PROFILE` *`имя_профиля`*

Эти предложения меняют атрибуты, изначально установленные командой `CREATE ROLE`. За дополнительными сведениями обратитесь к странице справки `CREATE ROLE`.

`ACCOUNT UNLOCK`

Разблокировать роль. Роль может заблокироваться при превышении допустимого числа ошибок входа (за подробностями обратитесь к описанию CREATE PROFILE).

`ACCOUNT LOCK`

Заблокировать роль явным образом.

`новое_имя`

Новое имя роли.

`имя_бд`

Имя базы данных, в которой устанавливается конфигурационная переменная.

`параметр_конфигурации`
`значение`

Указанный параметр конфигурации принимает заданное значение по умолчанию в сеансах роли. Если *`значение`* задано как `DEFAULT` или, что то же самое, применяется операция `RESET`, переопределение этого параметра для роли удаляется и роль будет получать в новых сеансах системное значение параметра. Для очистки значений всех параметров, связанных с ролью, применяется `RESET ALL`. `SET FROM CURRENT` сохраняет текущее значение параметра в активном сеансе в качестве значения для данной роли. Если указано `IN DATABASE`, параметр конфигурации настраивается или удаляется только для данной роли и указанной базы данных.

Определения переменных для роли применяются только в начале сеанса; команды `SET ROLE` и `SET SESSION AUTHORIZATION` эти определения не обрабатывают.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

Примечания

Для добавления новых ролей используйте команду `CREATE ROLE`, а для удаления роли — `DROP ROLE`.

`ALTER ROLE` не может управлять членством роли, для этого применяется `GRANT` и `REVOKE`.

Указывая в этой команде незашифрованный пароль, следует проявлять осторожность. Пароль будет передаваться на сервер открытым текстом и может также записаться в историю команд клиента или в протокол работы сервера. В psql есть команда `\password`, с помощью которой можно сменить пароль роли, не рискуя рассекретить пароль.

Также возможно связать сеансовые значения по умолчанию с определённой базой данных, а не с ролью (см. ALTER DATABASE). В случае конфликта параметры для базы данных и роли переопределяют параметры только для роли, которые, в свою очередь, переопределяют параметры для базы данных.

Примеры

Изменение пароля роли:

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

Удаление пароля роли:

```
ALTER ROLE davide WITH PASSWORD NULL;
```

Изменение срока действия пароля (в частности, определяется, что пароль должен перестать действовать в полдень 4 мая 2015 г. в часовом поясе UTC+1):

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Установка бесконечного срока действия пароля:

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

Наделение роли правами на управление другими ролями и создание новых баз данных:

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Определение нестандартного значения параметра maintenance_work_mem для роли:

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

Определение нестандартного значения параметра client_min_messages для роли и заданной базы:

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

Совместимость

Оператор `ALTER ROLE` является расширением Postgres Pro.

См. также

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET, CREATE PROFILE

ALTER ROUTINE

ALTER ROUTINE — изменить определение подпрограммы

Синтаксис

```
ALTER ROUTINE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    действие [ ... ] [ RESTRICT ]
ALTER ROUTINE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    RENAME TO новое_имя
ALTER ROUTINE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER ROUTINE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    SET SCHEMA новая_схема
ALTER ROUTINE имя [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION имя_расширения

Где действие может быть следующим:

    IMMUTABLE | STABLE | VOLATILE
    [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST стоимость_выполнения
    ROWS строк_в_результате
    SET параметр_конфигурации { TO | = } { значение | DEFAULT }
    SET параметр_конфигурации FROM CURRENT
    RESET параметр_конфигурации
    RESET ALL
```

Описание

`ALTER ROUTINE` изменяет определение подпрограммы, то есть агрегата, функции или процедуры. Описание параметров, дополнительные примеры и подробности представлены в описаниях ALTER AGGREGATE, ALTER FUNCTION и ALTER PROCEDURE.

Примеры

Переименование подпрограммы `foo` для типа `integer` в `foobar`:

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

Эта команда будет работать независимо от того, является ли `foo` процедурой, агрегатной или обычной функцией.

Совместимость

Этот оператор частично совместим с оператором `ALTER ROUTINE` в стандарте SQL. За подробностями обратитесь к описаниям ALTER FUNCTION и ALTER PROCEDURE. Возможность сослаться по имени подпрограммы на агрегатную функцию является расширением Postgres Pro.

См. также

ALTER AGGREGATE, ALTER FUNCTION, ALTER PROCEDURE, DROP ROUTINE

Заметьте, что команды `CREATE ROUTINE` нет.

ALTER RULE

ALTER RULE — изменить определение правила

Синтаксис

```
ALTER RULE имя ON имя_таблицы RENAME TO новое_имя
```

Описание

`ALTER RULE` изменяет свойства существующего правила. В настоящее время эта команда может только изменить имя правила.

Использовать `ALTER RULE` может только владелец таблицы (или представления), к которой применяется это правило.

Параметры

`имя`

Имя существующего правила, подлежащего изменению.

`имя_таблицы`

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

`новое_имя`

Новое имя правила.

Примеры

Переименование существующего правила:

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

Совместимость

Оператор `ALTER RULE` является языковым расширением Postgres Pro, как и вся система перезаписи запросов.

См. также

CREATE RULE, DROP RULE

ALTER SCHEMA

ALTER SCHEMA — изменить определение схемы

Синтаксис

```
ALTER SCHEMA имя RENAME TO новое_имя
ALTER SCHEMA имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
```

Описание

`ALTER SCHEMA` изменяет определение схемы.

Выполнить `ALTER SCHEMA` может только владелец соответствующей схемы. Чтобы переименовать схему, необходимо также иметь право `CREATE` в базе данных схемы. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в базе данных. (Суперпользователи наделяются этими правами автоматически.)

Параметры

`имя`

Имя существующей схемы.

`новое_имя`

Новое имя схемы. Новое имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных схем.

`новый_владелец`

Новый владелец схемы.

Совместимость

Оператор `ALTER SCHEMA` отсутствует в стандарте SQL.

См. также

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

ALTER SEQUENCE — изменить определение генератора последовательности

Синтаксис

```
ALTER SEQUENCE [ IF EXISTS ] имя
    [ AS тип_данных ]
    [ INCREMENT [ BY ] шаг ]
    [ MINVALUE мин_значение | NO MINVALUE ] [ MAXVALUE макс_значение | NO MAXVALUE ]
    [ START [ WITH ] начало ]
    [ RESTART [ [ WITH ] перезапуск ] ]
    [ CACHE кеш ] [ [ NO ] CYCLE ]
    [ OWNED BY { имя_таблицы.имя_столбца | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] имя SET { LOGGED | UNLOGGED }
ALTER SEQUENCE [ IF EXISTS ] имя OWNER TO { новый_владелец | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] имя RENAME TO новое_имя
ALTER SEQUENCE [ IF EXISTS ] имя SET SCHEMA новая_схема
```

Описание

`ALTER SEQUENCE` меняет параметры существующего генератора последовательности. Параметры, не определяемые явно в команде `ALTER SEQUENCE`, сохраняют свои предыдущие значения.

Выполнить `ALTER SEQUENCE` может только владелец соответствующей последовательности. Чтобы сменить схему последовательности, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме последовательности. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать последовательность. Однако суперпользователь может сменить владельца последовательности в любом случае.)

Параметры

`имя`

Имя (возможно, дополненное схемой) последовательности, подлежащей изменению.

`IF EXISTS`

Не считать ошибкой, если последовательность не существует. В этом случае будет выдано замечание.

`тип_данных`

Необязательное предложение `AS` *`тип_данных`* меняет тип данных для последовательности. Допустимые типы: `smallint`, `integer` и `bigint`.

При изменении типа данных автоматически меняются минимальное и максимальное значения последовательности, в том и только в том случае, если это были минимальные и максимальные значения старого типа данных (другими словами, если последовательность была создана со свойствами `NO MINVALUE` или `NO MAXVALUE`, неявно или явно). В противном случае минимальные и максимальные значения сохраняются, если только в этой же команде не указаны новые значения. Если минимальное/максимальное значение не умещается в новом типе данных, выдаётся ошибка.

`шаг`

Предложение `INCREMENT BY` *`шаг`* является необязательным. При положительном значении шага генерируется возрастающая последовательность, при отрицательном — убывающая; если шаг не указан, сохраняется предыдущее значение.

`мин_значение`
`NO MINVALUE`

Необязательное предложение `MINVALUE мин_значение` определяет минимальное значение, которое будет генерировать данная последовательность. Если указано `NO MINVALUE`, для возрастающей последовательности этим значением будет 1, а для убывающей — минимальное число для её типа данных. В отсутствие этих указаний будет сохранено текущее минимальное значение.

`макс_значение`
`NO MAXVALUE`

Необязательное предложение `MAXVALUE макс_значение` определяет максимальное значение, которое будет генерировать данная последовательность. Если указано `NO MAXVALUE`, для возрастающей последовательности этим значением будет максимальное число для её типа данных, а для убывающей — -1. В отсутствие этих указаний будет сохранено текущее максимальное значение.

`начало`

Необязательное предложение `START WITH начало` меняет записанное начальное значение последовательности. При этом *текущее* значение последовательности не меняется, а только устанавливается значение, которое будет применено будущими командами `ALTER SEQUENCE RESTART`.

`перезапуск`

Необязательное предложение `RESTART [ WITH перезапуск ]` меняет текущее значение последовательности. Оно подобно вызову функции `setval` с параметром `is_called` = `false`: указанное значение перезапуска будет возвращено при *следующем* вызове функции `nextval`. Отсутствие в `RESTART` значения `перезапуск` равносильно передаче стартового значения, записанного командой `CREATE SEQUENCE` или последнего установленного командой `ALTER SEQUENCE START WITH`.

В отличие от вызова `setval`, операция `RESTART` с последовательностью является транзакционной и не даёт параллельным транзакциям получать числа из той же последовательности. Если это поведение не устраивает, следует воспользоваться функцией `setval`.

`кеш`

Предложение `CACHE кеш` разрешает предварительно выделять и сохранять в памяти числа последовательности для ускорения доступа к ним. Минимальное значение равно 1 (т. е. за один раз генерируется только одно значение, кеширования нет). Если это предложение отсутствует, сохраняется старое значение размера кеша.

`CYCLE`

Необязательное ключевое слово `CYCLE` позволяет зациклить последовательность при достижении `макс_значения` или `мин_значения` для возрастающей и убывающей последовательности, соответственно. Когда этот предел достигается, следующим числом этих последовательностей будет соответственно `мин_значение` или `макс_значение`.

`NO CYCLE`

Если добавляется необязательное указание `NO CYCLE`, при каждом вызове `nextval` после достижения предельного значения будет возникать ошибка. Если же указания `CYCLE` и `NO CYCLE` отсутствуют, сохраняется предыдущее поведение зацикливания.

`SET { LOGGED | UNLOGGED }`

Эта форма меняет характеристику журналирования последовательности, делает последовательность журналируемой/нежурналируемой соответственно (см. CREATE SEQUENCE). К временной последовательности она неприменима.

`OWNED BY` *`имя_таблицы.имя_столбца`*
`OWNED BY NONE`

Указание `OWNED BY` связывает последовательность с определённым столбцом таблицы, с тем чтобы при удалении этого столбца (или всей таблицы) автоматически удалилась и последовательность. Это указание заменяет любую ранее установленную связь данной последовательности. Целевая таблица должна иметь того же владельца и находиться в той же схеме, что и последовательность. Указание `OWNED BY NONE` убирает все существующие связи, обозначая последовательность «независимой».

`новый_владелец`

Имя пользователя, назначаемого новым владельцем последовательности.

`новое_имя`

Новое имя последовательности.

`новая_схема`

Новая схема последовательности.

Примечания

`ALTER SEQUENCE` не оказывает немедленного влияния на результаты `nextval` в серверных процессах, кроме текущего, которые могли предварительно сгенерировать (кешировать) значения последовательности. Эти процессы заметят изменения только после того, как будут израсходованы все кешированные значения. Текущий серверный процесс реагирует на изменения сразу.

`ALTER SEQUENCE` не влияет на значение `currval` последовательности. (В PostgreSQL до версии 8.3 это могло происходить.)

`ALTER SEQUENCE` блокирует параллельные вызовы `nextval`, `currval`, `lastval` и `setval`.

По историческим причинам `ALTER TABLE` тоже может работать с последовательностями, но все разновидности `ALTER TABLE`, допустимые для управления последовательностями, равнозначны вышеперечисленным формам.

Примеры

Перезапуск последовательности `serial` с числа 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Совместимость

Оператор `ALTER SEQUENCE` соответствует стандарту SQL, за исключением предложений `AS`, `START WITH`, `OWNED BY`, `OWNER TO`, `RENAME TO` и `SET SCHEMA`, являющихся расширениями Postgres Pro.

См. также

CREATE SEQUENCE, DROP SEQUENCE

ALTER SERVER

ALTER SERVER — изменить определение стороннего сервера

Синтаксис

```
ALTER SERVER имя [ VERSION 'новая_версия' ]
    [ OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] ) ]
ALTER SERVER имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER SERVER имя RENAME TO новое_имя
```

Описание

`ALTER SERVER` изменяет определение стороннего сервера. Первая форма меняет строку версии сервера или общие параметры сервера (требуется минимум одно предложение). Вторая форма меняет владельца сервера.

Изменить свойства сервера может только его владелец. Чтобы изменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца и, кроме того, иметь право `USAGE` для обёртки сторонних данных сервера. (Суперпользователи удовлетворяют всем этим условиям автоматически.)

Параметры

имя

Имя существующего сервера.

новая_версия

Новая версия сервера.

`OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )`

Эти формы изменяют параметры сервера. Указания `ADD`, `SET` и `DROP` определяют выполняемое действие (добавление, установка и удаление, соответственно). Если действие не задано явно, подразумевается `ADD`. Имена параметров должны быть уникальными, они вместе со значениями также проверяются библиотекой обёртки сторонних данных.

новый_владелец

Имя пользователя, назначаемого новым владельцем стороннего сервера.

новое_имя

Новое имя стороннего сервера.

Примеры

Изменение свойств сервера `foo`, добавление параметров подключения:

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

Изменение свойств сервера `foo`: смена версии, изменение параметра `host`:

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

Совместимость

`ALTER SERVER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Формы `OWNER TO` и `RENAME` являются расширениями Postgres Pro.

См. также

CREATE SERVER, DROP SERVER

ALTER STATISTICS

ALTER STATISTICS — изменить определение объекта расширенной статистики

Синтаксис

```
ALTER STATISTICS имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER STATISTICS имя RENAME TO новое_имя
ALTER STATISTICS имя SET SCHEMA новая_схема
ALTER STATISTICS имя SET STATISTICS { новый_ориентир | DEFAULT }
```

Описание

`ALTER STATISTICS` меняет параметры существующего объекта расширенной статистики. Параметры, не определённые явно в команде `ALTER STATISTICS`, сохраняют свои предыдущие значения.

Выполнить `ALTER STATISTICS` может только владелец объекта статистики. Чтобы сменить схему объекта статистики, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, и эта роль должна иметь право `CREATE` в схеме объекта статистики. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать объект статистики. Однако суперпользователь может сменить владельца объекта статистики в любом случае.)

Параметры

имя

Имя (возможно, дополненное схемой) объекта статистики, подлежащего изменению.

новый_владелец

Имя пользователя, назначаемого новым владельцем объекта статистики.

новое_имя

Новое имя объекта статистики.

новая_схема

Новая схема объекта статистики.

новый_ориентир

Ориентир сбора статистики для данного объекта, который будет учитываться при последующих операциях `ANALYZE`. Значение ориентира может лежать в диапазоне от 0 до 10000. Чтобы использовался системный ориентир статистики по умолчанию, необходимо задать значение `DEFAULT` (default_statistics_target). Для этого также можно задать значение -1, но теперь этот способ считается устаревшим. За дополнительными сведениями об использовании статистики планировщиком запросов Postgres Pro обратитесь к Разделу 14.2.

Совместимость

Оператор `ALTER STATISTICS` отсутствует в стандарте SQL.

См. также

CREATE STATISTICS, DROP STATISTICS

ALTER SUBSCRIPTION

ALTER SUBSCRIPTION — изменить определение подписки

Синтаксис

```
ALTER SUBSCRIPTION имя CONNECTION 'строка_подключения'
ALTER SUBSCRIPTION имя SET PUBLICATION имя_публикации [, ...] [ WITH
 ( параметр_публикации [= значение] [, ... ] ) ]
ALTER SUBSCRIPTION имя ADD PUBLICATION имя_публикации [, ...] [ WITH
 ( параметр_публикации [= значение] [, ... ] ) ]
ALTER SUBSCRIPTION имя DROP PUBLICATION имя_публикации [, ...] [ WITH
 ( параметр_публикации [= значение] [, ... ] ) ]
ALTER SUBSCRIPTION имя REFRESH PUBLICATION [ WITH ( параметр_обновления [= значение]
 [, ... ] ) ]
ALTER SUBSCRIPTION имя ENABLE
ALTER SUBSCRIPTION имя DISABLE
ALTER SUBSCRIPTION имя SET ( параметр_подписки [= значение] [, ... ] )
ALTER SUBSCRIPTION имя SKIP ( параметр_пропуска = значение )
ALTER SUBSCRIPTION имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER SUBSCRIPTION имя RENAME TO новое_имя
```

Описание

`ALTER SUBSCRIPTION` может менять многие свойства подписки, которые могут задаваться в CREATE SUBSCRIPTION.

Чтобы выполнить `ALTER SUBSCRIPTION` для подписки, нужно быть её владельцем. Чтобы переименовать подписку или сменить владельца, необходимо иметь право `CREATE` в базе данных. Кроме того, чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца. Если для подписки задано `password_required=false`, изменять её могут только суперпользователи.

При обновлении подписки производится удаление отношений, которые были из неё исключены, а также удаление слотов синхронизации этих таблиц, если таковые имеются. Эти слоты необходимо удалять, чтобы были освобождены ресурсы, выделенные для данной подписки на удалённом сервере. Если по причине сбоя в сети или какой-то другой ошибки Postgres Pro не сможет удалить эти слоты, выполнение команды будет прервано ошибкой. Для разрешения такой ситуации можно либо попробовать повторить эту операцию, либо разорвать связь слота с подпиской и удалить подписку, выполнив DROP SUBSCRIPTION.

Команды `ALTER SUBSCRIPTION ... REFRESH PUBLICATION`, `ALTER SUBSCRIPTION ... {SET|ADD|DROP} PUBLICATION ...` с параметром `refresh`, равным `true`, и `ALTER SUBSCRIPTION ... SET (failover = true|false)` не могут выполняться в блоке транзакции.

Команды `ALTER SUBSCRIPTION ... REFRESH PUBLICATION` и `ALTER SUBSCRIPTION ... {SET|ADD|DROP} PUBLICATION ...` с параметром `refresh`, равным `true`, также не могут выполняться, если для подписки включён режим `two_phase`, за исключением случая, когда `copy_data` равен `false`. Состояние режима двухфазной фиксации показывается в столбце `subtwophasestate` каталога `pg_subscription`.

Параметры

имя

Имя подписки, свойства которой изменяются.

`CONNECTION '`*`строка_подключения`*`'`

Это предложение заменяет строку соединения, изначально установленную командой CREATE SUBSCRIPTION. За дополнительными сведениями обратитесь к описанию этой команды.

`SET PUBLICATION` *`имя_публикации`*
`ADD PUBLICATION` *`имя_публикации`*
`DROP PUBLICATION` *`имя_публикации`*

Изменяет список публикаций, на которые оформлена подписка. Предложение `SET` полностью заменяет один список публикаций другим, `ADD` добавляет публикации к списку публикаций, а `DROP` удаляет публикации из этого списка. В формах `ADD` и `SET` допускается указание несуществующих публикаций в расчёте на то, что они будут добавлены позже. За подробностями обратитесь к описанию CREATE SUBSCRIPTION. По умолчанию эта команда также выполняет действие `REFRESH PUBLICATION`.

В указании *`параметр_публикации`* задаются дополнительные свойства операции. Поддерживаются следующие параметры:

`refresh` (`boolean`)

Со значением `false` данная команда не будет обновлять информацию о таблицах. В этом случае следует выполнить `REFRESH PUBLICATION` отдельно. Значение по умолчанию — `true`.

Кроме того, здесь могут задаваться параметры, относящиеся к `REFRESH PUBLICATION` и управляющие неявной операцией обновления.

`REFRESH PUBLICATION`

Считывает недостающую информацию о таблицах с публикующего сервера. В результате производится репликация таблиц, добавленных в публикации, на которые оформлена подписка, после `CREATE SUBSCRIPTION` или последнего вызова `REFRESH PUBLICATION`.

В указании *`параметр_обновления`* задаются дополнительные свойства операции обновления. Поддерживаются следующие параметры:

`copy_data` (`boolean`)

Определяет, должны ли копироваться уже существующие данные в публикациях, на которые оформляется подписка, когда начинается репликация. Значение по умолчанию — `true`.

Таблицы, входившие в подписку ранее, не копируются, даже если предложение `WHERE` фильтра строк таблицы изменилось с момента подписки.

Подробнее взаимодействие `copy_data = true` с параметром `origin` описано в Замечания.

Чтобы получить дополнительные сведения о копировании уже существующих данных в двоичном формате, обратитесь к описанию параметра `binary` команды `CREATE SUBSCRIPTION`.

`ENABLE`

Включает ранее отключённую подписку, запуская процесс логической репликации в конце транзакции.

`DISABLE`

Отключает активную подписку, останавливая процесс логической репликации в конце транзакции.

`SET (` *`параметр_подписки`* `[=` *`значение`*`] [, ... ] )`

Это предложение изменяет параметры, изначально установленные командой CREATE SUBSCRIPTION. За подробностями обратитесь к её описанию. Данное предложение позволяет изменить параметры `slot_name`, `synchronous_commit`, `binary`, `streaming`, `disable_on_error`, `password_required`, `run_as_owner`, `origin` и `failover`. Указать `password_required = false` может только суперпользователь.

При изменении `slot_name` параметры `failover` и `two_phase` указанного слота могут отличаться от параметров `failover` и `two_phase`, указанных в подписке. При создании слота убедитесь, что значения слота в параметрах `failover` и `two_phase` совпадают с указанными в подписке, иначе слот на публикующем сервере может вести себя по-другому: например, слот на публикующем сервере может синхронизироваться с резервными серверами даже при выключенном в подписке параметре `failover` и не синхронизироваться даже при включённом в подписке параметре `failover`.

`SKIP (` *`параметр_пропуска`* `=` *`значение`* `)`

Пропускает применение всех изменений удалённой транзакции. Если входящие данные нарушают какие-либо ограничения, логическая репликация будет остановлена, пока конфликты не будут разрешены. Когда выполняется команда `ALTER SUBSCRIPTION ... SKIP`, рабочий процесс логической репликации пропускает все изменения данных в транзакции. Этот параметр не влияет на транзакции, которые уже подготовлены на стороне подписчика (в результате действия режима `two_phase`). После того, как рабочий процесс логической репликации успешно пропускает транзакцию или транзакция завершается, номер LSN (хранящийся в `pg_subscription.subskiplsn`) очищается. Подробнее конфликты логической репликации описаны в Разделе 28.6.

`Параметр_пропуска` определяет свойства данной операции. Поддерживается следующий параметр:

`lsn` (`pg_lsn`)

Указывает LSN завершения удалённой транзакции, изменения которой должны быть пропущены процессом логической репликации. LSN завершения — это LSN, с которым транзакция зафиксирована либо подготовлена. Пропуск отдельных подтранзакций не поддерживается. Значение `NONE` сбрасывает ранее установленный LSN.

`новый_владелец`

Имя пользователя, назначаемого новым владельцем подписки.

`новое_имя`

Новое имя подписки.

Если для параметра типа `boolean` опустить `=` *`значение`*, это равнозначно указанию значения `TRUE`.

Примеры

Изменение подписки, заключающееся в подписывании на публикацию `insert_only`:

```
ALTER SUBSCRIPTION mysub SET PUBLICATION insert_only;
```

Отключение (остановка) подписки:

```
ALTER SUBSCRIPTION mysub DISABLE;
```

Совместимость

`ALTER SUBSCRIPTION` является расширением Postgres Pro.

См. также

CREATE SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

ALTER SYSTEM

ALTER SYSTEM — изменить параметр конфигурации сервера

Синтаксис

```
ALTER SYSTEM SET параметр_конфигурации { TO | = } { значение [, ...] | DEFAULT }

ALTER SYSTEM RESET параметр_конфигурации
ALTER SYSTEM RESET ALL
```

Описание

Оператор `ALTER SYSTEM` применяется для изменения параметров конфигурации сервера, распространяющихся на весь кластер баз данных. Пользоваться им может быть удобнее, чем вручную редактировать файл `postgresql.conf`. `ALTER SYSTEM` записывает заданное значение параметра в файл `postgresql.auto.conf`, который считывается сервером в дополнение к `postgresql.conf`. При указании в качестве значения параметра `DEFAULT` или применении формы `RESET` соответствующий элемент конфигурации удаляется из `postgresql.auto.conf`. Удалить все настроенные таким способом параметры позволяет предложение `RESET ALL`.

Значения, установленные командой `ALTER SYSTEM`, вступают в силу после следующей перезагрузки конфигурации сервера либо после перезапуска сервера (если это параметры, воспринимаемые только при запуске). Перезагрузить конфигурацию сервера можно, вызвав SQL-функцию `pg_reload_conf()`, выполнив `pg_ctl reload` или отправив сигнал SIGHUP главному серверному процессу.

Изменить какой-либо параметр, используя `ALTER SYSTEM`, могут только суперпользователи и пользователи, которым дано право `ALTER SYSTEM` для этого параметра. А так как эта команда работает непосредственно с файловой системой и не может быть отменена, её нельзя поместить в блок транзакции или функцию.

Параметры

параметр_конфигурации

Имя устанавливаемого параметра конфигурации. Список доступных параметров приведён в Главе 18.

значение

Новое значение параметра. Значениями могут быть строковые константы, идентификаторы, числа или списки таких элементов через запятую, в зависимости от конкретного параметра. Значения, не являющиеся числами или действительными идентификаторами, должны заключаться в кавычки. Если в качестве значения указать `DEFAULT`, параметр и его значение удаляются из `postgresql.auto.conf`.

Для некоторых параметров, принимающих в качестве значения список, значения в кавычках будут записываться в файл в двойных кавычках для сохранения пробелов и запятых; в других случаях для достижения такого эффекта необходимо использовать двойные кавычки внутри строк, заключённых в одинарные кавычки.

Примечания

С помощью этой команды нельзя задать data_directory, allow_alter_system, равно как и другие параметры, недопустимые в `postgresql.conf` (например, предустановленные параметры).

Другие способы настройки параметров описаны в Разделе 18.1.

`ALTER SYSTEM` можно отключить, задав параметру allow_alter_system значение `off`, однако это не является мерой по обеспечению безопасности, о чём говорится в описании данного параметра.

Примеры

Установка уровня ведения журнала транзакций (`wal_level`):

```
ALTER SYSTEM SET wal_level = replica;
```

Отмена изменения, восстановление значения, заданного в `postgresql.conf`:

```
ALTER SYSTEM RESET wal_level;
```

Совместимость

Оператор `ALTER SYSTEM` является расширением Postgres Pro.

См. также

SET, SHOW

ALTER TABLE

ALTER TABLE — изменить определение таблицы

Синтаксис

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    действие [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
ALTER TABLE [ IF EXISTS ] [ ONLY ] имя [ * ]
    RENAME CONSTRAINT имя_ограничения TO имя_нового_ограничения
ALTER TABLE [ IF EXISTS ] имя
    RENAME TO новое_имя
ALTER TABLE [ IF EXISTS ] имя
    SET SCHEMA новая_схема
ALTER TABLE ALL IN TABLESPACE имя [ OWNED BY имя_роли [, ... ] ]
    SET TABLESPACE новое_табл_пространство [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] имя
    ATTACH PARTITION имя_секции { FOR VALUES указание_границ_секции | DEFAULT }
ALTER TABLE [ IF EXISTS ] имя
    DETACH PARTITION имя_секции [ CONCURRENTLY | FINALIZE ]
ALTER TABLE [ IF EXISTS ] имя
    SPLIT PARTITION имя_секции INTO
        (PARTITION имя_секции1 { FOR VALUES указание_границ_секции | DEFAULT },
         PARTITION имя_секции2 { FOR VALUES указание_границ_секции | DEFAULT } [, ...])
ALTER TABLE [ IF EXISTS ] имя
    MERGE PARTITIONS (имя_секции1, имя_секции2 [, ...])
        INTO имя_секции

Где действие может быть следующим:

    ADD [ COLUMN ] [ IF NOT EXISTS ] имя_столбца тип_данных
 [ COLLATE правило_сортировки ] [ ограничение_столбца [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] имя_столбца [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] имя_столбца [ SET DATA ] TYPE тип_данных
 [ COLLATE правило_сортировки ] [ USING выражение ]
    ALTER [ COLUMN ] имя_столбца SET DEFAULT выражение
    ALTER [ COLUMN ] имя_столбца DROP DEFAULT
    ALTER [ COLUMN ] имя_столбца { SET | DROP } NOT NULL
    ALTER [ COLUMN ] имя_столбца SET EXPRESSION AS ( выражение )
    ALTER [ COLUMN ] имя_столбца DROP EXPRESSION [ IF EXISTS ]
    ALTER [ COLUMN ] имя_столбца ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
 [ ( параметры_последовательности ) ]
    ALTER [ COLUMN ] имя_столбца { SET GENERATED { ALWAYS | BY DEFAULT } |
 SET параметр_последовательности | RESTART [ [ WITH ] перезапуск ] } [...]
    ALTER [ COLUMN ] имя_столбца DROP IDENTITY [ IF EXISTS ]
    ALTER [ COLUMN ] имя_столбца SET STATISTICS { integer | DEFAULT }
    ALTER [ COLUMN ] имя_столбца SET ( атрибут = значение [, ... ] )
    ALTER [ COLUMN ] имя_столбца RESET ( атрибут [, ... ] )
    ALTER [ COLUMN ] имя_столбца SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN |
 DEFAULT }
    ALTER [ COLUMN ] имя_столбца SET COMPRESSION метод_сжатия
    ADD ограничение_таблицы [ NOT VALID ]
    ADD ограничение_таблицы_по_индексу
    ALTER CONSTRAINT имя_ограничения [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY
 DEFERRED | INITIALLY IMMEDIATE ]
```

```
    VALIDATE CONSTRAINT имя_ограничения
    DROP CONSTRAINT [ IF EXISTS ]  имя_ограничения [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ имя_триггера | ALL | USER ]
    ENABLE TRIGGER [ имя_триггера | ALL | USER ]
    ENABLE REPLICA TRIGGER имя_триггера
    ENABLE ALWAYS TRIGGER имя_триггера
    DISABLE RULE имя_правила_перезаписи
    ENABLE RULE имя_правила_перезаписи
    ENABLE REPLICA RULE имя_правила_перезаписи
    ENABLE ALWAYS RULE имя_правила_перезаписи
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON имя_индекса
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    SET ACCESS METHOD { новый_метод_доступа | DEFAULT }
    SET TABLESPACE новое_табл_пространство
    SET { LOGGED | UNLOGGED }
    SET ( параметр_хранения [= значение] [, ... ] )
    RESET ( параметр_хранения [, ... ] )
    INHERIT таблица_родитель
    NO INHERIT таблица_родитель
    OF имя_типа
    NOT OF
    OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX имя_индекса | FULL | NOTHING }

и указание_границ_секции:

IN ( выражение_границ_секции [, ...] ) |
FROM ( { выражение_границ_секции | MINVALUE | MAXVALUE } [, ...] )
  TO ( { выражение_границ_секции | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS числовая_константа, REMAINDER числовая_константа )

и ограничение_столбца:

[ CONSTRAINT имя_ограничения ]
{ NOT NULL |
  NULL |
  CHECK ( выражение ) [ NO INHERIT ] |
  DEFAULT выражение_по_умолчанию |
  GENERATED ALWAYS AS ( генерирующее_выражение ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( параметры_последовательности ) ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] параметры_индекса |
  PRIMARY KEY параметры_индекса |
  REFERENCES целевая_таблица [ ( целевой_столбец ) ] [ MATCH FULL | MATCH PARTIAL |
 MATCH SIMPLE ]
    [ ON DELETE ссылочное_действие ] [ ON UPDATE ссылочное_действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

и ограничение_таблицы:

[ CONSTRAINT имя_ограничения ]
{ CHECK ( выражение ) [ NO INHERIT ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] ( имя_столбца [, ... ] ) параметры_индекса |
  PRIMARY KEY ( имя_столбца [, ... ] ) параметры_индекса |
```

```
  EXCLUDE [ USING индексный_метод ] ( элемент_исключения WITH оператор
 [, ... ] ) параметры_индекса [ WHERE ( предикат ) ] |
  FOREIGN KEY ( имя_столбца [, ... ] ) REFERENCES целевая_таблица [ ( целевой_столбец
 [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE ссылочное_действие ] [ ON
 UPDATE ссылочное_действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

и ограничение_таблицы_по_индексу:

    [ CONSTRAINT имя_ограничения ]
    { UNIQUE | PRIMARY KEY } USING INDEX имя_индекса
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

параметры_индекса в ограничениях UNIQUE, PRIMARY KEY и EXCLUDE:

[ INCLUDE ( имя_столбца [, ... ] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) ]
[ USING INDEX TABLESPACE табл_пространство ]

элемент_исключения в ограничении EXCLUDE:

{ имя_столбца | ( выражение ) } [ COLLATE правило_сортировки ] [ класс_операторов
 [ ( параметр_класса_оп = значение [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST |
 LAST } ]

ссылочное_действие в ограничении FOREIGN KEY/REFERENCES:

{ NO ACTION | RESTRICT | CASCADE | SET NULL [ ( имя_столбца [, ... ] ) ] | SET DEFAULT
 [ ( имя_столбца [, ... ] ) ] }
```

Описание

`ALTER TABLE` меняет определение существующей таблицы. Несколько её разновидностей описаны ниже. Заметьте, что для разных разновидностей могут требоваться разные уровни блокировок. Если явно не отмечено другое, запрашивается блокировка `ACCESS EXCLUSIVE`. При указании нескольких подкоманд будет запрашиваться самая сильная блокировка из требуемых ими.

`ADD COLUMN [ IF NOT EXISTS ]`

Эта форма добавляет в таблицу новый столбец, с тем же синтаксисом, что и `CREATE TABLE`. Если указано `IF NOT EXISTS` и столбец с таким именем уже существует, это не будет ошибкой.

`DROP COLUMN [ IF EXISTS ]`

Эта форма удаляет столбец из таблицы. При этом автоматически будут удалены индексы и ограничения таблицы, связанные с этим столбцом. Также будет удалена многовариантная статистика, охватывающая удаляемый столбец, если после его удаления в статистике останутся данные только одного столбца. Если от этого столбца зависят какие либо объекты вне этой таблицы, например, внешние ключи или представления, чтобы удалить их, необходимо добавить указание `CASCADE`. Если в команде указано `IF EXISTS` и этот столбец не существует, это не считается ошибкой, вместо этого просто выдаётся замечание.

`SET DATA TYPE`

Эта форма меняет тип столбца таблицы. Индексы и простые табличные ограничения, включающие этот столбец, будут автоматически преобразованы для использования нового типа столбца, для чего будет заново разобрано определяющее их выражение. Необязательное предложение `COLLATE` задаёт правило сортировки для нового столбца; если оно опущено, выбирается правило сортировки по умолчанию для нового типа. Необязательное предложение `USING` определяет, как новое значение столбца будет получено из старого; если оно отсутствует, выполняется приведение типа по умолчанию, как обычное присваивание значения старого типа новому. Предложение `USING` становится обязательным, если неявное приведение или присваивание с приведением старого типа к новому не определено.

При изменении типа столбца статистика столбца удаляется, поэтому после этой операции для таблицы рекомендуется выполнить `ANALYZE`.

`SET/DROP DEFAULT`

Эти формы задают или удаляют значение по умолчанию для столбца (удаление равносильно указанию NULL в качестве значения по умолчанию). Новые значения по умолчанию применяются только при последующих командах `INSERT` или `UPDATE`; их изменения не отражаются в строках, уже существующих в таблице.

`SET/DROP NOT NULL`

Эти формы определяют, будет ли столбец принимать значения NULL или нет.

Указание `SET NOT NULL` можно применить к столбцу, только если ни одна из записей таблицы не содержит в этом столбце значение `NULL`. Обычно эта проверка производится в момент выполнения `ALTER TABLE` и требует сканирования всей таблицы; однако в случае наличия ограничений `CHECK` (и если они не удаляются в той же команде), гарантирующих отсутствие `NULL`, сканирование таблицы пропускается.

Если данная таблица является секцией, операцию `DROP NOT NULL` нельзя выполнить для столбца, если он определён с характеристикой `NOT NULL` в родительской таблице. Чтобы удалить ограничение `NOT NULL` из всех секций, выполните `DROP NOT NULL` для родительской таблицы. Даже если ограничение `NOT NULL` в родительской таблице отсутствует, при желании такое ограничение может быть добавлено в отдельные секции. Другими словами, потомки могут запрещать значения NULL, даже если родитель их допускает, но не наоборот.

`SET EXPRESSION AS`

Эта форма заменяет выражение генерируемого столбца. Существующие данные в столбцах обновляются, для всех будущих изменений будет использоваться новое генерирующее выражение.

При изменении типа столбца статистика столбца удаляется, поэтому после этой операции для таблицы рекомендуется выполнить `ANALYZE`.

`DROP EXPRESSION [ IF EXISTS ]`

Эта форма преобразует хранимый генерируемый столбец в обычный. Существующие данные в столбцах сохраняются, но будущие изменения будут вноситься не генерирующим выражением.

Если указано `DROP EXPRESSION IF EXISTS` и заданный столбец не является хранимым генерируемым столбцом, это не считается ошибкой. В этом случае выдаётся только замечание.

`ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY`
`SET GENERATED { ALWAYS | BY DEFAULT }`
`DROP IDENTITY [ IF EXISTS ]`

Это формы меняют для столбца свойство идентификации или меняют характеристику генерирования для существующего столбца идентификации. За подробностями обратитесь к описанию `CREATE TABLE`. Как и `SET DEFAULT`, эти формы влияют на поведение только последующих команд `INSERT` и `UPDATE`; их действие не отражаются в строках, уже существующих в таблице.

Если указано `DROP IDENTITY IF EXISTS` и заданный столбец не является столбцом идентификации, это не считается ошибкой. В этом случае выдаётся только замечание.

`SET` *`параметр_последовательности`*
`RESTART`

Эти формы меняют нижележащую последовательность ранее созданного столбца идентификации. Здесь *`параметр_последовательности`* — это параметр, поддерживаемый командой `ALTER SEQUENCE`, например `INCREMENT BY`.

`SET STATISTICS`

Эта форма задаёт ориентир сбора статистики по столбцу для последующих операций `ANALYZE`. Значение ориентира может лежать в диапазоне от 0 до 10000. Чтобы использовался системный ориентир статистики по умолчанию, необходимо задать значение `DEFAULT` (default_statistics_target). Для этого также можно задать значение -1, но теперь этот способ считается устаревшим. За дополнительными сведениями об использовании статистики планировщиком запросов Postgres Pro обратитесь к Разделу 14.2.

`SET STATISTICS` запрашивает блокировку `SHARE UPDATE EXCLUSIVE`.

`SET (` *`атрибут`* `=` *`значение`* `[, ... ] )`
`RESET (` *`атрибут`* `[, ... ] )`

Эта форма устанавливает или сбрасывает параметры атрибутов. В настоящее время единственными параметрами атрибутов являются `n_distinct` и `n_distinct_inherited`, которые переопределяют оценку кол-ва_различных_значений, производимую последующими операциями `ANALYZE`. Атрибут `n_distinct` влияет на расчёт статистики по самой таблице, а `n_distinct_inherited` — на статистику по таблице и её потомкам. Если заданное значение положительно, `ANALYZE` будет считать, что столбец содержит именно это количество различных значений не NULL. Если заданное значение отрицательно (оно должно быть больше или равно -1), `ANALYZE` будет считать, что количество различных значений не NULL в столбце линейно зависит от размера таблицы; точное число будет получено умножением примерного размера таблицы на абсолютное значение параметра. Например, при -1 будет предполагаться, что различны все значения в столбце, а при -0,5 — что в среднем каждое значение повторяется дважды. Это может быть полезно, когда размер таблицы меняется со временем, так как умножение на число строк в таблице производится только во время планирования запроса. С 0 количество различных значений оценивается как обычно. За дополнительными сведениями об использовании статистики планировщиком запросов Postgres Pro обратитесь к Разделу 14.2.

Для изменения параметров атрибутов запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

`SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }`

Эта форма устанавливает режим хранения столбца. Она определяет, хранятся ли данные внутри таблицы или в отдельной таблице TOAST, а также, сжимаются ли они. Режим `PLAIN` должен применяться для значений фиксированной длины, таких как `integer`; это вариант хранения внутри, без сжатия. Режим `MAIN` применяется для хранения внутри, но сжатых данных, `EXTERNAL` — для внешнего хранения несжатых данных, а `EXTENDED` — для внешнего хранения сжатых данных. `EXTENDED` используется по умолчанию для большинства типов данных, поддерживающих хранилище не `PLAIN`. Применение `EXTERNAL` позволяет ускорить операции с подстроками на очень больших значениях `text` и `bytea`, за счёт проигрыша в объёме хранилища. Заметьте, что предложение `SET STORAGE` само по себе не меняет ничего в таблице, оно только задаёт стратегию, которая будет реализована при будущих изменениях в таблице. За дополнительными сведениями обратитесь к Разделу 63.2.

`SET COMPRESSION` *`метод_сжатия`*

Эта форма задаёт метод сжатия для столбца, определяющий, как будут сжиматься значения, помещаемые в этот столбец в будущем (если режим хранения вообще допускает сжатие). При изменении этого метода перезапись таблицы не производится, поэтому уже существующие в таблице данные могут оставаться сжатыми другими методами. Если таблица восстанавливается средством pg_restore, все значения перезаписываются с применением заданного метода сжатия. Если же данные вставляются из другого отношения (например, командой `INSERT ...`

`SELECT`), значения из исходной таблицы не обязательно будут распаковываться, так что ранее сжатые данные могут сохраниться с прежним методом сжатия, они не будут пережиматься методом, установленным для целевого столбца. Поддерживаемые методы сжатия: `pglz` и `lz4`. (`lz4` поддерживается, только если Postgres Pro был собран с ключом `--with-lz4`.) Кроме того, *метод_сжатия* может принимать значение `default`, с которым применяемый метод сжатия определяется текущим значением параметра default_toast_compression во время добавления данных.

`ADD` *`ограничение_таблицы`* `[ NOT VALID ]`

Эта форма добавляет в таблицу новое ограничение, принимая тот же синтаксис описания ограничения, что и `CREATE TABLE`, а также дополнительное указание `NOT VALID`, которое в настоящее время поддерживается только для ограничений внешнего ключа и ограничений-проверок.

Обычно эта форма влечёт сканирование всей таблицы для проверки, что все существующие строки в таблице удовлетворяют новому ограничению. Но если используется указание `NOT VALID`, эта потенциально длительная проверка пропускается. Тем не менее это ограничение будет действовать при последующих добавлениях или изменениях данных (то есть эти операции не будут выполнены, если новая строка нарушит условие ограничения-проверки, либо при наличии внешнего ключа в главной таблице не найдётся соответствующая строка). Но база данных не будет считать, что ограничение выполняется для всех строк таблицы, пока оно не будет проверено с применением указания `VALIDATE CONSTRAINT`. Дополнительную информацию об использовании указания `NOT VALID` вы найдете ниже, в разделе Замечания.

Хотя почти все разновидности `ADD` *`ограничение_таблицы`* требуют блокировку `ACCESS EXCLUSIVE`, для указания `ADD FOREIGN KEY` требуется только блокировка `SHARE ROW EXCLUSIVE`. Заметьте, что `ADD FOREIGN KEY` запрашивает такую блокировку как в таблице, в которой добавляется это ограничение, так и в таблице, на которую это ограничение ссылается.

С ограничениями уникальности и первичного ключа, добавляемыми в секционированные таблицы, связаны дополнительные требования; см. описание `CREATE TABLE`. Кроме того, в настоящее время ограничения внешнего ключа для секционированных таблиц не могут объявляться как непроверенные (`NOT VALID`).

`ADD` *`ограничение_таблицы_по_индексу`*

Эта форма добавляет в таблицу новое ограничение `PRIMARY KEY` или `UNIQUE` на базе существующего уникального индекса. В это ограничение будут включены все столбцы данного индекса.

Индекс не может быть частичным и включать столбцы-выражения. Кроме того, это должен быть индекс-B-дерево с порядком сортировки по умолчанию. С такими ограничениями добавляемые индексы не будут ничем отличаться от индексов, создаваемых обычными командами `ADD PRIMARY KEY` и `ADD UNIQUE`.

В случае с указанием `PRIMARY KEY`, если столбцы индекса ещё не помечены `NOT NULL`, данная команда попытается выполнить `ALTER COLUMN SET NOT NULL` для каждого столбца. При этом потребуется произвести полное сканирование таблицы, чтобы убедиться, что столбец(ы) не содержит NULL. Во всех остальных случаях это быстрая операция.

Если задано имя ограничения, индекс будет переименован и получит заданное имя. В противном случае именем ограничения станет имя индекса.

После выполнения этой команды индекс становится «принадлежащим» ограничению, так же, как если бы он был создан обычной командой `ADD PRIMARY KEY` или `ADD UNIQUE`. Это значит, в частности, что при удалении ограничения индекс будет удалён вместе с ним.

Эта форма с секционированными таблицами в настоящее время не поддерживается.

Примечание

Добавление ограничения на базе существующего индекса полезно в ситуациях, когда новое ограничение требуется добавить, не блокируя на долгое время внесение изменений в

> таблицу. Для этого можно создать индекс командой `CREATE UNIQUE INDEX CONCURRENTLY`, а затем преобразовать его в ограничение, используя эту запись. См. следующий пример.

`ALTER CONSTRAINT`

Эта форма меняет атрибуты созданного ранее ограничения. В настоящее время изменять можно только ограничения внешнего ключа.

`VALIDATE CONSTRAINT`

Эта форма проверяет ограничение внешнего ключа или ограничение-проверку, созданное ранее с указанием `NOT VALID`, сканируя всю таблицу с целью убедиться, что ограничению удовлетворяют все строки. Если ограничение уже помечено как проверенное, ничего не происходит. (В чём польза этой команды, вы можете узнать в разделе Замечания.)

Эта команда запрашивает блокировку `SHARE UPDATE EXCLUSIVE`.

`DROP CONSTRAINT [ IF EXISTS ]`

Эта форма удаляет указанное ограничение таблицы, вместе с нижележащим индексом, если таковой имеется. Если указано `IF EXISTS` и заданное ограничение не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

`DISABLE`/`ENABLE [ REPLICA | ALWAYS ] TRIGGER`

Эти формы настраивают срабатывание триггера(ов), принадлежащего таблице. Отключённый триггер сохраняется в системе, но не выполняется, когда происходит вызывающее его событие. (Для отложенных триггеров состояние включения проверяется при возникновении события, а не когда фактически вызывается функция триггера.) Эта команда может отключить или включить один триггер по имени, либо все триггеры таблицы, либо только пользовательские триггеры (кроме сгенерированных внутрисистемных триггеров ограничений, например, триггеров, реализующих ограничения внешнего ключа или отложенные ограничения уникальности и ограничения-исключения). Для отключения или включения сгенерированных внутрисистемных триггеров ограничений требуются права суперпользователя; отключать их следует с осторожностью, так как очевидно, что невозможно гарантировать целостность ограничений, если триггеры не работают.

На механизм срабатывания триггеров также влияет конфигурационная переменная session_replication_role. Включённые без дополнительных указаний (по умолчанию) триггеры будут срабатывать, когда роль репликации — «origin» (по умолчанию) или «local». Триггеры, включённые указанием `ENABLE REPLICA`, будут срабатывать, только если текущий режим сеанса — «replica», а триггеры, включённые указанием `ENABLE ALWAYS`, будут срабатывать независимо от текущего режима репликации.

Эффект этого механизма состоит в том, что в конфигурации по умолчанию триггеры не срабатывают в репликах. Это полезно, потому что если триггер используется в исходной базе для распределения данных по таблицам, система репликации будет реплицировать и распределённые таким образом данные, поэтому триггер не должен срабатывать в реплике второй раз, так как иначе будет иметь место дублирование. Однако если триггер используется для других целей, например, выдаёт уведомления, может иметь смысл установить для него свойство `ENABLE ALWAYS`, чтобы он также срабатывал в репликах.

Когда эта команда применяется к секционированной таблице, состояния соответствующих «клонированных» триггеров в секциях также изменяются, если не указано `ONLY`.

Эта команда запрашивает блокировку `SHARE ROW EXCLUSIVE`.

`DISABLE`/`ENABLE [ REPLICA | ALWAYS ] RULE`

Эти формы настраивают срабатывание правил перезаписи, относящихся к таблице. Отключённое правило сохраняется в системе, но не применяется во время переписывания запроса. По сути эти операции подобны операциям включения/отключения триггеров. Однако это не распространяется на правила `ON SELECT` — они применяются всегда, чтобы представления продолжали работать, даже в сеансах, исполняющих не основную роль репликации.

На механизм срабатывания правил также оказывает влияние конфигурационная переменная session_replication_role, подобное тому, что описано выше применительно к триггерам.

`DISABLE/ENABLE ROW LEVEL SECURITY`

Эти формы управляют применением относящихся к таблице политик защиты строк. Если защита включается, но политики для таблицы не определены, применяется политика запрета доступа по умолчанию. Заметьте, что политики могут быть определены для таблицы, даже если защита на уровне строк отключена. В этом случае политики *не* применяются и их ограничения игнорируются. См. также описание `CREATE POLICY`.

`NO FORCE/FORCE ROW LEVEL SECURITY`

Эти формы управляют применением относящихся к таблице политик защиты строк, когда пользователь является её владельцем. Если это поведение включается, политики защиты на уровне строк будут действовать и на владельца таблицы. Если оно отключено (по умолчанию), защита на уровне строк не будет действовать на пользователя, являющегося владельцем таблицы. См. также описание `CREATE POLICY`.

`CLUSTER ON`

Эта форма выбирает индекс по умолчанию для последующих операций `CLUSTER`. Собственно кластеризация таблицы при этом не выполняется.

Для изменения параметров кластеризации запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

`SET WITHOUT CLUSTER`

Эта форма удаляет последнее заданное указание индекса для `CLUSTER`. Её действие отразится на будущих операциях кластеризации, для которых не будет задан индекс.

Для изменения параметров кластеризации запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

`SET WITHOUT OIDS`

Обеспечивающий обратную совместимость синтаксис удаления системного столбца `oid`. Так как добавить системные столбцы `oid` теперь невозможно, это указание фактически не действует.

`SET ACCESS METHOD`

Эта форма изменяет метод доступа к таблице, переписывая её с использованием указанного метода доступа. Если указать `DEFAULT`, будет выбран метод, заданный в конфигурационном параметре default_table_access_method. За дополнительными сведениями обратитесь к Главе 59.

При этом в секционированной таблице данных для перезаписи нет, однако созданные позже секции будут иметь значение метода доступа по умолчанию, пока их не перезапишут с использованием предложения `USING`. Если указать `DEFAULT`, ранее установленное значение удалится, и в результате новые секции будут использовать метод по умолчанию, заданный в `default_table_access_method`.

`SET TABLESPACE`

Эта форма меняет табличное пространство таблицы на заданное и перемещает файлы данных, связанные с таблицей, в новое пространство. Индексы таблицы, если они имеются, не перемещаются; однако их можно переместить отдельно дополнительными командами `SET TABLESPACE`. В секционированной таблице при этом ничего не перемещается, но секции, созданные впоследствии с указанием `CREATE TABLE PARTITION OF`, будут использовать это табличное пространство (если только оно не будет переопределено предложением `TABLESPACE`).

Форма `ALL IN TABLESPACE` позволяет перенести в другое табличное пространство все таблицы текущей базы данных в текущем пространстве, при этом она сначала блокирует все таблицы, а затем переносит каждую из них. Эта форма также поддерживает предложение `OWNED BY`, с которым перемещаются только таблицы указанных владельцев. С указанием `NOWAIT` команда завершается ошибкой, если не может получить все требуемые блокировки немедленно. Заметьте, что системные каталоги эта форма не перемещает; если требуется переместить их, следует использовать `ALTER DATABASE` или явные вызовы `ALTER TABLE`. Отношения `information_schema` не считаются частью системных каталогов и подлежат перемещению. См. также описание `CREATE TABLESPACE`.

`SET { LOGGED | UNLOGGED }`

Эта форма меняет характеристику журналирования таблицы, делает таблицу журналируемой/нежурналируемой, соответственно (см. `UNLOGGED`). К временной таблице она неприменима.

При этом также меняется режим журналирования всех последовательностей, связанных с таблицей (с её столбцами идентификаторов или столбцами serial). Однако режим журналирования таких последовательностей можно изменить и отдельно.

`SET (` *`параметр_хранения`* `[=` *`значение`*`] [, ... ] )`

Эта форма меняет один или несколько параметров хранения таблицы. Подробнее допустимые параметры рассмотрены в разделе Параметры хранения описания `CREATE TABLE`. Заметьте, что эта команда не меняет содержимое таблицы немедленно; в зависимости от параметра может потребоваться перезаписать таблицы, чтобы получить желаемый эффект. Это можно сделать с помощью команд `VACUUM FULL`, `CLUSTER` или одной из форм `ALTER TABLE`, принудительно перезаписывающих таблицу. Изменения параметров, связанных с планировщиком, вступают в силу при следующей блокировке таблицы, так что в текущих запросах они не проявляются.

Данная форма затребует блокировку `SHARE UPDATE EXCLUSIVE` для изменения параметров хранения, связанных с фактором заполнения, TOAST и автоочисткой, а также параметра планировщика `parallel_workers`.

`RESET (` *`параметр_хранения`* `[, ... ] )`

Эта форма сбрасывает один или несколько параметров хранения к значениям по умолчанию. Как и с `SET`, для полного обновления таблицы может потребоваться перезаписать таблицу.

`INHERIT` *`таблица_родитель`*

Эта форма назначает целевую таблицу потомком заданной родительской таблицы. Впоследствии запросы к родительской таблице будут включать записи и целевой таблицы. Чтобы таблица могла стать потомком, она должна содержать те же столбцы, что и родительская (хотя она может включать и дополнительные столбцы). Столбцы должны иметь одинаковые типы данных и, если в родительской таблице какие-то из них имеют ограничение `NOT NULL`, они должны иметь ограничение `NOT NULL` и в таблице-потомке.

Также в таблице-потомке должны присутствовать все ограничения `CHECK` родительской таблицы, за исключением ненаследуемых (то есть созданных командой `ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT`), которые игнорируются; при этом все соответствующие ограничения в таблице-потомке не должны быть ненаследуемыми. В настоящее время ограничения `UNIQUE`, `PRIMARY KEY` и `FOREIGN KEY` не учитываются, но в будущем это может измениться.

`NO INHERIT` *`таблица_родитель`*

Эта форма удаляет целевую таблицу из списка потомков указанной родительской таблицы. Результаты запросов к родительской таблице после этого не будут включать записи, взятые из целевой таблицы.

`OF` *`имя_типа`*

Эта форма связывает таблицу с составным типом, как если бы она была сформирована командой `CREATE TABLE OF`. При этом список имён и типов столбцов должен точно соответствовать тому, что образует составной тип. Кроме того, таблица не должна быть потомком какой-либо другой таблицы. Эти ограничения гарантируют, что команда `CREATE TABLE OF` позволит создать таблицу с таким же определением.

`NOT OF`

Эта форма разрывает связь типизированной таблицы с её типом.

`OWNER TO`

Эта форма меняет владельца таблицы, последовательности, представления, материализованного представления или сторонней таблицы на заданного пользователя.

`REPLICA IDENTITY`

Эта форма меняет информацию, записываемую в журнал предзаписи для идентификации изменяемых или удаляемых строк. В большинстве случаев старое значение в каждом столбце записывается, только если оно отличается от нового, но значение, хранящееся отдельно, записывается всегда, даже если оно не изменилось. Данный параметр действует только при использовании логической репликации.

`DEFAULT`

Записываются старые значения столбцов первичного ключа, если он есть. Это режим по умолчанию для несистемных таблиц.

`USING INDEX` *`имя_индекса`*

Записываются старые значения столбцов, составляющих заданный индекс, который должен быть уникальным, не частичным, не отложенным и включать только столбцы, помеченные `NOT NULL`. Если этот индекс удалён, поведение будет таким же, как в режиме `NOTHING`.

`FULL`

Записываются старые значения всех столбцов в строке.

`NOTHING`

Информация о старой строке не записывается. Это режим по умолчанию для системных таблиц.

`RENAME`

Формы `RENAME` меняют имя таблицы (или индекса, последовательности, представления, материализованного представления или сторонней таблицы), имя отдельного столбца таблицы или имя ограничения таблицы. При переименовании ограничения, у которого имеется нижележащий индекс, этот индекс также переименовывается. На хранимые данные это не влияет.

`SET SCHEMA`

Эта форма перемещает таблицу в другую схему. Вместе с таблицей перемещаются связанные с ней индексы и ограничения, а также последовательности, принадлежащие столбцам таблицы.

`ATTACH PARTITION` *`имя_секции`* `{ FOR VALUES` *`указание_границ_секции`* `| DEFAULT }`

Эта форма присоединяет существующую таблицу (которая тоже может быть секционированной) в качестве секции к целевой таблице. С указанием `FOR VALUES` таблица станет секцией для определённых значений, а с указанием `DEFAULT` — секцией по умолчанию. Для каждого индекса в целевой таблице будет создан соответствующий индекс в присоединяемой таблице; или, если равнозначный индекс уже существует, он будет присоединён к индексу целевой таблицы, как при выполнении команды `ALTER INDEX ATTACH PARTITION`. Заметьте, что если существующая таблица является сторонней, в настоящее время её нельзя присоединить в качестве секции к целевой таблице, в которой имеются уникальные индексы (`UNIQUE`). (См. также CREATE FOREIGN TABLE.) Также для каждого существующего в целевой таблице пользовательского триггера уровня строк будет создан такой же триггер в присоединяемой таблице.

Для секции, добавляемой с `FOR VALUES`, используется то же *`указание_границ_секции`*, что и в `CREATE TABLE`. Это указание должно соответствовать стратегии секционирования и ключу секционирования целевой таблицы. Присоединяемая таблица должна иметь те же столбцы, что и целевая, и никаких других; более того, должны совпадать и типы столбцов. Кроме того, в ней должны быть те же ограничения `NOT NULL` и `CHECK`, что и в целевой таблице, не помеченные как `NO INHERIT`. Ограничения `FOREIGN KEY` в настоящее время не учитываются. Ограничения `UNIQUE` и `PRIMARY KEY` из родительской таблицы будут созданы в секции, если они там ещё не существуют.

Если новая секция является обычной таблицей, чтобы убедиться, что ни одна строка в таблице не нарушает ограничение секции, производится полное сканирование таблицы. Такого сканирования можно избежать, добавив перед выполнением этой команды в таблицу действующее ограничение `CHECK`, допускающее только такие строки, которые удовлетворяют задаваемому ограничению секции. Наличие этого ограничения позволит определить, что таблицу не нужно сканировать для проверки ограничения секции. Однако это не будет работать, если какие-либо ключи секционирования являются выражениями и секция не принимает значения `NULL`. При присоединении секции по списку, не принимающей значения `NULL`, также добавьте ограничение `NOT NULL` в столбец ключа секционирования, если это не выражение.

Если новая секция является сторонней таблицей, никакая проверка, удовлетворяют ли все строки сторонней таблицы ограничению секции, не выполняется. (Обсуждение ограничений сторонней таблицы вы можете найти в CREATE FOREIGN TABLE.)

Когда у таблицы есть секция по умолчанию, у данной секции при добавлении новой меняется ограничение секции. Секция по умолчанию не может содержать строки, которые должны быть перенесены в новую секцию, поэтому она будет просканирована и проверена на предмет их отсутствия. Этого сканирования, как и сканирования новой секции, можно избежать, если определить подходящее ограничение `CHECK`. Кроме того, сканирование этой секции, как и новой, всегда пропускается, когда данная секция является сторонней таблицей.

Для присоединения секции затребуется блокировка `SHARE UPDATE EXCLUSIVE` в родительской таблице, помимо блокировок `ACCESS EXCLUSIVE` в присоединяемых таблицах и секции по умолчанию (при наличии).

Подобные блокировки также должны устанавливаться во всех вложенных секциях, если присоединяемая таблица либо секция по умолчанию в свою очередь является секционированной таблицей. Блокировки вложенных секций можно избежать, добавив ограничение `CHECK`, как это описано в Подразделе 5.12.2.2.

`DETACH PARTITION` *`имя_секции`* `[ CONCURRENTLY | FINALIZE ]`

Эта форма отсоединяет заданную секцию от целевой таблицы. Отсоединяемая секция продолжит существовать как отдельная таблица, но более не будет иметь никаких связей с таблицей, от которой была отсоединена. Все индексы, которые были присоединены к индексам целевой таблицы, отсоединяются, а триггеры, созданные как копии существующих в целевой таблице, удаляются. Таблицы, ссылающиеся на секционированную таблицу через внешние ключи, блокируются в режиме `SHARE`.

С указанием `CONCURRENTLY` эта операция выполняется с пониженным уровнем блокировок во избежание блокирования других сеансов, которые могут обращаться к данной секционированной таблице. В этом режиме отсоединение производится в двух транзакциях. В первой транзакции в родительской таблице и заданной секции устанавливается блокировка `SHARE UPDATE EXCLUSIVE`, а секция помечается как отсоединяемая, после чего транзакция фиксируется; затем должны завершиться все остальные транзакции, использующие секционированную таблицу. Когда все они завершаются, вторая транзакция получает блокировку `SHARE UPDATE EXCLUSIVE` для секционированной таблицы и `ACCESS EXCLUSIVE` для отсоединяемой секции, после чего процесс отсоединения заканчивается. В секцию добавляется ограничение `CHECK`, повторяющее ограничение секции. Указание `CONCURRENTLY` нельзя использовать в блоке транзакции, а также оно не допускается, если секционированная таблица содержит секцию по умолчанию.

Указание `FINALIZE` позволяет завершить операцию `DETACH CONCURRENTLY`, прерванную или отменённую ранее. В один момент времени в состоянии отсоединения от некоторой секционированной таблицы может находиться не более одной секции.

`SPLIT PARTITION` *`имя_секции`* `INTO (PARTITION` *`имя_секции1`* `{ FOR VALUES` *`указание_границ_секции`* `| DEFAULT }, PARTITION` *`имя_секции2`* `{ FOR VALUES` *`указание_границ_секции`* `| DEFAULT } [, ...])`

Эта форма разделяет секцию целевой таблицы. Секционирование по хешу не поддерживается. Границы новых секций не должны пересекаться с новыми и существующими секциями (за исключением *`имя_секции`*). Если разделённая секция была секцией по умолчанию (`DEFAULT`), то одна из новых секций должна также стать секцией по умолчанию. Если одна из новых или существующих секций является секцией по умолчанию (`DEFAULT`), то у новых секций *`имя_секции1`*, *`имя_секции2`* и т.д. могут быть пробелы между границами с другими секциями. Если в секционированной таблице нет секции по умолчанию (`DEFAULT`), таковой можно назначить одну из новых секций.

Если в секционированной таблице и среди новых секций нет секции по умолчанию (`DEFAULT`), должны соблюдаться следующие условия: сумма границ всех новых секций (*`имя_секции1`*, *`имя_секции2`* и т.д.) должна равняться границе разделяемой секции *`имя_секции`*. Одна из новых секций *`имя_секции1`*, *`имя_секции2`* и т.д может иметь одинаковое имя с разделяемой секцией *`имя_секции`* (это удобно, если разделяется секция по умолчанию: после разделения остаётся секция с таким же именем). Разделять можно только простые, ещё не разделённые секции.

Новые секции создаются так же, как таблицы при помощи SQL-команды `CREATE TABLE` *`имя_секцииN`* `(LIKE` *`имя`* `INCLUDING ALL EXCLUDING INDEXES EXCLUDING IDENTITY EXCLUDING STATISTICS)`. Индексы и идентификаторы создаются после переноса данных в новые секции. Расширенная статистика не копируется из родительской таблицы в целях согласованности с командой `CREATE TABLE PARTITION OF`. Новые секции наследуют метод доступа от родительских таблиц. Если родительская таблица постоянная, то и новые секции будут созданы как постоянные. Если родительская таблица временная, то и секции будут временные. Новые секции создаются в том же табличном пространстве, что и родительские.

Примечание

Данная команда получает блокировку `ACCESS EXCLUSIVE`. При большой нагрузке это серьёзно ограничивает использование данной команды для работы с большими секционированными таблицами.

`MERGE PARTITIONS (`*`имя_секции1`*`,` *`имя_секции2`* `[, ...]) INTO` *`имя_секции`*

Эта форма объединяет несколько секций в одну секцию целевой таблицы. Секционирование по хешу не поддерживается. Если секции по умолчанию `DEFAULT` нет в списке секций *`имя_секции1`*, *`имя_секции2`* и т. д.:

- Для таблиц, секционируемых по диапазонам, необходимо, чтобы диапазоны секций *`имя_секции1`*, *`имя_секции2`* и т. д. можно было объединить в один диапазон без пробелов и пересечений, иначе возникнет ошибка. Объединённый диапазон включает в себя диапазон секции *`имя_секции`*.
- Для таблиц, секционируемых по спискам, списки значений всех секций *`имя_секции1`*, *`имя_секции2`* и т.д. соединяются и формируют список значений секции *`имя_секции`*.

Если секция по умолчанию `DEFAULT` есть в списке секций *`имя_секции1`*, *`имя_секции2`* и т. д.:

- Секция *`имя_секции`* становится секцией по умолчанию `DEFAULT`.

- Для таблиц, секционируемых по диапазонам или спискам, диапазоны или списки значений объединённых секций могут быть любыми.

У новой секции *`имя_секции`* может быть такое же имя, как у одной из объединённых секций. Объединять можно только простые, ещё не разделённые секции.

Новая секция создаётся так же, как таблица при помощи SQL-команды `CREATE TABLE` *`имя секции`* `(LIKE` *`имя`* `INCLUDING ALL EXCLUDING INDEXES EXCLUDING IDENTITY EXCLUDING STATISTICS)`. Индексы и идентификаторы создаются после переноса данных в новую секцию. Расширенная статистика не копируется из родительской таблицы в целях согласованности с командой `CREATE TABLE PARTITION OF`. Новая секция наследует метод доступа от родительских таблиц. Если родительская таблица постоянная, то и новая секция будет создана как постоянная. Если родительская таблица временная, то и секция будет временной. Новая секция создаётся в том же табличном пространстве, что и родительская таблица.

> ### Примечание
>
> Данная команда получает блокировку `ACCESS EXCLUSIVE`. При большой нагрузке это серьёзно ограничивает использование данной команды для работы с большими секционированными таблицами.

Все формы ALTER TABLE, затрагивающие одну таблицу, кроме `RENAME`, `SET SCHEMA`, `ATTACH PARTITION`, `DETACH PARTITION`, `SPLIT PARTITION`, а также `MERGE PARTITIONS` можно объединить в набор множественных изменений и применить вместе. Например, можно добавить несколько столбцов и/или изменить тип столбцов в одной команде. Это особенно полезно для больших таблиц, так как вся таблица обрабатывается за один проход.

Выполнить `ALTER TABLE` может только владелец соответствующей таблицы. Чтобы сменить схему или табличное пространство таблицы, необходимо также иметь право `CREATE` в новой схеме или табличном пространстве. Чтобы сделать таблицу потомком другой таблицы, нужно быть владельцем и родительской таблицы. Также, чтобы подсоединить таблицу к другой в качестве секции, необходимо быть владельцем подсоединяемой таблицы. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме таблицы. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать таблицу. Однако суперпользователь может сменить владельца таблицы в любом случае.) Чтобы добавить столбец, сменить тип столбца или применить предложение `OF`, необходимо также иметь право `USAGE` для соответствующего типа данных.

Параметры

`IF EXISTS`

Не считать ошибкой, если таблица не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) существующей таблицы, подлежащей изменению. Если перед именем таблицы указано `ONLY`, изменяется только заданная таблица. Без `ONLY` изменяется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание `*`, чтобы явно обозначить, что изменению подлежат все дочерние таблицы.

`имя_столбца`

Имя нового или существующего столбца.

`новое_имя_столбца`

Новое имя существующего столбца.

новое_имя

Новое имя таблицы.

тип_данных

Тип данных нового столбца или новый тип данных существующего столбца.

ограничение_таблицы

Новое ограничение таблицы.

имя_ограничения

Имя нового или существующего ограничения.

`CASCADE`

Автоматически удалять объекты, зависящие от удаляемого столбца или ограничения (например, представления, содержащие этот столбец), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении столбца или ограничения, если существуют зависящие от них объекты. Это поведение по умолчанию.

имя_триггера

Имя включаемого или отключаемого триггера.

`ALL`

Отключить или включить все триггеры, принадлежащие таблице. (Для этого требуются права суперпользователя, если в числе этих триггеров оказываются сгенерированные внутрисистемные триггеры исключений, например те, что реализуют ограничения внешнего ключа или отложенные ограничения уникальности и исключений.)

`USER`

Отключить или включить все триггеры, принадлежащие таблице, за исключением сгенерированных внутрисистемных триггеров исключений, например, тех, что реализуют ограничения внешнего ключа или отложенные ограничения уникальности и исключений.

имя_индекса

Имя существующего индекса.

параметр_хранения

Имя параметра хранения таблицы

значение

Новое значение параметра хранения таблицы. Это может быть число или строка, в зависимости от параметра.

таблица_родитель

Родительская таблица, с которой будет установлена или разорвана связь данной таблицы.

новый_владелец

Имя пользователя, назначаемого новым владельцем таблицы.

новый_метод_доступа

Имя метода доступа, который будет использоваться для таблицы после её преобразования.

новое_табл_пространство

Имя табличного пространства, в которое будет перемещена таблица.

новая_схема

Имя схемы, в которую будет перемещена таблица.

имя_секции

Имя таблицы, присоединяемой в качестве новой секции, или наоборот, отсоединяемой от данной таблицы, или имя разделяемой секции, или имя новой объединяемой секции.

указание_границ_секции

Указание границ для новой секции. Подробнее синтаксис этого указания рассматривается в описании CREATE TABLE.

Замечания

Ключевое слово `COLUMN` не несёт смысловой нагрузки и может быть опущено.

Когда конструкция `ADD COLUMN` добавляет столбец и для него задано неизменчивое выражение `DEFAULT`, значение по умолчанию вычисляется во время выполнения оператора и сохраняется в метаданных таблицы. Это значение затем будет использовано в качестве содержимого столбца для всех существующих строк. Если указание `DEFAULT` отсутствует, столбец будет содержать NULL. В обоих случаях перезапись таблицы не требуется.

Добавление столбца с изменчивым выражением `DEFAULT` или изменение типа существующего столбца влечёт за собой перезапись всей таблицы и её индексов. Но возможно исключение при смене типа существующего столбца: если предложение `USING` не меняет содержимое столбца и старый тип двоично приводится к новому или является неограниченным доменом поверх нового типа, то перезапись таблицы не требуется. Тем не менее, индексы нужно перестраивать в любом случае, кроме случаев, когда система может определить, что новый индекс будет логически эквивалентен существующему. Например, когда меняется правило сортировки для столбца, индекс требуется перестроить всегда, так как новый порядок сортировки может быть другим. Однако если правило сортировки не меняется, тип столбца можно поменять с `text` на `varchar` (или наоборот) без перестроения индексов, так как эти типы данных сортируются одинаково. Перестроение больших таблиц и/или их индексов может быть весьма длительной процедурой, которая при этом временно требует вдвое больше места на диске.

Добавление ограничений `CHECK` или `NOT NULL` влечёт за собой необходимость просканировать таблицу, чтобы проверить, что все существующие строки удовлетворяют ограничению, но перезаписывать таблицу при этом не требуется.

Подобным образом, при присоединении новой секции может производиться её сканирование для проверки, соответствуют ли существующие строки ограничению секции.

Возможность объединения множества изменений в одну команду `ALTER TABLE` полезна в основном тем, что позволяет совместить сканирования и перезаписи таблицы, требуемые этим операциям, и выполнить их за один проход.

Сканирование большой таблицы для проверки нового внешнего ключа или ограничения-проверки может занять длительное время и будет препятствовать внесению других изменений до фиксирования команды `ALTER TABLE ADD CONSTRAINT`. Основное предназначение указания `NOT VALID` при добавлении ограничения состоит в уменьшении влияния этой операции на параллельные изменения данных. С указанием `NOT VALID` команда `ADD CONSTRAINT` не сканирует таблицу и может быть зафиксирована немедленно. После этого можно выполнить команду `VALIDATE CONSTRAINT`, которая проверит все существующие строки на соответствие ограничению. Эта команда не будет препятствовать параллельным изменениям, так как ей известно, что в других транзакциях для добавляемых или изменяемых строк ограничение уже будет действовать; проверить нужно только уже существующие строки. Таким образом, для этой проверки в таблице затребуется только блокировка `SHARE UPDATE EXCLUSIVE`. (Если ограничение является внешним ключом, то в целевой таблице этого ключа также затребуется блокировка `ROW SHARE`.) Помимо оптимизации параллельной работы, указание `NOT VALID` и предложение `VALIDATE CONSTRAINT` полезно в случаях, когда заведомо известно, что в таблице есть строки, нарушающие ограничения. После создания ограничения добавить новые недопустимые строки будет невозможно, а все существующие проблемы могут разрешаться в удобное время, пока `VALIDATE CONSTRAINT` не выполнится успешно.

Форма `DROP COLUMN` не удаляет столбец физически, а просто делает его невидимым для операций SQL. При последующих операциях добавления или изменения в этот столбец будет записываться значение NULL. Таким образом, удаление столбца выполняется быстро, но при этом размер таблицы на диске не уменьшается, так как пространство, занимаемое удалённым столбцом, не высвобождается. Это пространство будет освобождено со временем, по мере изменения существующих строк.

Чтобы принудительно высвободить пространство, занимаемое столбцом, который был удалён, можно выполнить одну из форм `ALTER TABLE`, производящих перезапись всей таблицы. В результате все строки будут воссозданы так, что в удалённом столбце будет содержаться NULL.

Перезаписывающие формы `ALTER TABLE` небезопасны с точки зрения MVCC. После перезаписи таблица будет выглядеть пустой для параллельных транзакций, если они работают со снимком, полученным до момента перезаписи. За подробностями обратитесь к Разделу 13.6.

В указании `USING` предложения `SET DATA TYPE` на самом деле можно записать выражение со старыми значениями строки; то есть, оно может ссылаться как на преобразуемые столбцы, так и на другие. Это позволяет записывать в `SET DATA TYPE` очень общие преобразования данных. Ввиду такой гибкости, выражение `USING` не применяется к значению по умолчанию данного столбца (если таковое есть); результат может быть не константным выражением, что требуется для значения по умолчанию. Это означает, что в случае отсутствия явного приведения или присваивания старого типа новому, `SET DATA TYPE` может не справиться с преобразованием значения по умолчанию, несмотря на то, что применяется предложение `USING`. В этих случаях нужно удалить значение по умолчанию с помощью `DROP DEFAULT`, выполнить `ALTER TYPE`, а затем с помощью `SET DEFAULT` задать новое подходящее значение по умолчанию. Подобные соображения применимы и в отношении индексов и ограничений с этим столбцом.

Если у таблицы имеются дочерние таблицы, то добавлять, переименовывать столбцы или менять их тип в родительской таблице, не повторяя ту же операцию в дочерних таблицах, нельзя. Это правило гарантирует, что столбцы в дочерних таблицах всегда соответствуют родительской. Подобным образом, нельзя переименовать ограничение `CHECK` в родительской таблице, не переименовывая его во всех дочерних таблицах, что тоже гарантирует соответствие всех ограничений `CHECK`. (Однако это не касается ограничений, построенных на индексах.) И так как выборка из родительской таблицы влечёт за собой выборку из всех потомков, ограничение родителя не может быть помечено как действующее, если оно также не является действующим в потомках. Во всех этих случаях команда `ALTER TABLE ONLY` не будет выполнена.

Рекурсивная операция `DROP COLUMN` удалит столбец из дочерней таблицы, только если этот столбец не наследуется от каких-то других родителей и никогда не был определён в дочерней таблице независимо. Нерекурсивная операция `DROP COLUMN` (т. е., `ALTER TABLE ONLY ... DROP COLUMN`) никогда не удаляет унаследованные столбцы; вместо этого она помечает их как независимо определённые, а не наследуемые. С секционированной таблицей нерекурсивная команда `DROP COLUMN` выдаст ошибку, так как все секции таблицы должны содержать те же столбцы, что и главная таблица.

Действия для столбцов идентификации (`ADD GENERATED`, `SET` и т. д., `DROP IDENTITY`), а также действия `CLUSTER`, `OWNER` и `TABLESPACE`, никогда не распространяются рекурсивно на дочерние таблицы; то есть они всегда выполняются так, как будто указано `ONLY`. Действия, влияющие на состояния триггеров, распространяются рекурсивно на секции секционированных таблиц (если не указано `ONLY`), но не на потомков в традиционной иерархии наследования. Операция добавления ограничения выполняется рекурсивно только для ограничений `CHECK`, не помеченных как `NO INHERIT`.

Какие-либо изменения таблиц системного каталога не допускаются.

За более подробным описанием допустимых параметров обратитесь к CREATE TABLE. Дополнительно о наследовании можно узнать в Главе 5.

Примеры

Добавление в таблицу столбца типа `varchar`:

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

При этом во всех существующих строках таблицы новый столбец получит значение null.

Добавление столбца со значением по умолчанию (отличным от NULL):

```
ALTER TABLE measurements
  ADD COLUMN mtime timestamp with time zone DEFAULT now();
```

В существующих строках новый столбец будет содержать текущее время, а в добавляемых впоследствии строках — время их добавления.

Добавление столбца и заполнение его значением, отличным от значения по умолчанию, которое будет использоваться в дальнейшем:

```
ALTER TABLE transactions
  ADD COLUMN status varchar(30) DEFAULT 'old',
  ALTER COLUMN status SET default 'current';
```

Существующие строки будут заполнены значением `old`, но для последующих команд значением по умолчанию будет `current`. Результат этих двух указаний в одной команде будет таким же, как и при выполнении их в отдельных командах `ALTER TABLE`.

Удаление столбца из таблицы:

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

Изменение типов двух существующих столбцов в одной операции:

```
ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

Смена типа целочисленного столбца, содержащего время в стиле Unix, на тип `timestamp with time zone` с применением предложения `USING`:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

То же самое, но в случае, когда у столбца есть значение по умолчанию, не приводимое автоматически к новому типу данных:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

Переименование существующего столбца:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

Переименование существующей таблицы:

```
ALTER TABLE distributors RENAME TO suppliers;
```

Переименование существующего ограничения:

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

Добавление в столбец ограничения NOT NULL:

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

Удаление ограничения NOT NULL из столбца:

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

Добавление ограничения-проверки в таблицу и все её потомки:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

Добавление ограничения-проверки только в таблицу, но не в её потомки:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO
 INHERIT;
```

(Данное ограничение-проверка не будет наследоваться и будущими потомками тоже.)

Удаление ограничения-проверки из таблицы и из всех её потомков:

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

Удаление ограничения-проверки только из самой таблицы:

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

(Ограничение-проверка остаётся во всех дочерних таблицах.)

Добавление в таблицу ограничения внешнего ключа:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES
 addresses (address);
```

Добавление в таблицу ограничения внешнего ключа с наименьшим влиянием на работу других:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES
 addresses (address) NOT VALID;
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

Добавление в таблицу ограничения уникальности (по нескольким столбцам):

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

Добавление в таблицу первичного ключа с автоматическим именем (учтите, что в таблице может быть только один первичный ключ):

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

Перемещение таблицы в другое табличное пространство:

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

Перемещение таблицы в другую схему:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

Пересоздание ограничения первичного ключа без блокировки изменений в процессе перестроения индекса:

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

Присоединение секции к таблице, секционируемой по диапазонам:

```
ALTER TABLE measurement
    ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO
 ('2016-08-01');
```

Присоединение секции к таблице, секционируемой по списку:

```
ALTER TABLE cities
    ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

Присоединение секции к таблице, секционируемой по хешу:

```
ALTER TABLE orders
    ATTACH PARTITION orders_p4 FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Присоединение секции по умолчанию к секционированной таблице:

```
ALTER TABLE cities
    ATTACH PARTITION cities_partdef DEFAULT;
```

Удаление секции из секционированной таблицы:

```
ALTER TABLE measurement
    DETACH PARTITION measurement_y2015m12;
```

Разделение секции таблицы, секционированной по диапазонам:

```
ALTER TABLE sales_range SPLIT PARTITION sales_feb_mar_apr2023 INTO
   (PARTITION sales_feb2023 FOR VALUES FROM ('2023-02-01') TO ('2023-03-01'),
    PARTITION sales_mar2023 FOR VALUES FROM ('2023-03-01') TO ('2023-04-01'),
    PARTITION sales_apr2023 FOR VALUES FROM ('2023-04-01') TO ('2023-05-01'));
```

Разделение секции таблицы, секционированной по списку:

```
ALTER TABLE sales_list SPLIT PARTITION sales_all INTO
   (PARTITION sales_west FOR VALUES IN ('Lisbon', 'New York', 'Madrid'),
    PARTITION sales_east FOR VALUES IN ('Bejing', 'Delhi', 'Vladivostok'),
    PARTITION sales_central FOR VALUES IN ('Warsaw', 'Berlin', 'Kyiv'));
```

Объединение нескольких секций в одну в целевой таблице:

```
ALTER TABLE sales_list MERGE PARTITIONS (sales_west, sales_east, sales_central)
    INTO sales_all;
```

Совместимость

Формы `ADD` (без `USING INDEX`), `DROP [COLUMN]`, `DROP IDENTITY`, `RESTART`, `SET DEFAULT`, `SET DATA TYPE` (без `USING`), `SET GENERATED` и `SET` *`параметр_последовательности`* соответствуют стандарту SQL. Другие формы являются расширениями стандарта SQL, реализованными в Postgres Pro. Кроме того, расширением является возможность указать в одной команде `ALTER TABLE` несколько операций изменения.

`ALTER TABLE DROP COLUMN` позволяет удалить единственный столбец таблицы и оставить таблицу без столбцов. Это является расширением стандарта SQL, который не допускает существование таблиц с нулём столбцов.

См. также

CREATE TABLE

ALTER TABLESPACE

ALTER TABLESPACE — изменить определение табличного пространства

Синтаксис

```
ALTER TABLESPACE имя RENAME TO новое_имя
ALTER TABLESPACE имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER TABLESPACE имя SET ( параметр_табличного_пространства = значение [, ... ] )
ALTER TABLESPACE имя RESET ( параметр_табличного_пространства [, ... ] )
```

Описание

`ALTER TABLESPACE` может применяться для изменения определения табличного пространства.

Чтобы изменить определение табличного пространства, нужно быть его владельцем. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца. (Заметьте, что суперпользователи наделяются этими правами автоматически.)

Параметры

`имя`

Имя существующего табличного пространства.

`новое_имя`

Новое имя табличного пространства. Новое имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных табличных пространств.

`новый_владелец`

Новый владелец табличного пространства.

`параметр_табличного_пространства`

Устанавливаемый или сбрасываемый параметр табличного пространства. В настоящее время поддерживаются только параметры `seq_page_cost`, `random_page_cost` и `effective_io_concurrency`. При установке этих значений для заданного табличного пространства будет переопределена обычная оценка стоимости чтения страниц из таблиц в этом пространстве, настраиваемая одноимённым параметром конфигурации (см. seq_page_cost, random_page_cost, effective_io_concurrency). Это может быть полезно, если одно из табличных пространств размещено на диске, который быстрее или медленнее остальной дисковой системы.

Примеры

Переименование табличного пространства `index_space` в `fast_raid`:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

Смена владельца табличного пространства `index_space`:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

Совместимость

Оператор `ALTER TABLESPACE` отсутствует в стандарте SQL.

См. также

CREATE TABLESPACE, DROP TABLESPACE

ALTER TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION — изменить определение конфигурации текстового поиска

Синтаксис

```
ALTER TEXT SEARCH CONFIGURATION имя
    ADD MAPPING FOR тип_фрагмента [, ... ] WITH имя_словаря [, ... ]
ALTER TEXT SEARCH CONFIGURATION имя
    ALTER MAPPING FOR тип_фрагмента [, ... ] WITH имя_словаря [, ... ]
ALTER TEXT SEARCH CONFIGURATION имя
    ALTER MAPPING REPLACE старый_словарь WITH новый_словарь
ALTER TEXT SEARCH CONFIGURATION имя
    ALTER MAPPING FOR тип_фрагмента [, ... ] REPLACE старый_словарь WITH новый_словарь
ALTER TEXT SEARCH CONFIGURATION имя
    DROP MAPPING [ IF EXISTS ] FOR тип_фрагмента [, ... ]
ALTER TEXT SEARCH CONFIGURATION имя RENAME TO новое_имя
ALTER TEXT SEARCH CONFIGURATION имя OWNER TO { новый_владелец | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION имя SET SCHEMA новая_схема
```

Описание

`ALTER TEXT SEARCH CONFIGURATION` изменяет определение конфигурации текстового поиска. Эта команда позволяет настроить сопоставления типов фрагментов со словарями или сменить владельца или имя конфигурации.

Выполнить `ALTER TEXT SEARCH CONFIGURATION` может только владелец соответствующей конфигурации.

Параметры

имя

Имя (возможно, дополненное схемой) существующей конфигурации текстового поиска.

тип_фрагмента

Имя типа фрагмента, выдаваемое при разборе конфигурации.

имя_словаря

Имя словаря текстового поиска, в котором будет искаться указанный тип фрагмента. Если указаны несколько словарей, они просматриваются в порядке перечисления.

старый_словарь

Имя словаря текстового поиска, которое будет заменено в сопоставлении.

новый_словарь

Имя словаря текстового поиска, которое будет подставлено там, где был *старый_словарь*.

новое_имя

Новое имя конфигурации текстового поиска.

новый_владелец

Новый владелец конфигурации текстового поиска.

новая_схема

Новая схема конфигурации текстового поиска.

Форма `ADD MAPPING FOR` настраивает список словарей, которые будут просматриваться в поиске указанных типов фрагментов; если сопоставление для каких-либо типов уже задано, возникнет ошибка. Форма `ALTER MAPPING FOR` делает то же самое, но она сначала удаляет существующее сопоставление для этих типов фрагментов. Формы `ALTER MAPPING REPLACE` подставляют *новый_словарь* вместо *старый_словарь* везде, где упоминается последний. Это выполняется только для указанных типов фрагментов, когда присутствует `FOR`, либо для всех сопоставлений в конфигурации в противном случае. Форма `DROP MAPPING` удаляет все словари для заданных типов фрагментов, в результате чего фрагменты этих типов будут игнорироваться конфигурацией. Если сопоставлений для заданных типов фрагментов нет, возникает ошибка, если только не добавлено указание `IF EXISTS`.

Примеры

В следующем примере словарь `english` заменяется на `swedish` везде, где использовался `english` в конфигурации `my_config`.

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

Совместимость

Оператор `ALTER TEXT SEARCH CONFIGURATION` отсутствует в стандарте SQL.

См. также

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY — изменить определение словаря текстового поиска

Синтаксис

```
ALTER TEXT SEARCH DICTIONARY имя (
    параметр [ = значение ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY имя RENAME TO новое_имя
ALTER TEXT SEARCH DICTIONARY имя OWNER TO { новый_владелец | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH DICTIONARY имя SET SCHEMA новая_схема
```

Описание

`ALTER TEXT SEARCH DICTIONARY` изменяет определение словаря текстового поиска. Эта команда позволяет изменить параметры словаря, связанные с шаблонами, или сменить владельца или имя словаря.

Выполнить `ALTER TEXT SEARCH DICTIONARY` может только владелец словаря.

Параметры

имя

Имя (возможно, дополненное схемой) существующего словаря текстового поиска.

параметр

Имя параметра шаблона, устанавливаемого для данного словаря.

значение

Новое значение для параметра настройки шаблонов. Если знак равно и значение опущено, предыдущее значение параметра удаляется из словаря, что позволяет вернутся к значению по умолчанию.

новое_имя

Новое имя словаря текстового поиска.

новый_владелец

Новый владелец словаря текстового поиска.

новая_схема

Новая схема словаря текстового поиска.

Параметры настройки шаблонов могут перечисляться в любом порядке.

Примеры

Команда в следующем примере меняет список стоп-слов словаря на базе Snowball. Другие параметры остаются неизменными.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

Команда в следующем примере меняет параметр, определяющий язык, на `dutch`, и удаляет параметр, задающий список стоп-слов.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

Следующая команда «изменяет» определение словаря, на самом деле не меняя ничего.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

(Это работает потому, что код удаления параметра не считает ошибкой отсутствие такого параметра.) Этот трюк может быть полезен при изменении файлов конфигурации словаря; `ALTER` принудит все существующие сеансы перечитать файлы конфигурации, что в противном случае они не сделают никогда, если прочитали конфигурацию ранее.

Совместимость

Оператор `ALTER TEXT SEARCH DICTIONARY` отсутствует в стандарте SQL.

См. также

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER — изменить определение анализатора текстового поиска

Синтаксис

```
ALTER TEXT SEARCH PARSER имя RENAME TO новое_имя
ALTER TEXT SEARCH PARSER имя SET SCHEMA новая_схема
```

Описание

`ALTER TEXT SEARCH PARSER` изменяет определение анализатора текстового поиска. В настоящее время эта команда позволяет только сменить его имя.

Выполнить `ALTER TEXT SEARCH PARSER` может только суперпользователь.

Параметры

имя

Имя (возможно, дополненное схемой) существующего анализатора текстового поиска.

новое_имя

Новое имя анализатора текстового поиска.

новая_схема

Новая схема анализатора текстового поиска.

Совместимость

Оператор `ALTER TEXT SEARCH PARSER` отсутствует в стандарте SQL.

См. также

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

ALTER TEXT SEARCH TEMPLATE

ALTER TEXT SEARCH TEMPLATE — изменить определение шаблона текстового поиска

Синтаксис

```
ALTER TEXT SEARCH TEMPLATE имя RENAME TO новое_имя
ALTER TEXT SEARCH TEMPLATE имя SET SCHEMA новая_схема
```

Описание

`ALTER TEXT SEARCH TEMPLATE` изменяет определение шаблона текстового поиска. В настоящее время эта команда позволяет только сменить его имя.

Выполнить команду `ALTER TEXT SEARCH TEMPLATE` может только суперпользователь.

Параметры

`имя`

Имя (возможно, дополненное схемой) существующего шаблона текстового поиска.

`новое_имя`

Новое имя шаблона текстового поиска.

`новая_схема`

Новая схема шаблона текстового поиска.

Совместимость

Оператор `ALTER TEXT SEARCH TEMPLATE` отсутствует в стандарте SQL.

См. также

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

ALTER TRIGGER

ALTER TRIGGER — изменить определение триггера

Синтаксис

```
ALTER TRIGGER имя ON имя_таблицы RENAME TO новое_имя
ALTER TRIGGER имя ON имя_таблицы [ NO ] DEPENDS ON EXTENSION имя_расширения
```

Описание

`ALTER TRIGGER` изменяет свойства существующего триггера.

Предложение `RENAME` переименовывает данный триггер, не затрагивая его определение. Если таблица, к которой относится триггер, является секционированной, соответствующие «клонированные» триггеры в секциях также переименовываются.

Предложение `DEPENDS ON EXTENSION` помечает триггер как зависимый от расширения, так что при удалении расширения будет автоматически удаляться и триггер.

Изменять свойства триггера может только владелец таблицы, с которой работает триггер.

Параметры

`имя`

Имя существующего триггера, подлежащего изменению.

`имя_таблицы`

Имя таблицы, с которой работает триггер.

`новое_имя`

Новое имя триггера.

`имя_расширения`

Имя расширения, от которого будет зависеть триггер (или не будет, если указано `NO`). Триггер, помеченный как зависимый от расширения, автоматически удаляется при удалении расширения.

Примечания

Возможность временно включать или отключать триггер предоставляется командой `ALTER TABLE`, а не `ALTER TRIGGER`, так как `ALTER TRIGGER` не позволяет удобным образом выразить указание включить или отключить все триггеры таблицы сразу.

Примеры

Переименование существующего триггера:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Обозначение триггера как зависимого от расширения:

```
ALTER TRIGGER emp_stamp ON emp DEPENDS ON EXTENSION emplib;
```

Совместимость

`ALTER TRIGGER` — реализованное в Postgres Pro расширение стандарта SQL.

См. также

ALTER TABLE

ALTER TYPE

ALTER TYPE — изменить определение типа

Синтаксис

```
ALTER TYPE имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER TYPE имя RENAME TO новое_имя
ALTER TYPE имя SET SCHEMA новая_схема
ALTER TYPE имя RENAME ATTRIBUTE имя_атрибута TO новое_имя_атрибута [ CASCADE |
 RESTRICT ]
ALTER TYPE имя действие [, ... ]
ALTER TYPE имя ADD VALUE [ IF NOT EXISTS ] новое_значение_перечисления [ { BEFORE |
 AFTER } соседнее_значение_перечисления ]
ALTER TYPE имя RENAME VALUE существующее_значение_перечисления
 TO новое_значение_перечисления
ALTER TYPE имя SET ( свойство = значение [, ... ] )

Где действие может быть следующим:

    ADD ATTRIBUTE имя_атрибута тип_данных [ COLLATE правило_сортировки ] [ CASCADE |
 RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] имя_атрибута [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE имя_атрибута [ SET DATA ] TYPE тип_данных
  [ COLLATE правило_сортировки ] [ CASCADE | RESTRICT ]
```

Описание

`ALTER TYPE` изменяет определение существующего типа. Эта команда имеет несколько разновидностей:

`OWNER`

Эта форма меняет владельца типа.

`RENAME`

Эта форма меняет имя типа.

`SET SCHEMA`

Эта форма переносит тип в другую схему.

`RENAME ATTRIBUTE`

Эта форма работает только с составными типами. Она меняет имя отдельного атрибута такого типа.

`ADD ATTRIBUTE`

Эта форма добавляет в составной тип новый атрибут с тем же синтаксисом, что и `CREATE TYPE`.

`DROP ATTRIBUTE [ IF EXISTS ]`

Эта форма удаляет атрибут из составного типа. Если указано `IF EXISTS` и атрибут не существует, это не считается ошибкой. В этом случае выдаётся только замечание.

`ALTER ATTRIBUTE ... SET DATA TYPE`

Эта форма меняет тип атрибута составного типа.

`ADD VALUE [ IF NOT EXISTS ] [ BEFORE | AFTER ]`

Эта форма добавляет новое значение в тип-перечисление. Порядок нового значения в перечислении можно указать, добавив `BEFORE` (перед) или `AFTER` (после) с одним из существующих значений. Если такое указание отсутствует, новый элемент добавляется в конец списка значений.

С указанием `IF NOT EXISTS`, если тип уже содержит новое значение, ошибки не произойдёт: будет выдано замечание и ничего больше. Без этого указания, если такое значение уже представлено, возникнет ошибка.

`RENAME VALUE`

Эта форма переименовывает значение в типе-перечислении. Позиция значения в порядке перечисления при этом не меняется. Если это значение отсутствует или в перечислении уже есть новое имя, выдаётся ошибка.

`SET (` *`свойство`* `=` *`значение`* `[, ... ] )`

Эта форма применима только к базовым типам. Она позволяет изменить некоторые свойства типа, которые можно задать в `CREATE TYPE`, а именно:

- `RECEIVE` позволяет задать имя функции двоичного ввода, а значение `NONE` удаляет ссылку на такую функцию. Менять это свойство разрешено только суперпользователям.
- `SEND` позволяет задать имя функции двоичного вывода, а значение `NONE` удаляет ссылку на такую функцию. Менять это свойство разрешено только суперпользователям.
- `TYPMOD_IN` позволяет задать имя функции, предназначенной для ввода модификатора типа, а значение `NONE` удаляет ссылку на такую функцию. Менять это свойство разрешено только суперпользователям.
- `TYPMOD_OUT` позволяет задать имя функции, предназначенной для вывода модификатора типа, а значение `NONE` удаляет ссылку на такую функцию. Менять это свойство разрешено только суперпользователям.
- `ANALYZE` позволяет задать имя функции сбора статистики для этого типа, а значение `NONE` удаляет ссылку на такую функцию. Менять это свойство разрешено только суперпользователям.
- `SUBSCRIPT` позволяет задать имя функции-обработчика обращения по индексу для этого типа, а значение `NONE` удаляет ссылку на такую функцию. Менять это свойство разрешено только суперпользователям.
- `STORAGE`может принимать значения `plain`, `extended`, `external` и `main` (их описание можно найти в Разделе 63.2). При этом менять вариант `plain` на какой-либо другой разрешено только суперпользователям (так как для этого требуется, чтобы функции, реализующие тип на C, поддерживали TOAST), а сменить любое другое значение на `plain` не разрешается вовсе (так как значения этого типа в базе могут уже храниться в виде TOAST). Заметьте, что изменение этого свойства само по себе не влияет на сохранённые данные, оно меняет только стратегию TOAST по умолчанию, которая будет использоваться для столбцов, создаваемых в будущем. Изменить стратегию TOAST для существующих столбцов позволяет команда ALTER TABLE.

Более подробно эти свойства типов описаны в CREATE TYPE. Заметьте, что везде, где применимо, изменения свойств базового типа будут автоматически отражаться в основанных на этом типе доменах.

Операции `ADD ATTRIBUTE`, `DROP ATTRIBUTE` и `ALTER ATTRIBUTE` можно объединить в один список множественных изменений для параллельного выполнения. Например, в одной команде можно добавить сразу несколько атрибутов и/или изменить тип нескольких атрибутов.

Выполнить `ALTER TYPE` может только владелец соответствующего типа. Чтобы сменить схему типа, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме типа. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать тип. Однако суперпользователь может сменить владельца типа в любом случае.) Чтобы добавить атрибут или изменить его тип, также требуется иметь право `USAGE` для типа атрибута.

Параметры

`имя`

Имя (возможно, дополненное схемой) существующего типа, подлежащего изменению.

`новое_имя`

Новое имя типа.

`новый_владелец`

Имя пользователя, назначаемого новым владельцем типа.

`новая_схема`

Новая схема типа.

`имя_атрибута`

Имя атрибута, подлежащего добавлению, изменению или удалению.

`новое_имя_атрибута`

Новое имя атрибута

`тип_данных`

Тип данных добавляемого атрибута, либо новый тип данных изменяемого атрибута.

`новое_значение_перечисления`

Новое значение добавляется в список значений перечисления или для существующего значения задаётся новое имя. Как и все элементы перечисления, оно должно заключаться в кавычки.

`соседнее_значение_перечисления`

Существующие значение в перечислении, непосредственно перед или после которого по порядку перечисления будет добавлено новое значение. Как и все элементы перечисления, оно должно заключаться в кавычки.

`существующее_значение_перечисления`

Существующее значение в перечислении, которое будет переименовано. Как и все элементы перечисления, оно должно заключаться в кавычки.

`свойство`

Имя изменяемого свойства базового типа; возможные значения перечислены выше.

`CASCADE`

Автоматически распространять действие операции на типизированные таблицы, имеющий данный тип, и их потомки.

`RESTRICT`

Отказать в выполнении операции, если изменяемый тип является типом типизированной таблицы. Это поведение по умолчанию.

Примечания

Если `ALTER TYPE ... ADD VALUE` (форма, добавляющая в тип-перечисление новое значение) выполняется внутри блока транзакции, новое значение нельзя будет использовать до фиксирования транзакции.

Сравнения с добавленными значениями перечисления иногда бывают медленнее сравнений, в которых задействуются только начальные члены типа-перечисления. Обычно это происходит, только если `BEFORE` или `AFTER` устанавливает порядок нового элемента не в конце списка. Однако иногда это наблюдается даже тогда, когда новое значение добавляется в конец списка (это происходит, если счётчик OID «прокручивается» с момента изначального создания типа-перечисления). Это замедление обычно несущественное, но если это важно, вернуть максимальную производительность можно, удалив и создав заново это перечисление, либо выгрузив копию базы данных и загрузив её вновь.

Примеры

Переименование типа данных:

```
ALTER TYPE electronic_mail RENAME TO email;
```

Смена владельца типа `email` на `joe`:

```
ALTER TYPE email OWNER TO joe;
```

Смена схемы типа `email` на `customers`:

```
ALTER TYPE email SET SCHEMA customers;
```

Добавление нового атрибута в составной тип:

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

Добавление нового значения в тип-перечисление, в определённое положение по порядку:

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

Переименование значения в перечислении:

```
ALTER TYPE colors RENAME VALUE 'purple' TO 'mauve';
```

Создание функций двоичного ввода-вывода для существующего базового типа:

```
CREATE FUNCTION mytypesend(mytype) RETURNS bytea ...;
CREATE FUNCTION mytyperecv(internal, oid, integer) RETURNS mytype ...;
ALTER TYPE mytype SET (
    SEND = mytypesend,
    RECEIVE = mytyperecv
);
```

Совместимость

Формы команды, предназначенные для добавления и удаления атрибутов, являются частью стандарта SQL; другие формы относятся к расширениям Postgres Pro.

См. также

CREATE TYPE, DROP TYPE

ALTER USER

ALTER USER — изменить роль в базе данных

Синтаксис

```
ALTER USER указание_роли [ WITH ] параметр [ ... ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль' | PASSWORD NULL
    | VALID UNTIL 'дата_время'
    | PROFILE имя_профиля
    | ACCOUNT UNLOCK | ACCOUNT LOCK

ALTER USER имя RENAME TO новое_имя

ALTER USER { указание_роли | ALL } [ IN DATABASE имя_бд ] SET параметр_конфигурации
 { TO | = } { значение | DEFAULT }
ALTER USER { указание_роли | ALL } [ IN DATABASE имя_бд ] SET параметр_конфигурации
 FROM CURRENT
ALTER USER { указание_роли | ALL } [ IN DATABASE имя_бд ] RESET параметр_конфигурации
ALTER USER { указание_роли | ALL } [ IN DATABASE имя_бд ] RESET ALL

Здесь указание_роли:

    имя_роли
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Описание

Оператор `ALTER USER` теперь стал синонимом оператора `ALTER ROLE`.

Совместимость

Оператор `ALTER USER` является расширением Postgres Pro. В стандарте SQL определение пользователей считается зависимым от реализации.

См. также

ALTER ROLE

ALTER USER MAPPING

ALTER USER MAPPING — изменить определение сопоставления пользователей

Синтаксис

```
ALTER USER MAPPING FOR { имя_пользователя | USER | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER | PUBLIC }
    SERVER имя_сервера
    OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )
```

Описание

`ALTER USER MAPPING` изменяет определение сопоставления пользователей.

Владелец стороннего сервера может изменить сопоставление любых пользователей на этом сервере. Кроме того, пользователь может изменить сопоставление для своего собственного имени пользователя, если он наделён правом `USAGE` на данном сервере.

Параметры

имя_пользователя

Имя пользователя для сопоставления. Значения `CURRENT_ROLE`, `CURRENT_USER` и `USER` соответствуют имени текущего пользователя. Значение `PUBLIC` соответствует именам всех текущих и будущих пользователей системы.

имя_сервера

Имя сервера, для которого меняется сопоставление пользователей.

`OPTIONS ( [ ADD | SET | DROP ] параметр ['значение'] [, ... ] )`

Эти формы меняют параметры сопоставления пользователей. Новые параметры переопределяют любые определённые ранее. Возможные операции с параметрами: `ADD` (добавить), `SET` (установить) и `DROP` (удалить). По умолчанию подразумевается `ADD`. Имена параметров должны быть уникальными; кроме того, они проверяются обёрткой сторонних данных.

Примеры

Изменение пароля в сопоставлении пользователя `bob` на сервере `foo`:

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

Совместимость

`ALTER USER MAPPING` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Однако есть небольшое синтаксическое различие: в стандарте ключевое слово `FOR` опускается. Но так как и в `CREATE USER MAPPING`, и в `DROP USER MAPPING` слово `FOR` находится в аналогичных позициях, а IBM DB2 (ещё одна популярная реализация SQL/MED) требует его и для `ALTER USER MAPPING`, Postgres Pro в этом аспекте отклоняется от стандарта ради согласованности и совместимости.

См. также

CREATE USER MAPPING, DROP USER MAPPING

ALTER VIEW

ALTER VIEW — изменить определение представления

Синтаксис

```
ALTER VIEW [ IF EXISTS ] имя ALTER [ COLUMN ] имя_столбца SET DEFAULT выражение
ALTER VIEW [ IF EXISTS ] имя ALTER [ COLUMN ] имя_столбца DROP DEFAULT
ALTER VIEW [ IF EXISTS ] имя OWNER TO { новый_владелец | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER VIEW [ IF EXISTS ] имя RENAME [ COLUMN ] имя_столбца TO новое_имя_столбца
ALTER VIEW [ IF EXISTS ] имя RENAME TO новое_имя
ALTER VIEW [ IF EXISTS ] имя SET SCHEMA новая_схема
ALTER VIEW [ IF EXISTS ] имя SET ( имя_параметра_представления
 [= значение_параметра_представления] [, ... ] )
ALTER VIEW [ IF EXISTS ] имя RESET ( имя_параметра_представления [, ... ] )
```

Описание

`ALTER VIEW` изменяет различные дополнительные свойства представления. (Для изменения запроса, определяющего представление, используйте команду `CREATE OR REPLACE VIEW`.)

Выполнить `ALTER VIEW` может только владелец представления. Чтобы сменить схему представления, необходимо также иметь право `CREATE` в новой схеме. Чтобы сменить владельца, необходимо иметь право `SET ROLE` для новой роли-владельца, а эта роль должна иметь право `CREATE` в схеме представления. (С такими ограничениями при смене владельца не происходит ничего такого, что нельзя было бы сделать, имея право удалить и вновь создать представление. Однако суперпользователь может сменить владельца представления в любом случае.)

Параметры

имя

Имя существующего представления (возможно, дополненное схемой).

имя_столбца

Имя существующего столбца.

новое_имя_столбца

Новое имя существующего столбца.

`IF EXISTS`

Не считать ошибкой, если представление не существует. В этом случае будет выдано замечание.

`SET`/`DROP DEFAULT`

Эти формы устанавливают или удаляют значение по умолчанию в заданном столбце. Значение по умолчанию подставляется в команды `INSERT` и `UPDATE`, вносящие данные в представление, до применения каких-либо правил или триггеров в этом представлении. Таким образом, значения по умолчанию в представлении имеют приоритет перед значениями по умолчанию в нижележащих отношениях.

новый_владелец

Имя пользователя, назначаемого новым владельцем представления.

новое_имя

Новое имя представления.

новая_схема

Новая схема представления.

`SET ( имя_параметра_представления [= значение_параметра_представления] [, ... ] )`
`RESET ( имя_параметра_представления [, ... ] )`

Устанавливает или сбрасывает параметры представления. В настоящее время поддерживаются параметры:

`check_option` (enum)

Изменяет параметр проверки представления. Допустимые значения: `local` (локальная) или `cascaded` (каскадная).

`security_barrier` (boolean)

Изменяет свойство представления, включающее барьер безопасности. Значение должно быть логическим: `true` или `false`.

`security_invoker` (boolean)

Изменяет свойство представления, включающее контекст безопасности вызывающего. Значение должно быть логическим: `true` или `false`.

Примечания

По историческим причинам команду `ALTER TABLE` можно использовать и с представлениями; но единственно допустимые для работы с представлениями вариации `ALTER TABLE` равносильны вышеперечисленным командам.

Примеры

Переименование представления `foo` в `bar`:

```
ALTER VIEW foo RENAME TO bar;
```

Добавление значения столбца по умолчанию в изменяемое представление:

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- в ts окажется значение NULL
INSERT INTO a_view(id) VALUES(2);  -- в ts окажется текущее время
```

Совместимость

`ALTER VIEW` — реализованное в Postgres Pro расширение стандарта SQL.

См. также

CREATE VIEW, DROP VIEW

ANALYZE

ANALYZE — собрать статистику по базе данных

Синтаксис

```
ANALYZE [ ( параметр [, ...] ) ] [ таблица_и_столбцы [, ...] ]

Здесь допускается параметр:

    VERBOSE [ boolean ]
    SKIP_LOCKED [ boolean ]
    BUFFER_USAGE_LIMIT размер

и таблица_и_столбцы:

    имя_таблицы [ ( имя_столбца [, ...] ) ]
```

Описание

`ANALYZE` собирает статистическую информацию о содержимом таблиц в базе данных и сохраняет результаты в системном каталоге `pg_statistic`. Впоследствии планировщик запросов будет использовать эту статистику для выбора наиболее эффективных планов выполнения запросов.

Без списка *`таблица_и_столбцы`* команда `ANALYZE` обрабатывает все таблицы и материализованные представления в текущей базе данных, на анализ которых текущий пользователь имеет право. Со списком `ANALYZE` обрабатывает только указанные в нём таблицы. Дополнительно можно задать для таблицы список имён столбцов, в этом случае статистика будет собираться только по этим столбцам.

Параметры

`VERBOSE`

Включает вывод сообщений о процессе выполнения.

`SKIP_LOCKED`

Указывает, что команда `ANALYZE` не должна ждать освобождения конфликтующих блокировок, начиная обработку отношения: если отношение не удаётся заблокировать сразу, без ожидания, оно пропускается. Заметьте, что даже с этим указанием `ANALYZE` может заблокироваться, открывая индексы отношения или получая выборку строк из секций, потомков в иерархии наследования или некоторых видов сторонних таблиц. Учтите также, что при наличии конфликтующей блокировки в секционированной таблице команда `ANALYZE` пропускает все её секции, тогда как обычно все они обрабатываются.

`BUFFER_USAGE_LIMIT`

Указывает размер кольцевого буфера *стратегии доступа к буферам* (Buffer Access Strategy) для `ANALYZE`. Этот размер используется для расчёта количества общих буферов, переиспользуемых в рамках этой стратегии. С значением `0` использование стратегии доступа к буферам отключается. Если не задать этот параметр, `ANALYZE` использует значение из vacuum_buffer_usage_limit. С большим значением параметра `ANALYZE` может выполняться быстрее, но при слишком большом значении многие полезные страницы могут вытесняться из общих буферов. Минимальное значение — `128 kB`, максимальное — `16 GB`.

`boolean`

Включает или отключает заданный параметр. Для включения параметра можно написать `TRUE`, `ON` или `1`, а для отключения — `FALSE`, `OFF` или `0`. Значение *`boolean`* можно опустить, в этом случае подразумевается `TRUE`.

`размер`

Задаёт объём памяти в килобайтах. Можно также указать в виде строки, содержащей числовой размер, за которым следует одна из следующих единиц информации: `B` (байты), `kB` (килобайты), `MB` (мегабайты), `GB` (гигабайты) или `TB` (терабайты).

`имя_таблицы`

Имя (возможно, дополненное схемой) определённой таблицы, подлежащей анализу. Если опущено, анализироваться будут все обычные и секционированные таблицы, а также материализованные представления в текущей базе данных (но не сторонние таблицы). Если задано имя секционированной таблицы, обновлена будет как статистика наследования для этой таблицы, так и статистика отдельных её секций.

`имя_столбца`

Имя столбца, подлежащего анализу. По умолчанию анализируются все столбцы.

Выводимая информация

С указанием `VERBOSE` команда `ANALYZE` выдаёт сообщения о процессе анализа, отмечая текущую обрабатываемую таблицу. Также она выводит различные статистические сведения о таблицах.

Примечания

Чтобы осуществить анализ таблицы, обычно нужно иметь право `MAINTAIN` для таблицы. Однако владельцам баз данных также разрешено выполнять анализ всех таблиц в своих базах, за исключением общих каталогов. `ANALYZE` пропускает все таблицы, на анализ которых текущий пользователь не имеет прав.

Сторонние таблицы анализируются только при явном указании и только если соответствующая обёртка сторонних данных поддерживает команду `ANALYZE`. Если эта команда не поддерживается, при выполнении `ANALYZE` выводится предупреждение и больше ничего не происходит.

В стандартной конфигурации Postgres Pro работающий демон автоочистки (см. Подраздел 23.1.6) запускает анализ таблиц автоматически, когда они изначально заполняются данными, и периодически, по мере того, как они меняются. Если автоочистка отключена, рекомендуется запускать `ANALYZE` время от времени, либо после кардинальных изменений в таблице. Точная статистика помогает планировщику выбрать наиболее эффективный план запроса и тем самым увеличивает скорость выполнения запроса. Обычно для баз, где данные в основном читаются, выполняют `VACUUM` и `ANALYZE` раз в день, во время наименьшей активности. (Этого будет недостаточно, если данные меняются очень активно.)

На время выполнения команды `ANALYZE` search_path временно меняется на `pg_catalog, pg_temp`.

`ANALYZE` запрашивает для целевой таблицы блокировку только на чтение, поэтому данная команда может выполняться параллельно с другими не DDL-операциями с таблицей.

Статистика, собираемая командой `ANALYZE`, обычно включает список из нескольких самых частых значений в каждом столбце и гистограмму, отражающую примерное распределение данных во всех столбцах. Один или оба этих элемента статистики могут быть опущены, если `ANALYZE` сочтёт их неинтересными (например, в столбце уникального ключа нет повторяющихся значений), либо если тип данных столбца не поддерживает соответствующие операторы. Более подробно статистика описывается в Главе 23.

В больших таблицах `ANALYZE` не просматривает все строки, а обрабатывает только небольшую случайную выборку. Это позволяет проанализировать за короткое время даже очень большие таблицы. Однако учтите, что такая статистика будет лишь приблизительной и может немного меняться при каждом выполнении `ANALYZE`, даже если фактическое содержимое таблицы остаётся неизменным. Это может приводить к небольшим изменениям в оценках стоимости запросов, выводимых командой `EXPLAIN`. В редких случаях вследствие этой недетерменированности планировщик меняет свой выбор после выполнения `ANALYZE`. Чтобы избежать этого, увеличьте объём статистики, собираемой командой `ANALYZE`, как описано ниже.

Количеством статистики можно управлять, настраивая конфигурационную переменную default_statistics_target или устанавливая ориентир статистики на уровне столбцов командой `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS` (см. ALTER TABLE). Ориентир задаёт максимальное число записей в списке наиболее распространённых значений и максимальное число интервалов в гистограмме. По умолчанию значение ориентира равно 100, но его можно увеличить или уменьшить в поисках баланса между точностью оценок планировщика и временем, требующимся для выполнения `ANALYZE`, а также объёмом статистики в таблице `pg_statistic`. Если установить ориентир статистики равным нулю, статистика по таким столбцам собираться не будет. Это может быть полезно для столбцов, которые никогда не фигурируют в предложениях `WHERE`, `GROUP BY` и `ORDER BY`, так как планировщик никогда не будет использовать их статистику.

Число строк таблицы, выбираемых для подготовки статистики, определяется наибольшим ориентиром статистики по всем анализируемым столбцам этой таблицы. Увеличение ориентира приводит к пропорциональному увеличению времени и пространства, требуемого для выполнения `ANALYZE`.

Одним из показателей, оцениваемых командой `ANALYZE`, является число различных значений, встречающихся в каждом столбце. Так как рассматривается только подмножество всех строк, эта оценка иногда может быть весьма неточной, даже при самых больших ориентирах статистики. Если эта неточность приводит к плохому выбору плана запроса, более точное значение можно определить вручную и затем задать его командой `ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...)`.

Если у анализируемой таблицы есть потомки, `ANALYZE` соберёт два набора статистик: один по строкам только родительской таблицы, а второй по строкам родительской и всех дочерних таблиц. Второй набор статистики необходим для планирования запросов, обращающихся к дереву наследования как одному целому. Сами дочерние таблицы в этом случае отдельно не анализируются. Демон автоочистки, однако, принимая решение об автоматическом запуске анализа, будет учитывать операции добавления или изменения данных только в самой родительской таблице. Если именно в этой таблице изменение и добавление происходят редко, наследуемая статистика может терять актуальность, если не запускать `ANALYZE` вручную.

Для секционированных таблиц `ANALYZE` собирает статистику, обрабатывая выборки строки из всех секций, а также рекурсивно обновляет статистику во всех секциях. Даже в многоуровневой иерархии секционирования каждая конечная секция анализируется только один раз. Отдельно для родительских таблиц (без рассмотрения данных их секций) статистика не собирается, так как секционирование гарантирует, что они пустые.

Процесс автоочистки не затрагивает секционированные таблицы и родительские таблицы в иерархии наследования, если изменения происходят только в дочерних таблицах. Поэтому, чтобы статистика всей иерархии наследования была актуальной, обычно необходимо периодически выполнять `ANALYZE` вручную.

Если какие-либо из дочерних таблиц или секций являются сторонними таблицами и их обёртки сторонних данных не поддерживают `ANALYZE`, эти дочерние таблицы игнорируются при сборе статистики наследования.

Если анализируемая таблица оказалась пустой, `ANALYZE` не будет обновлять статистику по этой таблице; в базе сохранится статистика, собранная ранее.

Каждый процесс, выполняющий операцию `ANALYZE`, будет выдавать информацию о ходе её выполнения, отображаемую в представлении `pg_stat_progress_analyze`. За подробностями обратитесь к Подразделу 26.4.1.

Совместимость

Оператор `ANALYZE` отсутствует в стандарте SQL.

До версии Postgres Pro 11 использовался и по-прежнему поддерживается следующий синтаксис:

```
ANALYZE [ VERBOSE ] [ таблица_и_столбцы [, ...] ]
```

См. также

VACUUM, vacuumdb, Подраздел 18.4.4, Подраздел 23.1.6, Подраздел 26.4.1

BEGIN

BEGIN — начать блок транзакции

Синтаксис

```
BEGIN [ WORK | TRANSACTION ] [ режим_транзакции [, ...] ]

Где режим_транзакции может быть следующим:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Описание

`BEGIN` начинает блок транзакции, то есть обозначает, что все операторы после команды `BEGIN` и до явной команды `COMMIT` или `ROLLBACK` будут выполняться в одной транзакции. По умолчанию (без `BEGIN`) Postgres Pro выполняет транзакции в режиме «autocommit» (автофиксация), то есть каждый оператор выполняется в своей отдельной транзакции, которая неявно фиксируется в конце оператора (если оператор был выполнен успешно; в противном случае транзакция откатывается).

В блоке транзакции операторы выполняются быстрее, так как для запуска/фиксации транзакции производится масса операций, нагружающих процессор и диск. Кроме того, выполнение нескольких операторов в одной транзакции позволяет обеспечить целостность при внесении серии связанных изменений; другие сеансы не видят промежуточное состояние, когда произошли ещё не все связанные изменения.

Если указан уровень изоляции, режим чтения/записи или устанавливается отложенный режим, новая транзакция получает те же характеристики, что и после выполнения `SET TRANSACTION`.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

За описанием других параметров обратитесь к SET TRANSACTION.

Примечания

`START TRANSACTION` делает то же, что и `BEGIN`.

Для завершения блока транзакции используйте `COMMIT` или `ROLLBACK`.

При попытке выполнить `BEGIN` внутри уже начатого блока транзакции будет выдано предупреждение, а состояние транзакции не изменится. Для вложения подтранзакций внутри блока транзакций используйте точки сохранения (см. SAVEPOINT).

Для сохранения обратной совместимости допускается перечисление *`режимов_транзакции`* без запятых.

Примеры

Начало блока транзакции:

```
BEGIN;
```

Совместимость

`BEGIN` — это языковое расширение Postgres Pro. Эта команда равнозначна соответствующей стандарту SQL команде `START TRANSACTION`, в описании которой можно найти дополнительные сведения о совместимости.

Значение `DEFERRABLE` параметра *`режим_транзакции`* является языковым расширением Postgres Pro.

По стечению обстоятельств ключевое слово `BEGIN` имеет другое значение во встраиваемом SQL, поэтому при портировании приложений баз данных рекомендуется внимательно сверить семантику транзакций.

См. также

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

CALL

CALL — вызвать процедуру

Синтаксис

```
CALL имя ( [ аргумент ] [, ...] )
```

Описание

`CALL` вызывает процедуру.

Если у процедуры есть выходные параметры, возвращается строка результата, содержащая значения этих параметров.

Параметры

`имя`

Имя процедуры (возможно, дополненное схемой).

`аргумент`

Выражение, определяющее аргумент для вызова процедуры.

В аргументе может передаваться имя параметра в записи *`имя`* `=>` *`значение`*. Это работает так же, как и с обычными вызовами функций; за подробностями обратитесь к Разделу 4.3.

Аргументы должны задаваться для всех параметров процедуры, для которых не определены значения по умолчанию, включая параметры `OUT`. Однако аргументы, соответствующие параметрам `OUT`, не вычисляются, поэтому вместо них обычно часто просто пишут `NULL`. (Если передавать в параметрах `OUT` какие-то другие значения, возможны проблемы совместимости с будущими версиями Postgres Pro.)

Примечания

Чтобы вызывать процедуру, пользователь должен иметь право `EXECUTE` для неё.

Для вызова функции (не процедуры) следует использовать `SELECT`.

Если `CALL` выполняется в блоке транзакции, вызываемая процедура не может выполнять операторы управления транзакциями. Такие операторы допускаются, только если `CALL` выполняется в собственной транзакции.

Выходные параметры команд `CALL` в PL/pgSQL обрабатываются по-другому; см. Подраздел 41.6.3.

Примеры

```
CALL do_db_maintenance();
```

Совместимость

Команда `CALL` соответствует стандарту SQL, за исключением особенности обработки выходных параметров. Стандарт говорит, что для получения значений таких параметров нужно указывать переменные.

См. также

CREATE PROCEDURE

CHECKPOINT

CHECKPOINT — произвести контрольную точку в журнале предзаписи

Синтаксис

```
CHECKPOINT
```

Описание

Контрольная точка — это момент в последовательности событий в журнале предзаписи, когда все файлы данных приводятся в актуальное состояние, соответствующее информации в журнале. При этом все файлы данных сохраняются на диск. Более подробно о том, что происходит во время контрольной точки, можно узнать в Разделе 27.5.

Команда `CHECKPOINT` приводит к принудительному выполнению контрольной точки в момент вызова, не дожидаясь периодической контрольной точки, производимой системой по графику (это настраивается параметрами в Подраздел 18.5.2). Команда `CHECKPOINT` не предназначена для применения при обычном использовании базы данных.

Если `CHECKPOINT` выполняется в процессе восстановления, вместо записи новой контрольной точки производится точка перезапуска (см. Раздел 27.5).

Выполнять `CHECKPOINT` могут только суперпользователи и пользователи с правами роли `pg_checkpoint`.

Совместимость

Команда `CHECKPOINT` является языковым расширением Postgres Pro.

CLOSE

CLOSE — закрыть курсор

Синтаксис

```
CLOSE { имя | ALL }
```

Описание

`CLOSE` освобождает ресурсы, связанные с открытым курсором. Когда курсор закрыт, никакие операции с ним невозможны. Закрывать курсор следует, когда он становится ненужным.

Все не удерживаемые открытые курсоры закрываются неявно при завершении транзакции командами `COMMIT` или `ROLLBACK`. Удерживаемый курсор закрывается неявно, если транзакция, его создавшая, прерывается командой `ROLLBACK`. Если создавшая его транзакция завершается успешной фиксацией, удерживаемый курсор остаётся открытым до явного вызова команды `CLOSE` или отключения клиента.

Параметры

`имя`

Имя открытого курсора, который будет закрыт.

`ALL`

Закрывает все открытые курсоры.

Примечания

В Postgres Pro нет явной команды `OPEN` для курсора; курсор считается открытым при объявлении. Чтобы объявить курсор, используйте оператор `DECLARE`.

Получить список всех доступных курсоров можно, обратившись к системному представлению `pg_cursors`.

Если курсор был закрыт после точки сохранения, а затем произошёл откат к этой точке, действие команды `CLOSE` не отменяется; то есть курсор остаётся закрытым.

Примеры

Следующая команда закрывает курсор `liahona`:

```
CLOSE liahona;
```

Совместимость

Оператор `CLOSE` полностью соответствует стандарту SQL. `CLOSE ALL` является расширением Postgres Pro.

См. также

DECLARE, FETCH, MOVE

CLUSTER

CLUSTER — кластеризовать таблицу согласно индексу

Синтаксис

```
CLUSTER [ ( параметр [, ...] ) ] [ имя_таблицы [ USING имя_индекса ] ]

Здесь допускается параметр:

    VERBOSE [ boolean ]
```

Описание

Оператор `CLUSTER` указывает Postgres Pro кластеризовать таблицу, заданную параметром *`имя_таблицы`*, согласно индексу, заданному параметром *`имя_индекса`*. Указанный индекс уже должен быть определён в таблице *`имя_таблицы`*.

В результате кластеризации таблицы её содержимое физически переупорядочивается в зависимости от индекса. Кластеризация является одноразовой операцией: последующие изменения в таблице нарушают порядок кластеризации. Другими словами, система не пытается автоматически сохранять порядок новых или изменённых строк в соответствии с индексом. (Если такое желание возникает, можно периодически повторять кластеризацию, выполняя команду снова. Кроме того, если для заданной таблицы установить параметр `FILLFACTOR` меньше 100%, это может помочь сохранить порядок кластеризации при изменениях, так как изменяемые строки будут помещаться в ту же страницу, если в ней достаточно места.)

Когда таблица кластеризована, Postgres Pro запоминает, по какому именно индексу. Форма `CLUSTER` *`имя_таблицы`* повторно кластеризует таблицу по тому же индексу. Для установки индекса, который будет использоваться для будущих операций кластеризации, или очистки предыдущего значения можно также применить команду `CLUSTER` или формы `SET WITHOUT CLUSTER` команды `ALTER TABLE`.

`CLUSTER` без *`имени_таблицы`* повторно кластеризует все ранее кластеризованные таблицы в текущей базе данных, на которые у вызывающего пользователя есть права. Эту форму `CLUSTER` нельзя выполнять внутри блока транзакции.

В процессе кластеризации таблицы для неё запрашивается блокировка `ACCESS EXCLUSIVE`. Это препятствует выполнению всех других операций (чтению и записи) с таблицей до завершения `CLUSTER`.

Параметры

`имя_таблицы`

Имя таблицы (возможно, дополненное схемой).

`имя_индекса`

Имя индекса.

`VERBOSE`

Выводит отчёт о процессе кластеризации по мере обработки таблиц.

`boolean`

Включает или отключает заданный параметр. Для включения параметра можно написать `TRUE`, `ON` или `1`, а для отключения — `FALSE`, `OFF` или `0`. Значение *`boolean`* можно опустить, в этом случае подразумевается `TRUE`.

Примечания

Чтобы кластеризовать таблицу, необходимо иметь право `MAINTAIN` для этой таблицы.

В случаях, когда происходит обращение к случайным единичным строкам таблицы, фактический порядок данных в этой таблице не важен. Но если обращения к одним данным происходят чаще, чем к другим, и есть индекс, который собирает их вместе, применение команды `CLUSTER` может быть полезным. Например, когда из таблицы запрашивается диапазон индексированных значений, либо одно индексированное значение, которому соответствуют несколько строк, `CLUSTER` может помочь, так как страница таблицы, найденная по индексу для первой искомой строки, скорее всего, будет содержать и все остальные искомые строки. Таким образом, кластеризация помогает соптимизировать обращения к диску и ускорить запросы.

`CLUSTER` может переупорядочить таблицу, выполнив либо сканирование указанного индекса, либо (для индексов-B-деревьев) последовательное сканирование, а затем сортировку. Наилучший по скорости вариант будет выбран, исходя из имеющейся статистической информации и параметров планировщика.

На время выполнения команды `CLUSTER` search_path временно меняется на `pg_catalog, pg_temp`.

Когда выбирается сканирование индекса, создаётся временная таблица, содержащая данные целевой таблицы по порядку индекса. Также создаются копии всех индексов таблицы. Таким образом, для этой операции требуется объём дискового пространства не меньше, чем размер таблицы и индексов в сумме.

В случае выбора последовательного сканирования и сортировки создаётся ещё и временный файл для сортировки, так что пиковым требованием будет удвоенный размер таблицы плюс размер индексов. Этот метод часто быстрее, чем сканирование по индексу, но если требование к дисковому пространству неприемлемо, можно отключить его выбор, временно установив enable_sort в `off`.

Перед кластеризацией рекомендуется установить в maintenance_work_mem достаточно большое значение (но не больше, чем объём ОЗУ, который вы хотите выделить для операции `CLUSTER`).

Так как планировщик записывает статистику, связанную с порядком таблиц, для вновь кластеризуемых таблиц рекомендуется запускать `ANALYZE`. В противном случае планировщик может ошибиться с выбором плана запроса.

Так как `CLUSTER` запоминает, по каким индексам кластеризованы таблицы, достаточно лишь один раз вручную кластеризовать нужные таблицы, а затем настроить периодический скрипт обслуживания, который будет выполнять `CLUSTER` без параметров, с тем чтобы эти таблицы регулярно кластеризовались.

Каждый процесс, выполняющий операцию `CLUSTER`, будет выдавать информацию о ходе её выполнения, отображаемую в представлении `pg_stat_progress_cluster`. За подробностями обратитесь к Подразделу 26.4.2.

При кластеризации секционированной таблицы каждая из её секций кластеризуется согласно секции заданного секционированного индекса. Индекс при кластеризации секционированной таблицы должен указываться обязательно. `CLUSTER` нельзя выполнять внутри блока транзакции.

Примеры

Кластеризация таблицы `employees` согласно её индексу `employees_ind`:

```
CLUSTER employees USING employees_ind;
```

Кластеризация таблицы `employees` согласно тому же индексу, что был использован ранее:

```
CLUSTER employees;
```

Кластеризация всех таблиц в базе данных, что были кластеризованы ранее:

```
CLUSTER;
```

Совместимость

Оператор `CLUSTER` отсутствует в стандарте SQL.

До версии PostgreSQL 17 использовался и по-прежнему поддерживается следующий синтаксис:

```
CLUSTER [ VERBOSE ] [ имя_таблицы [ USING имя_индекса ] ]
```

До версии PostgreSQL 8.3 использовался и по-прежнему поддерживается следующий синтаксис:

```
CLUSTER имя_индекса ON имя_таблицы
```

См. также

clusterdb, Подраздел 26.4.2

COMMENT

COMMENT — задать или изменить комментарий объекта

Синтаксис

```
COMMENT ON
{
  ACCESS METHOD имя_объекта |
  AGGREGATE имя_агрегатной_функции ( сигнатура_агр_функции ) |
  CAST (исходный_тип AS целевой_тип) |
  COLLATION имя_объекта |
  COLUMN имя_отношения.имя_столбца |
  CONSTRAINT имя_ограничения ON имя_таблицы |
  CONSTRAINT имя_ограничения ON DOMAIN имя_домена |
  CONVERSION имя_объекта |
  DATABASE имя_объекта |
  DOMAIN имя_объекта |
  EXTENSION имя_объекта |
  EVENT TRIGGER имя_объекта |
  FOREIGN DATA WRAPPER имя_объекта |
  FOREIGN TABLE имя_объекта |
  FUNCTION имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  INDEX имя_объекта |
  LARGE OBJECT oid_большого_объекта |
  MATERIALIZED VIEW имя_объекта |
  OPERATOR имя_оператора (тип_слева, тип_справа) |
  OPERATOR CLASS имя_объекта USING индексный_метод |
  OPERATOR FAMILY имя_объекта USING индексный_метод |
  POLICY имя_политики ON имя_таблицы |
  [ PROCEDURAL ] LANGUAGE имя_объекта |
  PROCEDURE имя_процедуры [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  PROFILE имя_объекта |
  PUBLICATION имя_объекта |
  ROLE имя_объекта |
  ROUTINE имя_подпрограммы [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  RULE имя_правила ON имя_таблицы |
  SCHEMA имя_объекта |
  SEQUENCE имя_объекта |
  SERVER имя_объекта |
  STATISTICS имя_объекта |
  SUBSCRIPTION имя_объекта |
  TABLE имя_объекта |
  TABLESPACE имя_объекта |
  TEXT SEARCH CONFIGURATION имя_объекта |
  TEXT SEARCH DICTIONARY имя_объекта |
  TEXT SEARCH PARSER имя_объекта |
  TEXT SEARCH TEMPLATE имя_объекта |
  TRANSFORM FOR имя_типа LANGUAGE имя_языка |
  TRIGGER имя_триггера ON имя_таблицы |
  TYPE имя_объекта |
  VIEW имя_объекта
} IS { строковая_константа | NULL }
```

```
Здесь сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`COMMENT` сохраняет комментарий об объекте базы данных.

Для каждого объекта сохраняется только одна строка, так что для изменения комментария нужно просто выполнить `COMMENT` ещё раз для того же объекта. Чтобы удалить комментарий, вместо текстовой строки укажите `NULL`. При удалении объектов комментарии удаляются автоматически.

Для целевого объекта запрашивается блокировка `SHARE UPDATE EXCLUSIVE`.

Для большинства типов объектов комментарий может установить только владелец объекта. Но так как роли не имеют владельцев, `COMMENT ON ROLE` для ролей суперпользователей разрешено выполнять только суперпользователям или тем, кто имеет право `CREATEROLE` и получил право `ADMIN OPTION` для целевой роли. Так же не имеют владельцев и методы доступа; чтобы добавить комментарий для метода доступа, нужно быть суперпользователем. Разумеется, суперпользователи могут задавать комментарии для любых объектов.

Просмотреть комментарии можно в psql, используя семейство команд `\d`. Имеется возможность получать комментарии и в других пользовательских интерфейсах, используя те же встроенные функции, что использует psql, а именно `obj_description`, `col_description` и `shobj_description` (см. Таблицу 9.80).

Параметры

имя_объекта
имя_отношения.имя_столбца
имя_агрегатной_функции
имя_ограничения
имя_функции
имя_оператора
имя_политики
имя_процедуры
имя_подпрограммы
имя_правила
имя_триггера

Имя объекта, для которого задаётся комментарий. Имена объектов, располагающихся в схемах, например, имена таблиц, функций и т. п., можно дополнить именем схемы. Когда определяется комментарий для столбца, *имя_отношения* должно ссылаться на таблицу, представление, составной тип или стороннюю таблицу.

имя_таблицы
имя_домена

При создании комментария для ограничения, триггера, правила или политики эти параметры задают имя таблицы или домена, к которым относится этот объект.

исходный_тип

Имя исходного типа данных для приведения.

целевой_тип

Имя целевого типа данных для приведения.

режим_аргумента

Режим аргумента функции, процедуры или агрегата: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `COMMENT` не учитывает аргументы `OUT`, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы `IN`, `INOUT` и `VARIADIC`.

имя_аргумента

Имя аргумента функции, процедуры или агрегата. Заметьте, что на самом деле `COMMENT` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргумента функции, процедуры или агрегата.

oid_большого_объекта

OID большого объекта.

тип_слева
тип_справа

Тип данных аргументов оператора (возможно, дополненный именем схемы). В случае отсутствия аргумента префиксного оператора укажите вместо типа `NONE`.

`PROCEDURAL`

Это слово не несёт смысловой нагрузки.

имя_типа

Имя типа данных, для которого предназначена трансформация.

имя_языка

Имя языка, для которого предназначена трансформация.

строковая_константа

Содержание нового комментария, записанное в виде строковой константы.

`NULL`

Укажите `NULL` для удаления комментария.

Примечания

В настоящее время механизм безопасности в части просмотра комментариев отсутствует: любой пользователь, подключённый к базе данных, может видеть все комментарии всех объектов базы. Для общих объектов, таких как базы данных, роли и табличные пространства, комментарии хранятся глобально, так что их может видеть любой пользователь, подключённый к любой базе данных в кластере. Поэтому ничего секретного писать в комментариях не следует.

Примеры

Добавление комментария для таблицы `mytable`:

```
COMMENT ON TABLE mytable IS 'Это моя таблица.';
```

Удаление его:

```
COMMENT ON TABLE mytable IS NULL;
```

Ещё несколько примеров:

```
COMMENT ON ACCESS METHOD gin IS 'Метод доступа для индекса GIN';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Вычисляет дисперсию выборки';
COMMENT ON CAST (text AS int4) IS 'Выполняет приведение строк к int4';
COMMENT ON COLLATION "fr_CA" IS 'Канадский французский';
COMMENT ON COLUMN my_table.my_column IS 'Порядковый номер сотрудника';
COMMENT ON CONVERSION my_conv IS 'Перекодировка в UTF-8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Ограничение столбца col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Ограничение для домена';
COMMENT ON DATABASE my_database IS 'База данных разработчиков';
COMMENT ON DOMAIN my_domain IS 'Домен почтового адреса';
COMMENT ON EVENT TRIGGER abort_ddl IS 'Прерывает все команды DDL';
COMMENT ON EXTENSION hstore IS 'Реализует тип данных hstore';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'Моя обёртка сторонних данных';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Информация о сотрудниках в другой БД';
COMMENT ON FUNCTION my_function (timestamp) IS 'Возвращает число римскими цифрами';
COMMENT ON INDEX my_index IS 'Обеспечивает уникальность по коду сотрудника';
COMMENT ON LANGUAGE plpython IS 'Поддержка Python для хранимых процедур';
COMMENT ON LARGE OBJECT 346344 IS 'Документ планирования';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Сводка истории заказов';
COMMENT ON OPERATOR ^ (text, text) IS 'Вычисляет пересечение двух текстов';
COMMENT ON OPERATOR - (NONE, integer) IS 'Унарный минус';
COMMENT ON OPERATOR CLASS int4ops USING btree IS 'Операторы для четырёхбайтовых целых
 (для B-деревьев)';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'Все целочисленные операторы (для
 B-деревьев)';
COMMENT ON POLICY my_policy ON mytable IS 'Фильтр строк по пользователям';
COMMENT ON PROCEDURE my_proc (integer, integer) IS 'Строит отчёт';
COMMENT ON PUBLICATION alltables IS 'Публикует все операции во всех таблицах';
COMMENT ON ROLE my_role IS 'Административная группа для таблиц бухгалтерии';
COMMENT ON ROUTINE my_routine (integer, integer) IS 'Выполняет подпрограмму (функцию
 или процедуру)';
COMMENT ON RULE my_rule ON my_table IS 'Протоколирует изменения в записях сотрудников';
COMMENT ON SCHEMA my_schema IS 'Данные отдела';
COMMENT ON SEQUENCE my_sequence IS 'Предназначена для генерации первичных ключей';
COMMENT ON SERVER myserver IS 'Мой сторонний сервер';
COMMENT ON STATISTICS my_statistics IS 'Улучшает оценку числа строк для планировщика';
COMMENT ON SUBSCRIPTION alltables IS 'Подписка на все операции во всех таблицах';
COMMENT ON TABLE my_schema.my_table IS 'Данные сотрудников';
COMMENT ON TABLESPACE my_tablespace IS 'Табличное пространство для индексов';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Фильтрация специальных слов';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Стеммер Snowball для шведского языка';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Разделяет текст на слова';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Стеммер Snowball';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpython3u IS 'Трансформирует данные из hstore
 в словарь Python';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Обеспечивает ссылочную целостность';
COMMENT ON TYPE complex IS 'Тип данных комплексных чисел';
COMMENT ON VIEW my_view IS 'Представление расходов по отделам';
```

Совместимость

Оператор `COMMENT` отсутствует в стандарте SQL.

COMMIT

COMMIT — зафиксировать текущую транзакцию

Синтаксис

```
COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Описание

`COMMIT` фиксирует текущую транзакцию. Все изменения, произведённые транзакцией, становятся видимыми для других и гарантированно сохранятся в случае сбоя.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

`AND CHAIN`

Если добавляется указание `AND CHAIN`, сразу после окончания текущей транзакции начинается новая с такими же характеристиками транзакции (см. SET TRANSACTION). В противном случае новая транзакция не начинается.

Примечания

Для прерывания транзакции используйте ROLLBACK.

При попытке выполнить `COMMIT` вне транзакции ничего не произойдёт, но будет выдано предупреждение. Однако `COMMIT AND CHAIN` вне транзакции вызовет ошибку.

Примеры

Следующая команда фиксирует текущую транзакцию и сохраняет все изменения:

```
COMMIT;
```

Совместимость

Команда `COMMIT` соответствует стандарту SQL, а форма `COMMIT TRANSACTION` является расширением Postgres Pro.

См. также

BEGIN, ROLLBACK

COMMIT PREPARED

COMMIT PREPARED — зафиксировать транзакцию, которая ранее была подготовлена для двухфазной фиксации

Синтаксис

```
COMMIT PREPARED id_транзакции
```

Описание

`COMMIT PREPARED` фиксирует транзакцию, находящуюся в подготовленном состоянии.

Параметры

`id_транзакции`

Идентификатор транзакции, которая будет зафиксирована.

Примечания

Зафиксировать подготовленную транзакцию может либо пользователь, выполнявший её изначально, либо суперпользователь. При этом не обязательно работать в том же сеансе, где выполнялась транзакция.

Эту команду нельзя выполнить внутри блока транзакции. Подготовленная транзакция фиксируется немедленно.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении `pg_prepared_xacts`.

Примеры

Фиксация транзакции, имеющей идентификатор `foobar`:

```
COMMIT PREPARED 'foobar';
```

Совместимость

Оператор `COMMIT PREPARED` является расширением Postgres Pro. Он предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

См. также

PREPARE TRANSACTION, ROLLBACK PREPARED

COPY

COPY — копировать данные между файлом и таблицей

Синтаксис

```
COPY имя_таблицы [ ( имя_столбца [, ...] ) ]
    FROM { 'имя_файла' | PROGRAM 'команда' | STDIN }
    [ [ WITH ] ( параметр [, ...] ) ]
    [ WHERE условие ]

COPY { имя_таблицы [ ( имя_столбца [, ...] ) ] | ( query ) }
    TO { 'имя_файла' | PROGRAM 'команда' | STDOUT }
    [ [ WITH ] ( параметр [, ...] ) ]

Здесь допускается параметр:

    FORMAT имя_формата
    FREEZE [ boolean ]
    DELIMITER 'символ_разделитель'
    NULL 'маркер_NULL'
    DEFAULT 'строка_по_умолчанию'
    HEADER [ boolean | MATCH ]
    QUOTE 'символ_кавычек'
    ESCAPE 'символ_экранирования'
    FORCE_QUOTE { ( имя_столбца [, ...] ) | * }
    FORCE_NOT_NULL { ( имя_столбца [, ...] ) | * }
    FORCE_NULL { ( имя_столбца [, ...] ) | * }
    ON_ERROR обработка_ошибки
    ENCODING 'имя_кодировки'
    LOG_VERBOSITY уровень_детализации
```

Описание

`COPY` перемещает данные между таблицами Postgres Pro и обычными файлами в файловой системе. `COPY TO` копирует содержимое таблицы *в* файл, а `COPY FROM` — *из* файла в таблицу (добавляет данные к тем, что уже содержались в таблице). `COPY TO` может также скопировать результаты запроса `SELECT`.

Если указывается список столбцов, `COPY TO` копирует в файл только данные указанных столбцов, а `COPY FROM` вставляет каждое поле из файла в соответствующий ему по порядку столбец из указанного списка. В случае отсутствия в этом списке каких-либо столбцов таблицы при `COPY FROM` они получают значения по умолчанию.

`COPY` с именем файла указывает серверу Postgres Pro читать или записывать непосредственно этот файл. Заданный файл должен быть доступен пользователю Postgres Pro (тому пользователю, от имени которого работает сервер), и путь к файлу должен задаваться с точки зрения сервера. Когда указывается параметр `PROGRAM`, сервер выполняет заданную команду и читает данные из стандартного вывода программы, либо записывает их в стандартный ввод. Команда должна определяться с точки зрения сервера и быть доступной для исполнения пользователю Postgres Pro. Когда указывается `STDIN` или `STDOUT`, данные передаются через соединение клиента с сервером.

Каждый процесс, выполняющий операцию `COPY`, будет выдавать информацию о ходе её выполнения, отображаемую в представлении `pg_stat_progress_copy`. За подробностями обратитесь к Подразделу 26.4.3.

При возникновении ошибок команда `COPY` по умолчанию прерывается. Однако когда необходимо загрузить файл целиком, несмотря на ошибки, можно использовать предложение `ON_ERROR`, чтобы указывать, как обработать то или иное поведение.

Параметры

`имя_таблицы`

Имя существующей таблицы (возможно, дополненное схемой).

`имя_столбца`

Необязательный список столбцов, данные которых будут копироваться. Если этот список отсутствует, копируются все столбцы таблицы, за исключением генерируемых.

`query`

Команда `SELECT`, `VALUES`, `INSERT`, `UPDATE`, `DELETE` или `MERGE`, результаты которой будут скопированы. Запрос должен заключаться в скобки.

Для запросов `INSERT`, `UPDATE`, `DELETE` и `MERGE` должно задаваться предложение `RETURNING` и в целевом отношении не должно быть условного правила, правила `ALSO` или правила `INSTEAD`, разворачивающегося в несколько операторов.

`имя_файла`

Путь входного или выходного файла. Путь входного файла может быть абсолютным или относительным, но путь выходного должен быть только абсолютным. Пользователям Windows следует использовать формат `E''` и продублировать каждую обратную черту в пути файла.

`PROGRAM`

Выполняемая команда. `COPY FROM` читает стандартный вывод команды, а `COPY TO` записывает в её стандартный ввод.

Заметьте, что команда запускается через командную оболочку, так что если требуется передать какие-либо аргументы, поступающие из недоверенного источника, необходимо аккуратно избавиться от всех спецсимволов, имеющих особое значение в оболочке, либо экранировать их. По соображениям безопасности лучше ограничиться фиксированной строкой команды или как минимум не позволять пользователям вводить в неё произвольное содержимое.

`STDIN`

Указывает, что данные будут поступать из клиентского приложения.

`STDOUT`

Указывает, что данные будут выдаваться клиентскому приложению.

`boolean`

Включает или отключает заданный параметр. Для включения параметра можно написать `TRUE`, `ON` или `1`, а для отключения — `FALSE`, `OFF` или `0`. Значение *`boolean`* можно опустить, в этом случае подразумевается `TRUE`.

`FORMAT`

Выбирает формат чтения или записи данных: `text` (текстовый), `csv` (значения, разделённые запятыми, Comma Separated Values) или `binary` (двоичный). По умолчанию выбирается формат `text`.

`FREEZE`

Запросы копируют данные с уже замороженными строками, как после выполнения команды `VACUUM FREEZE`. Это позволяет увеличить производительность при начальном добавлении данных. Строки будут замораживаться, только если загружаемая таблица была создана или опустошена в текущей подтранзакции, с ней не связаны открытые курсоры и в данной транзакции нет других снимков. Выполнять `COPY FREEZE` с секционированной таблицей в настоящее время нельзя. Допускается только использование с `COPY FROM`.

Заметьте, что все другие сеансы будут немедленно видеть данные, как только они будут успешно загружены. Это нарушает принятые правила видимости MVCC, так что пользователи должны понимать, какие проблемы это может вызвать.

`DELIMITER`

Задаёт символ, разделяющий столбцы в строках файла. По умолчанию это символ табуляции в текстовом формате и запятая в формате `CSV`. Задаваемый символ должен быть однобайтовым. Для формата `binary` этот параметр не допускается.

`NULL`

Определяет строку, задающую значение NULL. По умолчанию в текстовом формате это `\N` (обратная косая черта и N), а в формате `CSV` — пустая строка без кавычек. Пустую строку можно использовать и в текстовом формате, если не требуется различать пустые строки и NULL. Для формата `binary` этот параметр не допускается.

Примечание

При выполнении `COPY FROM` любые значения, совпадающие с этой строкой, сохраняются как значение NULL, так что при переносе данных важно убедиться в том, что это та же строка, что применялась в `COPY TO`.

`DEFAULT`

Задаёт строку, представляющую значение по умолчанию. Каждый раз, когда указанная строка встречается во входном файле, вычисляется значение по умолчанию для соответствующего столбца. Этот параметр допускается только в команде `COPY FROM` и только не для формата `binary`.

`HEADER`

Указывает, что файл содержит строку заголовка с именами каждого столбца в файле. При выводе первая строка будет содержать имена столбцов из таблицы. При вводе первая строка отбрасывается, если для этого параметра установлено значение `true` (или равнозначное логическое значение). Если для этого параметра установлено значение `MATCH`, имена столбцов (и их количество) в строке заголовка должны совпадать с фактическими именами столбцов таблицы по порядку, иначе возникнет ошибка. Этот параметр не допускается при использовании формата `binary`. Параметр `MATCH` действителен только для команд `COPY FROM`.

`QUOTE`

Указывает символ кавычек, используемый для заключения данных в кавычки. По умолчанию это символ двойных кавычек. Задаваемый символ должен быть однобайтовым. Этот параметр поддерживается только для формата `CSV`.

`ESCAPE`

Задаёт символ, который будет выводиться перед символом данных, совпавшим со значением `QUOTE`. По умолчанию это тот же символ, что и `QUOTE` (то есть, при появлении в данных кавычек, они дублируются). Задаваемый символ должен быть однобайтовым. Этот параметр допускается только для режима `CSV`.

`FORCE_QUOTE`

Принудительно заключает в кавычки все значения не `NULL` в указанных столбцах. Выводимое значение `NULL` никогда не заключается в кавычки. Если указано `*`, в кавычки будут заключаться значения не `NULL` во всех столбцах. Этот параметр принимает только команда `COPY TO` и только для формата `CSV`.

`FORCE_NOT_NULL`

Не сопоставлять значения в указанных столбцах с маркером NULL. По умолчанию, когда маркер пуст, это означает, что пустые значения будут считаны как строки нулевой длины, а не NULL, даже когда они не заключены в кавычки. При указании `*` параметр применяется ко всем столбцам. Этот параметр допускается только в команде `COPY FROM` и только для формата `CSV`.

`FORCE_NULL`

Сопоставлять значения в указанных столбцах с маркером NULL, даже если они заключены в кавычки, и в случае совпадения устанавливать значение `NULL`. По умолчанию, когда этот маркер пуст, пустая строка в кавычках будет преобразовываться в NULL. При указании `*` параметр применяется ко всем столбцам. Этот параметр допускается только в команде `COPY FROM` и только для формата `CSV`.

`ON_ERROR`

Указывает, как обрабатывать ошибку при преобразовании входящих значений в тип данных столбца. При значении `stop` параметра *`обработка_ошибки`* команда завершается с ошибкой, в то время как при значении `ignore` входящая строка пропускается и происходит переход к следующей. Значение по умолчанию: `stop`.

Значение `ignore` можно применить только для команды `COPY FROM`, если у `FORMAT` стоит значение `text` или `csv`.

Если была пропущена хотя бы одна строчка, в конце команды `COPY FROM` выводится сообщение `NOTICE` с числом пропущенных строк. При значении `verbose` параметра `LOG_VERBOSITY` сообщение `NOTICE` выводится для каждой пропущенной строки и содержит строку входящего файла и имя столбца, где преобразование завершилось с ошибкой.

`ENCODING`

Указывает, что файл имеет кодировку *`имя_кодировки`*. Если этот параметр опущен, выбирается текущая кодировка клиента. Подробнее об этом говорится ниже, в примечаниях.

`LOG_VERBOSITY`

Указывает количество сообщений, выводимых командой `COPY`, возможные значения: `default` или `verbose`. При значении `verbose` во время работы команды выводятся дополнительные сообщения.

На данный момент это используется для команды `COPY FROM` при значении `ignore` параметра `ON_ERROR`.

`WHERE`

Необязательное предложение `WHERE` имеет общую форму

```
WHERE условие
```

, где *`условие`* — любое выражение, выдающее результат типа `boolean`. Строки, не удовлетворяющие этому условию, добавляться в таблицу не будут. Строка удовлетворяет условию, если оно возвращает true при подстановке вместо ссылок на переменные фактических значений из этой строки.

В настоящее время выражения `WHERE` не могут включать подзапросы, а при вычислении выражений не видны изменения, которые вносит сама команда `COPY` (это играет роль, когда в них вызываются функции с характеристикой `VOLATILE`).

Выводимая информация

В случае успешного завершения, `COPY` возвращает метку команды в виде

```
COPY число
```

Здесь *число* — количество скопированных записей.

> **Примечание**
>
> psql выводит эту метку, только если выполнялась не команда `COPY ... TO STDOUT` или её аналог в psql, метакоманда `\copy ... to stdout`. Это сделано для того, чтобы метка команды не смешалась с данными, выведенными перед ней.

Примечания

Команду `COPY TO` можно использовать только с простыми таблицами, не представлениями, и при этом она не копирует строки из дочерних таблиц или секций. То есть, `COPY` *`таблица`* `TO` копирует те же строки, что выдаёт запрос `SELECT * FROM ONLY` *`таблица`*. Для выгрузки всех строк представления или таблицы с учётом иерархии наследования или секционирования можно применить `COPY (SELECT * FROM` *`таблица`*`) TO ...`.

`COPY FROM` можно применять с обычными, сторонними и секционированными таблицами или представлениями, в которых установлены триггеры `INSTEAD OF INSERT`.

В таблице, данные которой читает команда `COPY TO`, требуется иметь право на выборку данных, а в таблице, куда вставляет значения `COPY FROM`, требуется право на добавление. При этом, если в команде перечисляются избранные столбцы, достаточно иметь права только для них.

Если для таблицы включена защита на уровне строк, соответствующие политики `SELECT` будут применяться и к операторам `COPY` *`таблица`* `TO`. Операторы `COPY FROM` для таблиц с защитой строк в настоящее время не поддерживаются. Вместо них следует использовать равнозначные операторы `INSERT`.

Файлы, указанные в команде `COPY`, читаются или записываются непосредственно сервером, не клиентским приложением. Поэтому они должны располагаться на сервере или быть доступными серверу, а не клиенту. Они должны быть доступны на чтение или запись пользователю Postgres Pro (пользователю, от имени которого работает сервер), не клиенту. Аналогично, команда, указанная параметром `PROGRAM`, выполняется непосредственно сервером, а не клиентским приложением, и должна быть доступна на выполнение пользователю Postgres Pro. Выполнять `COPY` с указанием файла или внешней команды разрешено только суперпользователям базы данных или членам ролей `pg_read_server_files`, `pg_write_server_files` или `pg_execute_server_program`, так как это позволяет читать/записывать любые файлы и запускать любые программы, к которым имеет доступ сервер.

Не путайте команду `COPY` с реализованной в psql метакомандой `\copy`. Метакоманда `\copy` вызывает `COPY FROM STDIN` или `COPY TO STDOUT`, а затем работает с данными в файле, доступном клиенту psql. Таким образом, когда применяется команда `\copy`, доступность файла и права доступа зависят от клиента, а не от сервера.

Путь файла, указываемый в `COPY`, рекомендуется всегда задавать как абсолютный, а не относительный. Это обязательное условие для команды `COPY TO`, но `COPY FROM` позволяет прочитать файл, заданный и относительным путём. Такой путь будет интерпретироваться относительно рабочего каталога серверного процесса (обычно это каталог данных кластера), а не рабочего каталога клиента.

Выполнение команды в `PROGRAM` может быть ограничено и другими работающими в ОС механизмами контроля доступа, например SELinux.

`COPY FROM` вызывает все триггеры и обрабатывает все ограничения-проверки в целевой таблице. Однако правила при загрузке данных не вызываются.

Для столбцов идентификации команда `COPY FROM` всегда переносит значения, содержащиеся во входных данных, как команда `INSERT` с указанием `OVERRIDING SYSTEM VALUE`.

При вводе и выводе данных `COPY` учитывается `DateStyle`. Для обеспечения переносимости на другие инсталляции Postgres Pro, в которых могут использоваться нестандартные значения `DateStyle`, значение `DateStyle` следует установить равным `ISO` до вызова `COPY TO`. Также рекомендуется не выгружать данные с `IntervalStyle` равным `sql_standard`, так как сервер с другим значением `IntervalStyle` может неправильно воспринимать отрицательные интервалы в таких данных.

Входные данные интерпретируются согласно кодировке, заданной параметром `ENCODING`, или текущей кодировке клиента, а выходные кодируются в кодировке `ENCODING` или текущей кодировке клиента, даже если данные не проходят через клиента, а считываются или записываются в файл непосредственно сервером.

Во время работы команда `COPY FROM` вставляет входящие строки в таблицу. В случае сбоя эти строки остаются в удалённом состоянии: они не будут видимыми и доступными, но будут занимать место на диске. Если сбой происходит при копировании большого объёма данных, это может приводить к значительным потерям дискового пространства. При желании вернуть потерянный объём, это можно сделать с помощью команды `VACUUM`.

`FORCE_NULL` и `FORCE_NOT_NULL` можно применить одновременно к одному столбцу. В результате NULL-значения в кавычках будут преобразованы в NULL, а NULL-значения без кавычек — в пустые строки.

Postgres Pro не принимает нулевые байты в данных. Если вы хотите импортировать такие данные с помощью команды `COPY FROM`, вы можете определить в параметре конфигурации nul_byte_replacement_on_import ASCII-символ для замены нулевых байтов на лету.

Форматы файлов

Текстовый формат

Когда применяется формат text, читаемые или записываемые данные представляют собой текстовый файл, строка в котором соответствует строке таблицы. Столбцы в строке разделяются символом-разделителем. Значения самих столбцов — текстовые строки, выдаваемые функцией вывода, либо воспринимаемые функцией ввода, соответствующей типу данных столбца. Заданный маркер NULL выводится и считывается вместо столбцов со значением NULL. `COPY FROM` выдаёт ошибку, если в любой из строк во входном файле оказывается больше или меньше столбцов, чем ожидается.

Конец данных может обозначаться одной строкой, содержащей только обратную косую и точку (`\.`). Маркер конца данных не требуется при чтении из файла, так как его роль вполне выполняет конец файла; он необходим только при передаче данных в/из клиентского приложения по протоколу обмена до версии 3.0.

Символы обратной косой черты (`\`) в данных `COPY` позволяют экранировать символы данных, которые без них считались бы разделителями строк или столбцов. В частности, предваряться обратной косой *должны* следующие символы, когда они оказываются в значении столбца: сама обратная косая черта, перевод строки, возврат каретки и текущий разделитель.

Маркер NULL передаётся команде `COPY TO` как есть, без добавления обратной косой; `COPY FROM`, со своей стороны, ищет во вводимых данных маркеры NULL до удаления обратных косых. Таким образом, маркер NULL, например такой как `\N`, отличается от значения `\N` в данных (оно должно представляться в виде `\\N`).

Команда `COPY FROM` распознаёт следующие спецпоследовательности:

Последовательность	**Представляет**
`\b`	Забой (ASCII 8)
`\f`	Подача формы (ASCII 12)

Последовательность	**Представляет**
`\n`	Новая строка (ASCII 10)
`\r`	Возврат каретки (ASCII 13)
`\t`	Табуляция (ASCII 9)
`\v`	Вертикальная табуляция (ASCII 11)
`\`*цифры*	Обратная косая с последующими 1–3 восьмеричными цифрами представляет байт с заданным числовым кодом
`\x`*цифры*	Обратная косая с последующим `x` и 1-2 шестнадцатеричными цифрами представляет байт с заданным числовым кодом

В настоящее время `COPY TO` никогда не выводит спецпоследовательности с восьмеричными или шестнадцатеричными кодами, однако выводит другие вышеперечисленные спецпоследовательности вместо управляющих символов.

Любой другой символ после обратной косой, отсутствующий в приведённой выше таблице, будет представлять себя. Однако опасайтесь излишнего добавления обратных косых, так как это может привести к случайному образованию строки, обозначающей маркер конца данных (`\.`) или маркер NULL (`\N` по умолчанию). Эти строки будут восприняты прежде, чем обработаются спецпоследовательности с обратной косой.

В приложениях, генерирующих данные для `COPY`, настоятельно рекомендуется преобразовать символы новой строки и возврата каретки в последовательности `\n` и `\r`, соответственно. В настоящее время можно представить возврат каретки в данных как обратная косая и возврат каретки, а перевод строки как обратная косая и перевод строки, однако это может не поддерживаться в будущих версиях. Такие символы также подвержены искажениям, если файл с выводом `COPY` переносится между разными системами (например, с Unix в Windows и наоборот).

Все последовательности с обратной косой чертой обрабатываются после преобразования кодировки. Байты, заданные в таких последовательностях восьмеричными или шестнадцатеричными цифрами, должны представлять допустимые символы в кодировке базы данных.

`COPY TO` завершает каждую строку символом новой строки в стиле Unix («`\n`»). Серверы, работающие в Microsoft Windows, вместо этого выводят символы возврат каретки/новая строка («`\r\n`»), но только при выводе `COPY` в файл на сервере; для согласованности на разных платформах, `COPY TO STDOUT` всегда передаёт «`\n`», вне зависимости от платформы сервера. `COPY FROM` может воспринимать строки, завершающиеся символами новая строка, перевод каретки, либо возврат каретки+новая строка. Чтобы уменьшить риск ошибки из-за неэкранированных символов новой строки и возврата каретки, которые должны были быть данными, `COPY FROM` сигнализирует о проблеме, если концы строк во входных данных различаются.

Формат CSV

Этот формат применяется для импорта и экспорта данных в виде списка значений, разделённых запятыми (`CSV`), с которым могут работать многие другие программы, например электронные таблицы. Вместо правил экранирования значений, введённых в Postgres Pro для текстового формата, этот формат использует стандартный механизм экранирования `CSV`.

Значения в каждой записи разделяются символами `DELIMITER`. Если значение содержит символ разделителя, символ `QUOTE`, маркер `NULL`, символ возврата каретки или перевода строки, то всё значение дополнятся спереди и сзади символами `QUOTE`, а любое вхождение символа `QUOTE` или спецсимвола (`ESCAPE`) в данных предваряется спецсимволом. С указанием `FORCE_QUOTE` в кавычки будут принудительно заключаться любые значения не `NULL` в указанных столбцах.

В формате `CSV` отсутствует стандартный способ отличить значение `NULL` от пустой строки. В Postgres Pro команда `COPY` решает это с помощью кавычек. Значение `NULL` выводится в виде строки, задаваемой параметром `NULL`, и не заключается в кавычки, тогда как значение не `NULL`, со строкой, задаваемой параметром `NULL`, заключается. Например, с параметрами по умолчанию `NULL` записывается в виде пустой строки без кавычек, тогда как пустая строка записывается в двойных кавычках (`""`). При чтении значений действуют похожие правила. Указание `FORCE_NOT_NULL` позволяет избежать сравнений на `NULL` во входных данных в заданных столбцах, а `FORCE_NULL` — преобразовывать в `NULL` маркеры NULL, даже заключённые в кавычки.

Так как обратная косая черта не является спецсимволом в формате `CSV`, маркер конца данных `\.` может быть и значением данных. Во избежание ошибок интерпретации данные `\.`, выводимые в виде единственного элемента строки, автоматически заключаются в кавычки при выводе, а при вводе этот маркер, заключённый в кавычки, не воспринимается как маркер конца данных. При загрузке файла, созданного другой программой, в котором в единственном столбце без кавычек оказалось значение `\.`, потребуется дополнительно заключить это значение в кавычки.

Примечание

В формате `CSV` все символы являются значимыми. Заключённое в кавычки значение, дополненное пробелами или любыми другими символами, кроме `DELIMITER`, будет включать и эти символы. Это может приводить к ошибкам при импорте данных из системы, дополняющей строки `CSV` пробельными символами до некоторой фиксированной ширины. В случае возникновения такой проблемы необходимо обработать файл `CSV` и удалить из него замыкающие пробельные символы, прежде чем загружать данные из него в Postgres Pro.

Примечание

Обработчик формата `CSV` воспринимает и генерирует файлы `CSV` со значениями в кавычках, которые могут содержать символы возврата каретки и перевода строки. Таким образом, число строк в этих файлах не строго равно числу строк в таблице, как в файлах текстового формата.

Примечание

Многие программы генерируют странные и иногда неприемлемые файлы `CSV`, так что этот формат используется скорее по соглашению, чем по стандарту. Поэтому вам могут встретиться файлы, которые невозможно импортировать, используя этот механизм, а `COPY` может сформировать такие файлы, что их не смогут обработать другие программы.

Двоичный формат

При выборе формата `binary` все данные сохраняются/считываются в двоичном, а не текстовом виде. Иногда этот формат обрабатывается быстрее, чем текстовый и `CSV`, но он может оказаться непереносимым между разными машинными архитектурами и версиями Postgres Pro. Кроме того, двоичный формат сильно зависит от типов данных; например, он не позволяет вывести данные из столбца `smallint`, а затем прочитать их в столбец `integer`, хотя с текстовым форматом это вполне возможно.

Формат `binary` включает заголовок файла, ноль или более записей, содержащих данные строк, и окончание файла. Для заголовков и данных принят сетевой порядок байт.

Примечание

В PostgreSQL до версии 7.4 использовался другой двоичный формат.

Заголовок файла

Заголовок файла содержит 15 байт фиксированных полей, за которыми следует область расширения заголовка переменной длины. Фиксированные поля:

Сигнатура

Последовательность из 11 байт `PGCOPY\n\377\r\n\0` — заметьте, что нулевой байт является обязательной частью сигнатуры. (Эта сигнатура позволяет легко выявить файлы, испорченные при передаче, не сохраняющей все 8 бит данных. Она изменится при прохождении через фильтры, меняющие концы строк, отбрасывающие нулевые байты или старшие биты, либо добавляющие чётность.)

Поле флагов

Маска из 32 бит, обозначающая важные аспекты формата файла. Биты нумеруются от 0 (LSB) до 31 (MSB). Учтите, что это поле хранится в сетевом порядке байт (наиболее значащий байт первый), как и все целочисленные поля в этом формате. Биты 16–31 зарезервированы для обозначения критичных особенностей формата; обработчик должен прервать чтение, встретив любой неожиданный бит в этом диапазоне. Биты 0–15 зарезервированы для обозначения особенностей, связанных с обратной совместимостью; обработчик может просто игнорировать любые неожиданные биты в этом диапазоне. В настоящее время определён только один битовый флаг, остальные должны быть равны 0:

Бит 16

При 1 в данные включается OID, при 0 — не включается. Системные столбцы oid в Postgres Pro больше не поддерживаются, но этот индикатор всё ещё сохраняется.

Длина области расширения заголовка

Целое 32-битное число, определяющее длину в байтах остального заголовка, не включая само это значение. В настоящее время содержит 0, и сразу за ним следует первая запись. При будущих изменениях формата в заголовок могут быть добавлены дополнительные данные. Обработчик должен просто пропускать все расширенные данные заголовка, о которых ему ничего не известно.

Область расширения заголовка предусмотрена для размещения последовательности самоопределяемых блоков. Поле флагов не должно содержать указаний о том, что содержится в области расширения. Точное содержимое области расширения может быть определено в будущих версиях.

При таком подходе возможно как обратно-совместимое дополнение заголовка (добавить блоки расширения заголовка или установить младшие биты флагов), так и не обратно-совместимое (установить старшие биты флагов, сигнализирующие о подобном изменении, и добавить вспомогательные данные в область расширения, если это потребуется).

Записи

Каждая запись начинается с 16-битного целого числа, определяющего количество полей в записи. (В настоящее время во всех записях должно быть одинаковое число полей, но так может быть не всегда.) Затем, для каждого поля в записи указывается 32-битная длина поля, за которой следует это количество байт с данными поля. (Значение длины не включает свой размер, и может быть равно нулю.) В качестве особого варианта, -1 обозначает, что в поле содержится NULL. В случае с NULL за длиной не следуют байты данных.

Выравнивание или какие-либо дополнительные данные между полями не вставляются.

В настоящее время предполагается, что все значения данных в файле двоичного формата содержатся в двоичном формате (формате под кодом 1). Возможно, в будущем расширении в заголовок будет добавлено поле, позволяющее задавать другие коды форматов для разных столбцов.

Если в файл включается OID, поле OID следует немедленно за числом, определяющим количество полей. Это поле не отличается от других ничем, кроме того, что оно не учитывается в количестве полей. Заметьте, что в текущих версиях Postgres Pro системные столбцы oid не поддерживаются.

Окончание файла

Окончание файла состоит из 16-битного целого, содержащего -1. Это позволяет легко отличить его от счётчика полей в записи.

Обработчик, читающий файл, должен выдать ошибку, если число полей в записи не равно -1 или ожидаемому числу столбцов. Это обеспечивает дополнительную проверку синхронизации данных.

Примеры

В следующем примере таблица передаётся клиенту с разделителем полей «вертикальная черта» (`|`):

```
COPY country TO STDOUT (DELIMITER '|');
```

Копирование данных из файла в таблицу `country`:

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

Копирование в файл только данных стран, название которых начинается с 'A':

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/
a_list_countries.copy';
```

Для копирования данных в сжатый файл можно направить вывод через внешнюю программу сжатия:

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

Пример данных, подходящих для копирования в таблицу из `STDIN`:

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

Примечание: пробелы в каждой строке на самом деле обозначают символы табуляции.

Ниже приведены те же данные, но выведенные в двоичном формате. Данные показаны после обработки Unix-утилитой `od -c`. Таблица содержит три столбца; первый имеет тип `char(2)`, второй — `text`, а третий — `integer`. Последний столбец во всех строках содержит NULL.

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

Совместимость

Оператор `COPY` отсутствует в стандарте SQL.

До версии PostgreSQL 9.0 использовался и по-прежнему поддерживается следующий синтаксис:

```
COPY имя_таблицы [ ( имя_столбца [, ...] ) ]
    FROM { 'имя_файла' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ DELIMITER [ AS ] 'символ_разделитель' ]
```

```
          [ NULL [ AS ] 'маркер_NULL' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'символ_кавычек' ]
                [ ESCAPE [ AS ] 'символ_экранирования' ]
                [ FORCE NOT NULL имя_столбца [, ...] ] ] ]

COPY { имя_таблицы [ ( имя_столбца [, ...] ) ] | ( запрос ) }
    TO { 'имя_файла' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ DELIMITER [ AS ] 'символ_разделитель' ]
          [ NULL [ AS ] 'маркер_NULL' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'символ_кавычек' ]
                [ ESCAPE [ AS ] 'символ_экранирования' ]
                [ FORCE QUOTE { имя_столбца [, ...] | * } ] ] ]
```

Заметьте, что в этом синтаксисе ключевые слова `BINARY` и `CSV` обрабатываются как независимые, а не как аргументы параметра `FORMAT`.

До версии PostgreSQL 7.3 использовался и по-прежнему поддерживается следующий синтаксис:

```
COPY [ BINARY ] имя_таблицы
    FROM { 'имя_файла' | STDIN }
    [ [USING] DELIMITERS 'символ_разделитель' ]
    [ WITH NULL AS 'маркер_NULL' ]

COPY [ BINARY ] имя_таблицы
    TO { 'имя_файла' | STDOUT }
    [ [USING] DELIMITERS 'символ_разделитель' ]
    [ WITH NULL AS 'маркер_NULL' ]
```

См. также

Подраздел 26.4.3

CREATE ACCESS METHOD

CREATE ACCESS METHOD — создать новый метод доступа

Синтаксис

```
CREATE ACCESS METHOD имя
    TYPE тип_метода_доступа
    HANDLER функция_обработчик
```

Описание

Команда `CREATE ACCESS METHOD` создаёт новый метод доступа.

Имя метода доступа должно быть уникальным в базе данных.

Определять новые методы доступа могут только суперпользователи.

Параметры

`имя`

Имя создаваемого метода доступа.

`тип_метода_доступа`

Это предложение задаёт тип создаваемого метода доступа. В настоящее время поддерживается только `TABLE` и `INDEX`.

`функция_обработчик`

В аргументе *`функция_обработчик`* указывается имя (возможно, дополненное схемой) ранее зарегистрированной функции, представляющей метод доступа. Функция-обработчик должна принимать один аргумент типа `internal`, а тип её результата зависит от типа метода доступа; для методов доступа типа `TABLE` это должен быть `table_am_handler`, а для `INDEX` — `index_am_handler`. Также от типа метода доступа зависит API уровня C, который должна реализовывать эта функция-обработчик. API табличных методов доступа описан в Главе 59, а индексных — в Главе 60.

Примеры

Создание метода доступа индекса `heptree` с функцией-обработчиком `heptree_handler`:

```
CREATE ACCESS METHOD heptree TYPE INDEX HANDLER heptree_handler;
```

Совместимость

`CREATE ACCESS METHOD` является расширением Postgres Pro.

См. также

DROP ACCESS METHOD, CREATE OPERATOR CLASS, CREATE OPERATOR FAMILY

CREATE AGGREGATE

CREATE AGGREGATE — создать агрегатную функцию

Синтаксис

```
CREATE [ OR REPLACE ] AGGREGATE имя ( [ режим_аргумента ] [ имя_аргумента
 ] тип_данных_аргумента [ , ... ] ) (
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = комбинирующая_функция ]
    [ , SERIALFUNC = функция_сериализации ]
    [ , DESERIALFUNC = функция_десериализации ]
    [ , INITCOND = начальное_условие ]
    [ , MSFUNC = функция_состояния_движ ]
    [ , MINVFUNC = обратная_функция_движ ]
    [ , MSTYPE = тип_данных_состояния_движ ]
    [ , MSSPACE = размер_данных_состояния_движ ]
    [ , MFINALFUNC = функция_завершения_движ ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = начальное_условие_движ ]
    [ , SORTOP = оператор_сортировки ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)

CREATE [ OR REPLACE ] AGGREGATE имя ( [ [ режим_аргумента ] [ имя_аргумента
 ] тип_данных_аргумента [ , ... ] ]
                        ORDER BY [ режим_аргумента ] [ имя_аргумента
 ] тип_данных_аргумента [ , ... ] ) (
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , INITCOND = начальное_условие ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
    [ , HYPOTHETICAL ]
)

или старый синтаксис

CREATE [ OR REPLACE ] AGGREGATE имя (
    BASETYPE = базовый_тип,
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = комбинирующая_функция ]
    [ , SERIALFUNC = функция_сериализации ]
```

```
    [ , DESERIALFUNC = функция_десериализации ]
    [ , INITCOND = начальное_условие ]
    [ , MSFUNC = функция_состояния_движ ]
    [ , MINVFUNC = обратная_функция_движ ]
    [ , MSTYPE = тип_данных_состояния_движ ]
    [ , MSSPACE = размер_данных_состояния_движ ]
    [ , MFINALFUNC = функция_завершения_движ ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = начальное_условие_движ ]
    [ , SORTOP = оператор_сортировки ]
)
```

Описание

`CREATE AGGREGATE` создаёт новую агрегатную функцию, а `CREATE OR REPLACE AGGREGATE` создаёт новую или заменяет определение уже существующей агрегатной функции. Некоторое количество базовых и часто используемых агрегатных функций включено в дистрибутив; они описаны в Разделе 9.21. Но если нужно адаптировать их к новым типам или создать недостающие агрегатные функции, это можно сделать с помощью команды `CREATE AGGREGATE`.

При замене существующего определения изменить типы и количество непосредственных аргументов, а также тип результата нельзя. Кроме того, новое определение должно быть того же вида (обычный, сортирующий или гипотезирующий агрегат), что и старое.

Если указывается имя схемы (например, `CREATE AGGREGATE myschema.myagg ...`), агрегатная функция создаётся в указанной схеме. В противном случае она создаётся в текущей схеме.

Агрегатная функция идентифицируется по имени и типам входных данных. Две агрегатных функции в одной схеме могут иметь одно имя, только если они работают с разными типами данных. Имя и тип(ы) входных данных агрегата не могут совпадать с именем и типами данных любой другой обычной функции в той же схеме. Это же правило действует при перегрузке имён обычных функций (см. CREATE FUNCTION).

Простую агрегатную функцию образуют одна или две обычные функции: функция перехода состояния *функция_состояния* и необязательная функция окончательного вычисления *функция_завершения*. Они используются следующим образом:

```
функция_состояния( внутреннее-состояние, следующие-значения-данных ) ---> следующее-внутреннее-состояние
функция_завершения( внутреннее-состояние ) ---> агрегатное_значение
```

Postgres Pro создаёт временную переменную типа *тип_данных_состояния* для хранения текущего внутреннего состояния агрегата. Затем для каждой поступающей строки вычисляются значения аргументов агрегата и вызывается функция перехода состояния с текущим значением состояния и полученными аргументами; эта функция вычисляет следующее внутреннее состояние. Когда таким образом будут обработаны все строки, вызывается завершающая функция, которая должна вычислить возвращаемое значение агрегата. Если функция завершения отсутствует, просто возвращается конечное значение состояния.

Агрегатная функция может определить начальное условие, то есть начальное значение для внутренней переменной состояния. Это значение задаётся и сохраняется в базе данных в виде строки типа `text`, но оно должно быть допустимым внешним представлением константы типа данных переменной состояния. По умолчанию начальным значением состояния считается NULL.

Если функция перехода состояния объявлена как «strict» (строгая), её нельзя вызывать с входными значениями NULL. В этом случае агрегатная функция выполняется следующим образом. Строки со значениями NULL игнорируются (функция перехода не вызывается и предыдущее значение состояния не меняется) и если начальное состояние равно NULL, то в первой же строке, в которой все входные значения не NULL, первый аргумент заменяет значение состояния, а функция перехода вызывается для каждой последующей строки, в которой все входные значения не NULL. Это поведение удобно для реализации таких агрегатных функций, как `max`. Заметьте, что такое поведение возможно, только если *`тип_данных_состояния`* совпадает с первым *`типом_данных_аргумента`*. Если же эти типы различаются, необходимо задать начальное условие не NULL или использовать нестрогую функцию перехода состояния.

Если функция перехода состояния не является строгой, она вызывается безусловно для каждой поступающей строки и должна сама обрабатывать вводимые значения и переменную состояния, равные NULL. Это позволяет разработчику агрегатной функции полностью управлять тем, как она воспринимает значения NULL.

Если функция завершения объявлена как «strict» (строгая), она не будет вызвана при конечном значении состояния, равном NULL; вместо этого автоматически возвращается результат NULL. (Разумеется, это вполне нормальное поведение для строгих функций.) Когда функция завершения вызывается, она в любом случае может возвратить значение NULL. Например, функция завершения для `avg` возвращает NULL, если определяет, что было обработано ноль строк.

Иногда бывает полезно объявить функцию завершения как принимающую не только состояние, но и дополнительные параметры, соответствующие входным данным агрегата. В основном это имеет смысл для полиморфных функций завершения, которым может быть недостаточно знать тип данных только переменной состояния, чтобы вывести тип результата. Эти дополнительные параметры всегда передаются как NULL (так что функция завершения не должна быть строгой, когда применяется `FINALFUNC_EXTRA`), но в остальном это обычные параметры. Функция завершения может выяснить фактические типы аргументов в текущем вызове, воспользовавшись системным вызовом `get_fn_expr_argtype`.

Агрегатная функция может дополнительно поддерживать *режим движущегося агрегата*, как описано в Подразделе 36.12.1. Для этого режима требуются параметры `MSFUNC`, `MINVFUNC` и `MSTYPE`, а также могут задаваться `MSSPACE`, `MFINALFUNC`, `MFINALFUNC_EXTRA`, `MFINALFUNC_MODIFY` и `MINITCOND`. За исключением `MINVFUNC`, эти параметры работают как соответствующие параметры простого агрегата без начальной буквы `M`; они определяют отдельную реализацию агрегата, включающую функцию обратного перехода.

Если в список параметров добавлено указание `ORDER BY`, создаётся особый типа агрегата, называемый *сортирующим агрегатом*; с указанием `HYPOTHETICAL` создаётся *гипотезирующий агрегат*. Эти агрегаты работают с группами отсортированных значений и зависят от порядка сортировки, поэтому определение порядка сортировки входных данных является неотъемлемой частью их вызова. Кроме того, они могут иметь *непосредственные* аргументы, которые вычисляются единожды для всей процедуры агрегирования, а не для каждой поступающей строки. Гипотезирующие агрегаты представляют собой подкласс сортирующих агрегатов, в которых непосредственные аргументы должны совпадать, по количеству и типам данных, с агрегируемыми аргументами. Это позволяет добавить значения этих непосредственных аргументов в набор агрегируемых строк в качестве дополнительной «гипотетической» строки.

Агрегатная функция может дополнительно поддерживать *частичное агрегирование*, как описано в Подразделе 36.12.4. Для этого требуется задать параметр `COMBINEFUNC`. Если в качестве *`типа_данных_состояния`* выбран `internal`, обычно уместно также задать `SERIALFUNC` и `DESERIALFUNC`, чтобы было возможно параллельное агрегирование. Заметьте, что для параллельного агрегирования агрегатная функция также должна быть помечена как `PARALLEL SAFE` (безопасная для распараллеливания).

Агрегаты, работающие подобно `MIN` и `MAX`, иногда можно соптимизировать, заменив сканирование всех строк таблицы обращением к индексу. Если агрегат подлежит такой оптимизации, это можно указать, определив *оператор сортировки*. Основное требование при этом: агрегат должен выдавать в результате первый элемент по порядку сортировки, задаваемому оператором; другими словами:

```
SELECT agg(col) FROM tab;
```

должно быть равнозначно:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Дополнительно предполагается, что агрегат игнорирует значения NULL и возвращает NULL, только если строк со значениями не NULL не нашлось. Обычно оператор `<` является подходящим оператором сортировки для `MIN`, а `>` — для `MAX`. Заметьте, что обращение к индексу может дать эффект, только если заданный оператор реализует стратегию «меньше» или «больше» в классе операторов индекса-B-дерева.

Чтобы создать агрегатную функцию, необходимо иметь право `USAGE` для типов аргументов, типа(ов) состояния и типа результата, а также право `EXECUTE` для опорных функций.

Параметры

имя

Имя создаваемой агрегатной функции (возможно, дополненное схемой).

режим_аргумента

Режим аргумента: `IN` или `VARIADIC`. (Агрегатные функции не поддерживают выходные аргументы (`OUT`).) По умолчанию подразумевается `IN`. Режим `VARIADIC` может быть указан только последним.

имя_аргумента

Имя аргумента. В настоящее время используется только в целях документирования. Если опущено, соответствующий аргумент будет безымянным.

тип_данных_аргумента

Тип входных данных, с которым работает эта агрегатная функция. Для создания агрегатной функции без аргументов вставьте `*` вместо списка с определениями аргументов. (Пример такой агрегатной функции: `count(*)`.)

базовый_тип

В прежнем синтаксисе `CREATE AGGREGATE` тип входных данных задавался параметром `basetype`, а не записывался после имени агрегата. Это позволяло указать только один входной параметр. Чтобы определить функцию без аргументов, используя этот синтаксис, в качестве значения `basetype` нужно указать `"ANY"` (не `*`). Создать сортирующий агрегат старый синтаксис не позволял.

функция_состояния

Имя функции перехода состояния, вызываемой для каждой входной строки. Для обычных агрегатных функций с *N* аргументами, *функция_состояния* должна принимать *N*+1 аргумент, первый должен иметь тип *тип_данных_состояния*, а остальные — типы соответствующих входных данных. Возвращать она должна значение типа *тип_данных_состояния*. Эта функция принимает текущее значение состояния и текущие значения входных данных, и возвращает следующее значение состояния.

В сортирующих (и в том числе, гипотезирующих) агрегатах функция перехода состояния получает только текущее значение состояния и агрегируемые аргументы, без непосредственных аргументов. Других отличий у неё нет.

тип_данных_состояния

Тип данных значения состояния для агрегатной функции.

размер_данных_состояния

Средний размер значения состояния агрегата (в байтах). Если этот параметр опущен или равен нулю, применяемая оценка по умолчанию определяется по *типу_данных_состояния*. Планировщик использует это значение для оценивания объёма памяти, требуемого для агрегатного запроса с группировкой.

функция_завершения

Имя функции завершения, вызываемой для вычисления результата агрегатной функции после обработки всех входных строк. Для обычного агрегата эта функция должна принимать единственный аргумент типа *тип_данных_состояния*. Возвращаемым типом агрегата будет тип, который возвращает эта функция. Если *функция_завершения* не указана, результатом агрегата будет конечное значение состояния, а типом результата — *тип_данных_состояния*.

В сортирующих (и в том числе, гипотезирующих) агрегатах функция завершения получает не только конечное значение состояния, но и значения всех непосредственных аргументов.

Если команда содержит указание `FINALFUNC_EXTRA`, то в дополнение к конечному значению состояния и всем непосредственным аргументам функция завершения получает добавочные значения NULL, соответствующие обычным (агрегируемым) аргументам агрегата. Это в основном полезно для правильного определения типа результата при создании полиморфной агрегатной функции.

`FINALFUNC_MODIFY` = { `READ_ONLY` | `SHAREABLE` | `READ_WRITE` }

Этот параметр указывает, является ли функция завершения чистой функцией, которая не изменяет свои аргументы. Это свойство функции передаёт значение `READ_ONLY`; два других значения показывают, что она может менять значение переходного состояния. Подробнее об этом говорится в разделе Замечания ниже. По умолчанию подразумевается значение `READ_ONLY`, за исключением сортирующих агрегатов (для них значение по умолчанию — `READ_WRITE`).

комбинирующая_функция

Дополнительно может быть указана *комбинирующая_функция*, чтобы агрегатная функция поддерживала частичное агрегирование. Если задаётся, *комбинирующая_функция* должна комбинировать два значения *типа_данных_состояния*, содержащих результат агрегирования по некоторому подмножеству входных значений, и вычислять новое значение *типа_данных_состояния*, представляющее результат агрегирования по обоим множествам данных. Эту функцию можно считать своего рода *функцией_состояния*, которая вместо обработки отдельной входной строки и включения её данных в текущее агрегируемое состояние включает некоторое агрегированное состояние в текущее.

Указанная *комбинирующая_функция* должна быть объявлена как принимающая два аргумента *типа_данных_состояния* и возвращающая значение *типа_данных_состояния*. Эта функция дополнительно может быть объявлена «строгой». В этом случае данная функция не будет вызываться, когда одно из входных состояний — NULL; в качестве корректного результата будет выдано другое состояние.

Для агрегатных функций, у которых *тип_данных_состояния* — `internal`, *комбинирующая_функция* не должна быть «строгой». При этом *комбинирующая_функция* должна позаботиться о том, чтобы состояния NULL обрабатывались корректно и возвращаемое состояние располагалось в контексте памяти агрегирования.

функция_сериализации

Агрегатная функция, у которой *тип_данных_состояния* — `internal`, может участвовать в параллельном агрегировании, только если для неё задана *функция_сериализации*, которая должна сериализовать агрегатное состояние в значение `bytea` для передачи другому процессу. Эта функция должна принимать один аргумент типа `internal` и возвращать тип `bytea`. Также при этом нужно задать соответствующую *функцию_десериализации*.

функция_десериализации

Десериализует ранее сериализованное агрегатное состояние обратно в *тип_данных_состояния*. Эта функция должна принимать два аргумента типов `bytea` и `internal` и выдавать результат типа `internal`. (Замечание: второй аргумент типа `internal` не используется, но требуется из соображений типобезопасности.)

начальное_условие

Начальное значение переменной состояния. Оно должно задаваться строковой константой в форме, пригодной для ввода в *тип_данных_состояния*. Если не указано, начальным значением состояния будет NULL.

функция_состояния_движ

Имя функции прямого перехода состояния, вызываемой для каждой входной строки в режиме движущегося агрегата. Это точно такая же функция, как и обычная функция перехода, но её первый аргумент и результат имеют тип *тип_данных_состояния_движ*, который может отличаться от типа *тип_данных_состояния*.

обратная_функция_движ

Имя функции обратного перехода состояния, применяемой в режиме движущегося агрегата. У этой функции те же типы аргумента и результатов, что и у *функции_состояния_движ*, но она предназначена не для добавления, а для удаления значения из текущего состояния агрегата. Функция обратного перехода должна иметь ту же характеристику строгости, что и функция прямого перехода.

тип_данных_состояния_движ

Тип данных значения состояния для агрегатной функции в режиме движущегося агрегата.

размер_данных_состояния_движ

Примерный размер значения состояния в режиме движущегося агрегата. Он имеет то же значение, что и *размер_данных_состояния*.

функция_завершения_движ

Имя функции завершения, вызываемой в режиме движущегося агрегата для вычисления результата агрегатной функции после обработки всех входных строк. Она работает так же, как *функция_завершения*, но её первый аргумент имеет тип *тип_данных_состояния_движ*, а дополнительными пустыми аргументами управляет параметр `MFINALFUNC_EXTRA`. Тип результата, который определяет *функция_завершения_движ*, или *тип_данных_состояния_движ*, должен совпадать с типом результата обычной реализации агрегата.

`MFINALFUNC_MODIFY` = { `READ_ONLY` | `SHAREABLE` | `READ_WRITE` }

Этот параметр подобен `FINALFUNC_MODIFY`, но описывает поведение функции завершения для движущегося агрегата.

начальное_условие_движ

Начальное значение переменной состояния в режиме движущегося агрегата. Оно применяется так же, как *начальное_условие*.

оператор_сортировки

Связанный оператор сортировки для реализации агрегатов, подобных `MIN` или `MAX`. Здесь указывается просто имя оператора (возможно, дополненное схемой). Предполагается, что оператор поддерживает те же типы входных данных, что и агрегат (который должен быть обычным и иметь один аргумент).

`PARALLEL` = { `SAFE` | `RESTRICTED` | `UNSAFE` }

Указания `PARALLEL SAFE`, `PARALLEL RESTRICTED` и `PARALLEL UNSAFE` имеют те же значения, что и в `CREATE FUNCTION`. Агрегатная функция не будет считаться распараллеливаемой, если она имеет характеристику `PARALLEL UNSAFE` (она подразумевается по умолчанию!) или `PARALLEL`

`RESTRICTED`. Заметьте, что планировщик не обращает внимание на допустимость распараллеливания опорных функций агрегата, а учитывает только характеристику самой агрегатной функции.

`HYPOTHETICAL`

Этот признак, допустимый только для сортирующих агрегатов, указывает, что агрегатные аргументы должны обрабатываться согласно требованиям гипотезирующих агрегатов: то есть последние несколько непосредственных аргументов должны соответствовать по типам агрегатным аргументам (`WITHIN GROUP`). Признак `HYPOTHETICAL` не влияет на поведение во время выполнения, он учитывается только при разрешении типов данных и правил сортировки аргументов.

Параметры `CREATE AGGREGATE` могут записываться в любом порядке, не обязательно так, как показано выше.

Замечания

В параметрах, определяющих имена вспомогательных функций, при необходимости можно написать имя схемы, например: `SFUNC = public.sum`. Однако типы аргументов там не указываются — типы аргументов вспомогательных функций определяются другими параметрами.

Обычно функции Postgres Pro должны быть действительно функциями, не меняющими свои входные значения. Однако функциям агрегатного перехода, *используемым в контексте агрегатной функции*, разрешено действовать хитрее и изменять аргумент с переходным состоянием на месте. Это может дать значительный выигрыш в скорости по сравнению с созданием новой копии переходного состояния при каждом вызове.

Подобным образом, хотя функция завершения агрегата обычно не должна изменять свои входные значения, иногда может быть полезно допустить изменение аргумента с переходным состоянием. Соответствующее поведение должно обозначаться параметром `FINALFUNC_MODIFY`. Его значение `READ_WRITE` показывает, что функция завершения модифицирует переходное состояние неопределённым образом. При этом значении предотвращается использование агрегата в качестве оконной функции, а также не допускается объединение переходных состояний при вызове агрегатов с одинаковыми входными данными и функциями перехода. Значение `SHAREABLE` указывает, что функцию перехода нельзя применять после функции завершения, но с конечным значением состояния могут быть выполнены несколько вызовов функции завершения. При этом значении предотвращается использование агрегата в качестве оконной функции, но переходные состояния могут объединяться. (То есть оптимизация в данном случае состоит не в многократном применении одной функции завершения, а в применении различных функций завершения к одному и тому же конечному переходному состоянию. Это допускается, только если ни одна из функций завершения не помечена как `READ_WRITE`.)

Если агрегатная функция поддерживает режим движущегося агрегата, это увеличивает эффективность вычислений, когда она применяется в качестве оконной функции для окна с движущимся началом рамки (то есть когда начало определяется не как `UNBOUNDED PRECEDING`). По сути, функция прямого перехода добавляет входные значения к состоянию агрегата, когда они поступают в рамку окна снизу, а функция обратного перехода снова вычитает их, когда они покидают рамку сверху. Поэтому вычитаются значения в том же порядке, в каком добавлялись. Когда бы ни вызывалась функция обратного перехода, она таким образом получит первое из добавленных, но ещё не удалённых значений аргумента. Функция обратного перехода может рассчитывать на то, что после того, как она удалит самые старые данные, в текущем состоянии останется ещё как минимум одна строка. (Когда это правило могло бы нарушиться, механизм оконных функций просто начинает агрегировать данные заново, а не вызывает функцию обратного перехода.)

Функция прямого перехода для режима движущегося агрегата не может возвращать NULL в качестве нового значения состояния. Если NULL возвращает функция обратного перехода, это показывает, что она не может произвести обратное вычисление для этих конкретных данных, и что вычисление агрегата следует выполнить заново от начальной позиции текущей рамки. Благодаря этому соглашению, режим движущегося агрегата можно использовать, даже если иногда возникают ситуации, в которых обратный расчёт состояния производить непрактично.

Агрегатную функцию можно использовать с движущимися рамками и без реализации движущегося агрегата, но при этом Postgres Pro будет заново агрегировать все данные при каждом перемещении начала рамки. Заметьте, что вне зависимости от того, поддерживает ли агрегатная функция режим движущегося агрегата, Postgres Pro может обойтись без повторных вычислений при сдвиге конца рамки; новые значения просто продолжают добавляться в состояние агрегата. Именно поэтому для использования агрегата в качестве оконной функции требуется, чтобы функция завершения была только читающей: она не должна изменять значение состояния агрегата, чтобы агрегирование могло продолжаться и после получения результата для набора строк в определённой рамке.

Синтаксис сортирующих агрегатных функций позволяет указать `VARIADIC` и в последнем непосредственном параметре, и в последнем агрегатном (`WITHIN GROUP`). Однако в текущей реализации на применение `VARIADIC` накладываются два ограничения. Во-первых, в сортирующих агрегатах можно использовать только `VARIADIC "any"`, но не другие типы переменных массивов. Во-вторых, если последним непосредственным аргументом является `VARIADIC "any"`, то допускается только один агрегатный аргумент и это тоже должен быть `VARIADIC "any"`. (В представлении, используемом в системных каталогах, эти два параметра объединяются в один элемент `VARIADIC "any"`, так как в `pg_proc` нельзя представить функцию с несколькими параметрами `VARIADIC`.) Если агрегатная функция является гипотезирующей, непосредственные аргументы, соответствующие параметру `VARIADIC "any"`, будут гипотетическими; любые предшествующие параметры представляют дополнительные непосредственные аргументы, которые могут не соответствовать агрегатным.

В настоящее время сортирующие агрегатные функции не поддерживают режим движущегося агрегата, так как их нельзя применять в качестве оконных функций.

Частичное (в том числе, параллельное) агрегирование в настоящее время не поддерживается для сортирующих агрегатных функций. Также оно никогда не будет применяться для агрегатных вызовов с предложениями `DISTINCT` или `ORDER BY`, так как они по природе своей не могут быть реализованы с частичным агрегированием.

Примеры

См. Раздел 36.12.

Совместимость

Оператор `CREATE AGGREGATE` является языковым расширением Postgres Pro. В стандарте SQL не предусмотрено создание пользовательских агрегатных функций.

См. также

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

CREATE CAST — создать приведение

Синтаксис

```
CREATE CAST (исходный_тип AS целевой_тип)
    WITH FUNCTION имя_функции [ (тип_аргумента [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (исходный_тип AS целевой_тип)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (исходный_тип AS целевой_тип)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

Описание

`CREATE CAST` создаёт новое приведение. Приведение определяет, как выполнить преобразование из одного типа в другой. Например,

```
SELECT CAST(42 AS float8);
```

преобразует целочисленную константу 42 к типу `float8`, вызывая ранее определённую функцию, в данном случае `float8(int4)`. (Если подходящее приведение не определено, возникнет ошибка преобразования.)

Два типа могут быть *двоично-сводимыми*; это означает, что преобразование может быть выполнено «бесплатно», без вызова какой-либо функции. Для этого требуется, чтобы соответствующие значения имели одинаковое внутреннее представление. Например, типы `text` и `varchar` двоично-сводимые в обе стороны. Отношение двоичной сводимости не обязательно симметрично. Например, приведение типа `xml` к типу `text` в текущей реализации можно выполнить бесплатно, но для преобразования в обратном направлении требуется функция, выполняющая как минимум синтаксическую проверку. (Два типа, двоично-сводимые в обе стороны, также называются двоично-совместимыми.)

Приведение можно определить как *преобразование ввода-вывода*, используя указание `WITH INOUT`. В этом случае для приведения одного типа к другому вызывается функция вывода исходного типа данных, а выданная ей строка передаётся функции ввода целевого типа. Во многих случаях эта возможность избавляет от необходимости писать для преобразования всех типов отдельные функции приведения. Преобразование ввода-вывода работает так же, как и обычное приведение с функцией; отличается только реализация.

По умолчанию, приведение можно вызвать, только записав его явно, то есть применив конструкцию `CAST(x AS имя_типа)` или `x::имя_типа`.

Если приведение помечено `AS ASSIGNMENT`, его можно вызывать неявно, присваивая значение столбцу с целевым типом данных. Например, если `foo.f1` — столбец типа `text`, то команда:

```
INSERT INTO foo (f1) VALUES (42);
```

будет допустимой, если приведение типа `integer` к `text` помечено `AS ASSIGNMENT`, и не будет в противном случае. (Для описания такого типа приведений мы обычно используем термин *приведение присваивания*.)

Если приведение помечено `AS IMPLICIT`, оно будет вызываться неявно в любом контексте, будь то присваивание или внутреннее преобразование в выражении. (Обычно мы называет приведение такого типа *неявным приведением*.) Например, рассмотрите этот запрос:

```
SELECT 2 + 4.0;
```

При разборе запроса константам сначала назначаются типы `integer` и `numeric`. Однако в системных каталогах нет оператора `integer + numeric`, хотя есть оператор `numeric + numeric`. Таким образом, запрос выполнится успешно, если существует преобразование типа `integer` к `numeric` с пометкой `AS IMPLICIT` — и на самом деле это так. Анализатор запроса применит неявное приведение и запрос будет обработан, как если бы он был записан в виде

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

Системные каталоги также содержат приведение типа `numeric` к `integer`. Если бы это приведение тоже было бы помечено `AS IMPLICIT` (на самом деле это не так), анализатору запроса пришлось бы выбирать между предыдущим вариантом и приведением константы `numeric` к типу `integer` с последующим применением оператора `integer + integer`. Не имея возможности выбрать лучший вариант, анализатор бы не смог разрешить запрос и объявил бы его неоднозначным. Именно благодаря тому, что только одно из двух приведений сделано неявным, анализатор приходит к пониманию, что предпочитаемым является преобразование выражения `numeric`-и-`integer` в `numeric`; отдельного встроенного знания об этом нет.

Определяя, объявлять ли приведения неявными, разумно проявлять консерватизм. При чрезмерном количестве способов неявного приведения Postgres Pro может выбирать неожиданные интерпретации команд, или вовсе не сможет выполнить команды из-за наличия множества возможных интерпретаций. Как правило, следует делать приведение неявно вызываемым только для преобразований, сохраняющих информацию, между типами в одной общей категории типов. Например, приведение `int2` к `int4` разумно сделать неявным, но приведение `float8` к `int4`, возможно, лучше сделать только приведением присваивания. Приведения типов разных категорий, например, `text` к `int4`, лучше делать только явными.

Примечание

Иногда ради удобства или соответствия стандартам требуется ввести множество неявных преобразований для нескольких типов, что приводит к неизбежной неоднозначности. Чтобы анализатор запроса мог обеспечить желаемое поведение в таких случаях, он дополнительно принимает во внимание *категории типов* и *предпочитаемые типы*. Подробнее это описано в CREATE TYPE.

Чтобы создать приведение, необходимо быть владельцем одного (исходного или целевого) типа и иметь право `USAGE` для другого типа. Создать двоично-сводимое приведение могут только суперпользователи. (Это ограничение введено потому, что преобразование данных с ошибочным двоичным сведением может легко вызывать сбой сервера.)

Параметры

`исходный_тип`

Имя исходного типа данных для приведения.

`целевой_тип`

Имя целевого типа данных для приведения.

`имя_функции``[(`*`тип_аргумента`*` [, ...])]`

Функция, вызываемая для выполнения приведения. Имя функции может быть дополнено схемой; в противном случае для поиска функции просматривается путь поиска. Тип данных результата должен соответствовать целевому типу приведения. Аргументы функции рассматриваются ниже. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

`WITHOUT FUNCTION`

Обозначает, что исходный тип сводится к целевому на двоичном уровне, так что функция для приведения не требуется.

`WITH INOUT`

Обозначает, что приведение выполняется как преобразование ввода-вывода, то есть вызывается функция вывода исходного типа данных, а её результат-строка передаётся функции ввода целевого типа.

`AS ASSIGNMENT`

Обозначает, что приведение может вызываться неявно в контексте присваивания.

`AS IMPLICIT`

Обозначает, что приведение может вызываться неявно в любом контексте.

Функции, реализующие приведение, могут иметь от одного до трёх аргументов. Тип первого аргумента должен быть идентичен или двоично-сводимым к исходному типу приведения. Второй аргумент, если он есть, должен иметь тип `integer`; в нём передаётся модификатор типа, связанный с целевым типом, или `-1`, если он отсутствует. Третий аргумент, если он есть, должен иметь тип `boolean`; в нём передаётся `true`, если приведение выполняется явно, либо `false` в противном случае. (Это довольно экстравагантно, но стандарт SQL предусматривает разное поведение для явного и неявного приведения в некоторых случаях. Этот аргумент предназначен для функций, которые должны реализовывать такие приведения. Однако создавать собственные типы данных, для которых это имело бы значение, не рекомендуется.)

Возвращаемый тип функции приведения должен быть идентичным или двоично-сводимым к целевому типу приведения.

Обычно исходный и целевой типы в приведении различаются, однако можно объявить приведение одного типа к такому же, если функция, реализующая преобразование, имеет более одного аргумента. Это используется для представления в системных каталогах функций, сводящих разные длины типов. Реализующая такое приведение функция будет сводить значение типа к значению с определённым модификатором, заданному вторым аргументом.

Когда исходный и целевой типы приведения различаются и функция принимает более одного аргумента, преобразование типа из одного в другой и сведение к нужной длине может выполняться за один шаг. Если же соответствующей записи не находится, приведение к типу с определённым модификатором выполняется в два этапа: сначала выполняется преобразование типа, а затем применяется модификатор типа.

Приведение типа домена или к типу домена в настоящее время не осуществляется. При попытке выполнить такое приведение вместо него выполняется приведение, связанное с базовым типом домена.

Примечания

Для удаления приведений, созданных пользователем, применяется `DROP CAST`.

Помните, что когда требуется преобразовывать типы в обе стороны, необходимо явно описать два приведения.

Обычно не требуется создавать приведения между пользовательскими типами и стандартными строковыми типами (`text`, `varchar` и `char(n)`, а также пользовательскими типами, относящимися к категории строковых). Для них Postgres Pro предоставляет автоматическое преобразование ввода-вывода. Автоматические приведения к строковым типам считаются приведениями присваивания, а автоматические приведения строковых типов к другим могут быть только явными. Это поведение можно переопределить, создав собственное приведение, заменяющее автоматическое, но обычно это нужно, только чтобы сделать вызов более удобным, чем стандартное только присваивание или явное указание. Возможен и другой повод для такого переопределения — желание создать приведение, работающее не так, как функция ввода-вывода типа; но это настолько удивительно, что следует дважды подумать, хороша ли эта идея. (На самом деле у небольшого количества встроенных типов имеются подобные специфические приведения, в основном из-за требований стандарта SQL.)

Хотя это и не обязательно, но рекомендуется следовать старому соглашению называть функции, реализующие приведение, по целевому типу данных. Многие привыкли выполнять преобразование типов данных, записывая его в стиле функций, т. е. *имя_типа*(*x*). Эта запись на самом деле ни больше ни меньше как просто вызов функции, реализующей приведение; такой вызов не воспринимается как именно приведение. Если называть функции, не следуя этому соглашению, это может оказаться неожиданным для пользователей. Так как Postgres Pro позволяет перегружать одно и то же имя функции с разными типами аргументов, ничто не мешает создать множество функций приведения разных типов к одному, названных по имени этого целевого типа.

Примечание

Вообще говоря, в предыдущем абзаце допущено некоторое упрощение: есть два случая, когда конструкция с вызовом функции исполняется как приведение, без сопоставления с фактической функцией. Если вызову функции *имя*(*x*) в точности не соответствует существующая функция, но имеется тип данных *имя* и в `pg_cast` есть двоично-сводимое приведение типа *x* к этому типу, такой вызов будет воспринят как приведение. Это исключение введено, чтобы двоично-сводимое приведение можно было вызывать, используя синтаксис функций, несмотря на то, что никакой функции преобразования у него нет. Аналогично, если запись приведения в `pg_cast` отсутствует, но в случае приведения это было бы преобразование в/из строкового типа, такой вызов будет выполнен как преобразование ввода-вывода. Это исключение позволяет вызывать преобразование ввода-вывода, используя синтаксис вызова функции.

Примечание

Но есть исключение и из этого исключения: преобразование ввода-вывода из составных типов в строковые нельзя вызвать в виде функции, его необходимо записать как явное приведение (используя `CAST` или запись `::`). Это исключение было добавлено, потому что после введения автоматически предоставляемых преобразований ввода-вывода, оказалось слишком легко случайно вызвать такое приведение, тогда как имелась в виду ссылка на столбец или функцию.

Примеры

Создание приведения присваивания типа `bigint` к типу `int4` с помощью функции `int4(bigint)`:

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

(Это приведение уже предопределено в системе.)

Совместимость

Команда `CREATE CAST` соответствует стандарту SQL, за исключением того, что в стандарте ничего не говорится о двоично-сводимых типах и дополнительных аргументах реализующих функций. Указание `AS IMPLICIT` тоже является расширением Postgres Pro.

См. также

CREATE FUNCTION, CREATE TYPE, DROP CAST

CREATE COLLATION

CREATE COLLATION — создать правило сортировки

Синтаксис

```
CREATE COLLATION [ IF NOT EXISTS ] имя (
    [ LOCALE = локаль, ]
    [ LC_COLLATE = категория_сортировки, ]
    [ LC_CTYPE = категория_типов_символов, ]
    [ PROVIDER = провайдер, ]
    [ DETERMINISTIC = boolean, ]
    [ RULES = правила, ]
    [ VERSION = версия ]
)
CREATE COLLATION [ IF NOT EXISTS ] имя FROM существующее_правило
```

Описание

`CREATE COLLATION` определяет новое правило сортировки, используя параметры локали операционной системы, либо копируя существующее правило.

Чтобы создать правило сортировки, необходимо иметь право `CREATE` в целевой схеме.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если правило сортировки с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее правило сортировки как-то соотносится с тем, которое могло бы быть создано.

`имя`

Имя правила сортировки, возможно, дополненное схемой. Если схема не указана, правило сортировки создаётся в текущей схеме. Заданное имя правила должно быть уникальным в этой схеме. (Системные каталоги могут содержать правила сортировки с одним именем, но предназначенные для разных кодировок, однако они будут игнорироваться, если их кодировка не совпадает с кодировкой базы данных.)

`локаль`

Имя локали для правила сортировки. За подробностями обратитесь к Подразделу 22.2.2.3.1 и Подразделу 22.2.2.3.2.

Если указан *`провайдер`* `libc`, это краткая запись для одновременной установки `LC_COLLATE` и `LC_CTYPE`. Если указана *`локаль`*, задать любой из этих параметров отдельно нельзя.

Если указан *`провайдер`* `builtin`, должна быть указана *`локаль`* со значением `C` или `C.UTF-8`.

`категория_сортировки`

Если указан *`провайдер`* `libc`, указанная локаль операционной системы устанавливается в качестве категории локали `LC_COLLATE`.

`категория_типов_символов`

Если указан *`провайдер`* `libc`, указанная локаль операционной системы устанавливается в качестве категории локали `LC_CTYPE`.

`провайдер`

Задаёт провайдер, который будет использоваться для функций локализации, связанных с данным правилом сортировки. Возможные значения: `builtin`, `icu`(если сервер собран с поддержкой ICU) или `libc`. По умолчанию выбирается `libc`. За подробностями обратитесь к Подразделу 22.1.4.

`DETERMINISTIC`

Определяет, будет ли правило сортировки использовать детерминированное сравнение. По умолчанию выбирается именно такое сравнение. При детерминированном сравнении строки, состоящие из различных байтов, считаются неравными, даже если на логическом уровне они одинаковы. Postgres Pro решает вопрос их равенства, сравнивая их по байтам. Если же для правила сортировки выбрать недетерминированное сравнение, это правило может стать, например, независящим от ударения или регистра символов. Для этого необходимо выбрать подходящее значение `LOCALE` *и* сделать это правило сортировки недетерминированным.

Недетерминированные правила сортировки поддерживаются только с провайдером ICU.

`правила`

Указывает дополнительные правила для настройки поведения сортировки. Параметр поддерживается только для ICU. За дополнительной информацией обратитесь к Подразделу 22.2.3.4.

`версия`

Задаёт строку версии, сохраняемую с правилом сортировки. Обычно её не следует задавать — тогда эта версия будет получена из фактической версии правила сортировки, сообщённой операционной системой. Это указание предназначено для того, чтобы команда `pg_upgrade` смогла скопировать версию из существующей инсталляции.

Что делать при несовпадении версий правил сортировки, описано в ALTER COLLATION.

`существующее_правило`

Имя копируемого существующего правила сортировки. Новое правило сортировки получит те же свойства, что и существующее, но будет независимым объектом.

Примечания

Команда `CREATE COLLATION` устанавливает блокировку `SHARE ROW EXCLUSIVE` в системном каталоге `pg_collation`. Эта блокировка конфликтует с такой же, поэтому в один момент времени может выполняться только одна команда `CREATE COLLATION`.

Для удаления созданных пользователем правил сортировки применяется команда `DROP COLLATION`.

Подробнее узнать о создании правил сортировки можно в Подразделе 22.2.2.3.

Когда используется провайдер `libc`, локаль должна быть применимой к кодировке текущей базы данных. Точные правила описаны в CREATE DATABASE.

Примеры

Создание правила сортировки из локали операционной системы `ru_RU.utf8` (предполагается, что кодировка текущей базы данных — `UTF-8`):

```
CREATE COLLATION russian (locale = 'ru_RU.utf8');
```

Создание правила сортировки с порядком, при котором латинские буквы идут перед буквами кириллицы, с использованием провайдера ICU:

```
CREATE COLLATION latn_cyrl (provider = icu, locale = 'ru-RU-u-kr-latn-cyrl');
```

Создание правила сортировки с пользовательскими условиями с использованием провайдера ICU на основе корневой локали ICU:

```
CREATE COLLATION custom (provider = icu, locale = 'und', rules = '&V << w <<< W');
```

За дополнительной информацией и примерами синтаксиса правил обратитесь к Подразделу 22.2.3.4.

Создание правила сортировки из уже существующего:

```
CREATE COLLATION german FROM "de_DE";
```

Иногда удобно использовать в приложениях имена правил сортировки, не зависящие от операционной системы.

Совместимость

Оператор `CREATE COLLATION` определён в стандарте SQL, но его действие ограничено копированием существующего правила сортировки. Синтаксис создания нового правила сортировки представляет собой расширение Postgres Pro.

См. также

ALTER COLLATION, DROP COLLATION

CREATE CONVERSION

CREATE CONVERSION — создать перекодировку

Синтаксис

```
CREATE [ DEFAULT ] CONVERSION имя
    FOR исходная_кодировка TO целевая_кодировка FROM имя_функции
```

Описание

`CREATE CONVERSION` определяет новую перекодировку между двумя наборами символов.

Перекодировки, помеченные как `DEFAULT`, могут применяться для автоматического преобразования кодировки между клиентом и сервером. Чтобы такое преобразование было возможно, должны быть определены две перекодировки: из кодировки A в B *и* из кодировки B в A.

Чтобы создать перекодировку, необходимо иметь право `EXECUTE` для реализующей функции и право `CREATE` в целевой схеме.

Параметры

`DEFAULT`

Предложение `DEFAULT` показывает, что эта перекодировка должна использоваться по умолчанию для преобразования заданной исходной кодировки в целевую. Для каждой пары кодировок может быть только одна перекодировка по умолчанию.

имя

Имя перекодировки, возможно, дополненное схемой. Если схема не указана, перекодировка создаётся в текущей схеме. Имя перекодировки должно быть уникально в этой схеме.

исходная_кодировка

Имя исходной кодировки.

целевая_кодировка

Имя целевой кодировки.

имя_функции

Функция, выполняющая перекодирование. Имя функции может быть дополнено схемой, в противном случае для поиска функции просматривается путь поиска.

Функция должна иметь следующую сигнатуру:

```
conv_proc(
    integer,  -- идентификатор исходной кодировки
    integer,  -- идентификатор целевой кодировки
    cstring,  -- исходная строка (строка, завершающаяся 0, как в C)
    internal, -- целевая строка (заполняется строкой, завершающейся 0, как в C)
    integer,  -- длина исходной строки
    boolean   -- если true, игнорировать ошибку преобразования
) RETURNS integer;
```

Возвращать она должна количество успешно преобразованных байт целевой строки. Если последний аргумент равен false, функция должна выдать ошибку при неверных входных данных и в любом случае возвратить в результате длину исходной строки.

Примечания

И исходная, и целевая кодировки должны отличаться от `SQL_ASCII`, так как поведение сервера с «кодировкой» `SQL_ASCII` предопределено.

Для удаления перекодировок, созданных пользователем, применяется `DROP CONVERSION`.

Набор прав, требуемых для создания перекодировки, может измениться в будущих версиях.

Примеры

Создание перекодировки из кодировки `UTF8` в `LATIN1` с использованием функции `myfunc`:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

Совместимость

Оператор `CREATE CONVERSION` является расширением Postgres Pro. В стандарте SQL отсутствует оператор `CREATE CONVERSION`, но есть очень похожий по предназначению и синтаксису оператор `CREATE TRANSLATION`.

См. также

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

CREATE DATABASE — создать базу данных

Синтаксис

```
CREATE DATABASE имя
    [ WITH ] [ OWNER [=] имя_пользователя ]
           [ TEMPLATE [=] шаблон ]
           [ ENCODING [=] кодировка ]
           [ STRATEGY [=] стратегия ]
           [ LOCALE [=] локаль ]
           [ LC_COLLATE [=] категория_сортировки ]
           [ LC_CTYPE [=] категория_типов_символов ]
           [ BUILTIN_LOCALE [=] локаль_builtin ]
           [ ICU_LOCALE [=] локаль_icu ]
           [ ICU_RULES [=] правила_icu ]
           [ LOCALE_PROVIDER [=] провайдер_локали ]
           [ COLLATION_VERSION = версия_правил_сортировки ]
           [ TABLESPACE [=] табл_пространство ]
           [ ALLOW_CONNECTIONS [=] разр_подключения ]
           [ CONNECTION LIMIT [=] предел_подключений ]
           [ IS_TEMPLATE [=] это_шаблон ]
           [ OID [=] oid ]
```

Описание

Команда `CREATE DATABASE` создаёт базу данных Postgres Pro.

Чтобы создать базу данных, необходимо быть суперпользователем или иметь специальное право `CREATEDB`. См. CREATE ROLE.

По умолчанию новая база данных создаётся копированием стандартной системной базы данных `template1`. Задать другой шаблон можно, добавив указание `TEMPLATE` *имя*. В частности, написав `TEMPLATE template0`, можно создать чистую базу данных (в которой никаких пользовательских объектов нет, есть только системные объекты в первозданном виде), содержащую только стандартные объекты, предопределённые установленной версией Postgres Pro. Это бывает полезно, когда копировать в новую базу любые дополнительные объекты, добавленные локально в `template1`, нежелательно.

Параметры

имя

Имя создаваемой базы данных.

имя_пользователя

Имя пользователя (роли), назначаемого владельцем новой базы данных, либо `DEFAULT`, чтобы владельцем стал пользователь по умолчанию (а именно, пользователь, выполняющий команду). Чтобы создать базу данных и сделать её владельцем другую роль, необходимо иметь право `SET ROLE` для этой роли.

шаблон

Имя шаблона, из которого будет создаваться новая база данных, либо `DEFAULT`, чтобы выбрать шаблон по умолчанию (`template1`).

кодировка

Кодировка символов в новой базе данных. Укажите строковую константу (например, `'SQL_ASCII'`) или целочисленный номер кодировки, либо `DEFAULT`, чтобы выбрать кодировку по умолчанию (а именно, кодировку шаблона). Наборы символов, которые поддерживает Postgres Pro, перечислены в Подразделе 22.3.1. Дополнительные ограничения описаны ниже.

стратегия

Стратегия, которая будет использоваться при создании новой базы данных. Если используется стратегия `WAL_LOG`, база данных будет копироваться блок за блоком и каждый блок будет отдельно записан в журнал предзаписи. Эта стратегия наиболее эффективна в случаях, когда шаблонная база данных небольшого размера, поэтому такая стратегия используется по умолчанию. Также доступна старая стратегия `FILE_COPY`. Если выбрана эта стратегия, делается небольшая запись в журнал предзаписи для каждого табличного пространства, используемого целевой базой данных. Каждая такая запись является копией всего каталога в другом месте на уровне файловой системы. Хотя такая стратегия существенно уменьшает объём журнала упреждающей записи, особенно если шаблонная база данных большого размера, она требует выполнения процедуры контрольной точки как до, так и после создания новой базы данных. В некоторых ситуациях это может оказать заметное негативное влияние на общую производительность системы.

локаль

Задаёт порядок сортировки по умолчанию и особенности классификации символов для новой базы данных. Правила сортировки определяют порядок сортировки строк, например в запросах с `ORDER BY`, а также порядок индексов по текстовым столбцам. Классификация символов определяет принадлежность символов категориям, например строчные, заглавные и цифры. Этот параметр также устанавливает соответствующие переменные окружения ОС, `LC_COLLATE` и `LC_CTYPE`. По умолчанию используется значение из шаблона. Подробное описание представлено в Подразделе 22.2.2.3.1 и Подразделе 22.2.2.3.2.

Параметр можно переопределить, задав один из следующих параметров: *категория_сортировки*, *категория_типов_символов*, *локаль_builtin* или *локаль_icu*.

При значении `builtin` у *провайдер_локали* необходимо указать *локаль* или *локаль_builtin* и задать для них значение `C` или `C.UTF-8`.

Подсказка

Другие параметры локали lc_messages, lc_monetary, lc_numeric и lc_time задаются не на уровне базы данных и этой командой не устанавливаются. Чтобы изменить их значения по умолчанию для конкретной базы, воспользуйтесь командой `ALTER DATABASE ... SET`.

категория_сортировки

Устанавливает значение `LC_COLLATE` в окружении ОС сервера баз данных. По умолчанию используется значение *локаль* (если оно указано), в противном случае — значение из шаблона. Дополнительные ограничения описаны ниже.

Если для *провайдер_локали* задано значение `libc`, этот параметр также задаёт порядок сортировки по умолчанию для новой базы данных, переопределяя значение *локаль*.

категория_типов_символов

Устанавливает значение `LC_CTYPE` в окружении ОС сервера баз данных. По умолчанию используется значение *локаль* (если оно указано), в противном случае — значение из шаблона. Дополнительные ограничения описаны ниже.

Если для *провайдер_локали* задано значение `libc`, этот параметр также задаёт классификацию символов по умолчанию для новой базы данных, переопределяя значение *локаль*.

`локаль_builtin`

Задаёт локаль провайдера `builtin` для установки порядка сортировки по умолчанию и особенностей классификации символов, переопределяя значение *`локаль`*. При этом должно быть задано значение `builtin` для провайдера локалей. По умолчанию используется значение из *`локаль`* (если оно указано), в противном случае — значение из базы-шаблона.

Локали, доступные для провайдера `builtin`: `C` и `C.UTF-8`.

`локаль_icu`

Задаёт локаль ICU (см. Подраздел 22.2.2.3.2) для установки порядка сортировки по умолчанию и особенностей классификации символов, переопределяя значение *`локаль`*. При этом должен быть задан провайдер локалей ICU. По умолчанию используется значение параметра *`локаль`*, если оно указано, в противном случае — значение параметра *`локаль`* из базы-шаблона, если оно указано. Если же и оно не указано — значение lc_collate.

`правила_icu`

Указывает дополнительные правила для настройки поведения сортировки по умолчанию. Параметр поддерживается только для ICU. За дополнительной информацией обратитесь к *https://unicode-org.github.io/icu/userguide/collation/customization/*.

`провайдер_локали`

Задаёт провайдера, который будет использоваться для основного правила сортировки в этой базе данных. Возможные значения: `builtin`, `icu`(если сервер собран с поддержкой ICU) или `libc`. По умолчанию выбирается провайдер, указанный в *`шаблон`*. За подробностями обратитесь к Подразделу 22.1.4.

`версия_правил_сортировки`

Задаёт строку версии правила сортировки, которая будет записана с объектом базы данных. Обычно её не следует задавать — тогда эта строка будет получена из фактической версии правила сортировки БД, предоставляемого операционной системой. Это указание предназначено для того, чтобы утилита `pg_upgrade` могла скопировать версию из существующей инсталляции.

Что делать в случае несовпадения версий правил сортировки, описано в ALTER DATABASE.

`табл_пространство`

Имя табличного пространства, связываемого с новой базой данных, или `DEFAULT` для использования табличного пространства шаблона. Это табличное пространство будет использоваться по умолчанию для объектов, создаваемых в этой базе. За подробностями обратитесь к CREATE TABLESPACE.

`разр_подключения`

Если false, никто не сможет подключаться к этой базе данных. По умолчанию имеет значение true, то есть подключения принимаются (если не ограничиваются другими механизмами, например, `GRANT`/`REVOKE CONNECT`).

`предел_подключений`

Максимальное количество одновременных подключений к этой базе данных. Значение -1 (по умолчанию) снимает ограничение.

`это_шаблон`

Если true, базу данных сможет клонировать любой пользователь с правами `CREATEDB`; в противном случае (по умолчанию), клонировать эту базу смогут только суперпользователи и её владелец.

`oid`

Идентификатор объекта, который будет назначен новой базе данных. Если этот параметр не указан, Postgres Pro автоматически выберет подходящий OID. Этот параметр в первую очередь предназначен для внутреннего использования в pg_upgrade, и только pg_upgrade может указывать значение меньше 16384.

Дополнительные параметры могут записываться в любом порядке, не обязательно так, как показано выше.

Примечания

`CREATE DATABASE` нельзя выполнять внутри блока транзакции.

Ошибки, содержащие сообщение «не удалось инициализировать каталог базы данных», чаще всего связаны с нехваткой прав в каталоге данных, заполнением диска или другими проблемами в файловой системе.

Для удаления базы данных применяется `DROP DATABASE`.

Программа createdb представляет собой оболочку этой команды, созданную ради удобства.

Конфигурационные параметры уровня базы данных (устанавливаемые командой `ALTER DATABASE`) и разрешения уровня базы (устанавливаемые командой `GRANT`) из шаблона не копируются.

Хотя с помощью этой команды можно скопировать любую базу данных, а не только `template1`, указав её имя в качестве имени шаблона, она не предназначена (пока) для использования в качестве универсального средства вроде «`COPY DATABASE`». Принципиальным ограничением является невозможность копирования базы данных шаблона, если установлены другие подключения к ней. `CREATE DATABASE` выдаёт ошибку, если при запуске команды есть другие подключения к этой базе; в противном случае новые подключения к базе блокируются до завершения команды `CREATE DATABASE`. За дополнительными сведениями обратитесь к Разделу 21.3.

Кодировка символов, указанная для новой базы данных, должна быть совместима с выбранными параметрами локали (`LC_COLLATE` и `LC_CTYPE`). Если выбрана локаль `C` (или равнозначная ей `POSIX`), допускаются все кодировки, но для других локалей правильно будет работать только одна кодировка. (В Windows, однако, кодировку UTF-8 можно использовать с любой локалью.) `CREATE DATABASE` позволяет суперпользователям указать кодировку `SQL_ASCII` вне зависимости от локали, но этот вариант считается устаревшим и может привести к ошибочному поведению строковых функций, если в базе хранятся данные в кодировке, несовместимой с заданной локалью.

Параметры локали и кодировка должны соответствовать тем, что установлены в шаблоне, если только это не `template0`. Это ограничение объясняется тем, что другие базы данных могут содержать данные в кодировке, отличной от заданной, или индексы, порядок сортировки которых определяются параметрами `LC_COLLATE` и `LC_CTYPE`. При копировании таких данных получится база, которая будет испорченной согласно новым параметрам локали. Однако `template0` определённо не содержит какие-либо данные или индексы, зависящие от кодировки или локали.

В настоящее время использовать для базы данных локаль с недетерминированными сравнениями нельзя (объяснение этому можно найти в описании `CREATE COLLATION`). Если нужны такие сравнения, требуемые правила сортировки следует устанавливать на уровне столбцов.

Ограничение `CONNECTION LIMIT` действует только приблизительно; если одновременно запускаются два сеанса, тогда как в базе остаётся только одно «свободное место», может так случиться, что будут отклонены оба подключения. Кроме того, это ограничение не распространяется на суперпользователей и фоновые рабочие процессы.

Примеры

Создание базы данных:

```
CREATE DATABASE lusiadas;
```

Создание базы данных `sales`, принадлежащей пользователю `salesapp`, с табличным пространством по умолчанию `salesspace`:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

Создание базы данных `music` с другой локалью:

```
CREATE DATABASE music
    LOCALE 'sv_SE.utf8'
    TEMPLATE template0;
```

В этом примере предложение `TEMPLATE template0` необходимо, только если указанная локаль отличается от локали в `template1`. (В противном случае явное указание локали является избыточным.)

Создание базы данных `music2` с другой локалью и другой кодировкой символов:

```
CREATE DATABASE music2
    LOCALE 'sv_SE.iso885915'
    ENCODING LATIN9
    TEMPLATE template0;
```

Свойства кодировки должны соответствовать локали, иначе возникнет ошибка.

Заметьте, что имена локалей зависят от операционной системы, так что показанные выше команды могут не везде работать одинаково.

Совместимость

Оператор `CREATE DATABASE` отсутствует в стандарте SQL. Базы данных равнозначны каталогам, а их создание в стандарте определяется реализацией.

См. также

ALTER DATABASE, DROP DATABASE

CREATE DOMAIN

CREATE DOMAIN — создать домен

Синтаксис

```
CREATE DOMAIN имя [ AS ] тип_данных
    [ COLLATE правило_сортировки ]
    [ DEFAULT выражение ]
    [ ограничение_домена [ ... ] ]

Здесь ограничение_домена:

[ CONSTRAINT имя_ограничения ]
{ NOT NULL | NULL | CHECK (выражение) }
```

Описание

`CREATE DOMAIN` создаёт новый домен. Домен по сути представляет собой тип данных с дополнительными условиями (ограничивающими допустимый набор значений). Владельцем домена становится пользователь его создавший.

Если задаётся имя схемы (например, `CREATE DOMAIN myschema.mydomain ...`), домен создаётся в указанной схеме, в противном случае — в текущей. Имя домена должно быть уникальным среди имён типов и доменов, существующих в этой схеме.

Домены полезны для абстрагирования и вынесения общих характеристик разных полей в единое место для упрощения сопровождения. Например, в нескольких таблицах может присутствовать столбец, содержащий электронный адрес, и для всех требуются одинаковые ограничения CHECK, проверяющие синтаксис адреса. В этом случае лучше определить домен, а не задавать для каждой таблицы отдельные ограничения.

Чтобы создать домен, необходимо иметь право `USAGE` для нижележащего типа.

Параметры

`имя`

Имя создаваемого домена (возможно, дополненное схемой).

`тип_данных`

Нижележащий тип данных домена (может включать определение массива с этим типом).

`правило_сортировки`

Необязательное указание правила сортировки для домена. Если это указание отсутствует, в домене используется правило сортировки нижележащего типа данных. Указать `COLLATE` можно, только если нижележащий тип данных является сортируемым.

`DEFAULT` *`выражение`*

Предложение `DEFAULT` определяет значение по умолчанию для столбцов, типом данных которых является этот домен. Значением может быть любое выражение без переменных (подзапросы также не допускаются). Тип данных этого выражения должен соответствовать типу данных домена. Если значение по умолчанию не указано, им будет значение NULL.

Значение по умолчанию будет использоваться в любой операции добавления строк, в которой не задано значение для этого столбца. Если значение по умолчанию установлено для конкретного столбца, оно будет переопределять значение по умолчанию, связанное с доменом. В свою очередь, значение по умолчанию для домена переопределяет любое значение по умолчанию, связанное с нижележащим типом данных.

`CONSTRAINT` *`имя_ограничения`*

Имя ограничения. Если не указано явно, имя будет сгенерировано системой.

`NOT NULL`

Значения этого домена будут отличны от NULL (но см. замечания ниже).

`NULL`

Этот домен может содержать значение NULL. Это свойство домена по умолчанию.

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

`CHECK (`*`выражение`*`)`

Предложения `CHECK` задают ограничения целостности или проверки, которым должны удовлетворять значения домена. Каждое ограничение должно представлять собой выражение, выдающее результат типа Boolean. Проверяемое значение в этом выражении обозначается ключевым словом `VALUE`. Если выражение выдаёт FALSE, сообщается об ошибке и приведение значения к типу домена запрещается.

В настоящее время выражения `CHECK` не могут содержать переменные, кроме `VALUE`, и подзапросы.

Когда для домена задано несколько ограничений `CHECK`, они будут проверяться в алфавитном порядке имён. (До версии 9.5 в PostgreSQL не было установлено никакого определённого порядка обработки ограничений `CHECK`.)

Примечания

Ограничения домена, в частности `NOT NULL`, проверяются при преобразовании значения к типу домена. Однако из столбца, который номинально имеет тип домена, всё же можно прочитать NULL, несмотря на такое ограничение. Например, это может происходить в запросе внешнего соединения, если столбец домена окажется в обнуляемой стороне внешнего соединения. Более тонкий пример:

```
INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false));
```

Пустой скалярный вложенный SELECT выдаст значение NULL, типом которого будет считаться домен, так что к этому значению не будут применены дополнительные проверки ограничений и строка будет успешно добавлена.

Избежать таких проблем очень сложно, так как в SQL вообще предполагается, что значение NULL является подходящим для любого типа данных. Таким образом, лучше всего разрабатывать ограничения так, чтобы значения NULL допускались, а затем при необходимости применять ограничения `NOT NULL` к столбцам доменного типа, а не непосредственно к самому этому типу.

В Postgres Pro предполагается, что условия ограничений `CHECK` являются постоянными, то есть при одинаковых входных значениях они всегда выдают одинаковый результат. Именно этим предположением оправдывается то, что ограничения `CHECK` проверяются только при первом преобразовании значения в тип домена, а не при каждом обращении к нему. (По сути таким же образом обрабатываются ограничения `CHECK` для таблиц, как описано в Подразделе 5.5.1.)

Однако это предположение может нарушаться, как часто бывает, когда в выражении `CHECK` используется пользовательская функция, поведение которой впоследствии меняется. Postgres Pro не запрещает этого, и если сохранённые значения типа домена перестанут удовлетворять ограничению `CHECK`, это останется незамеченным. В итоге при попытке загрузить выгруженные позже данные могут возникнуть проблемы. Поэтому подобные изменения рекомендуется осуществлять следующим образом: удалить ограничение (используя `ALTER DOMAIN`), изменить определение функции, а затем пересоздать ограничение той же командой, которая при этом перепроверит сохранённые данные.

Рекомендуется следить за тем, чтобы выражения `CHECK` не выдавали ошибки.

Примеры

В этом примере создаётся тип данных `us_postal_code` (почтовый индекс США), который затем используется в определении таблицы. Для проверки значения на соответствие формату почтовых индексов США применяется проверка с регулярными выражениями:

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

Совместимость

Команда `CREATE DOMAIN` соответствует стандарту SQL.

Синтаксис `NOT NULL` данной команды является расширением Postgres Pro (соответствующий стандарту аналог для несоставных типов данных — это `CHECK (VALUE IS NOT NULL)`; однако в соответствии с Разделом «Примечания» подобных ограничений лучше избегать). «Ограничение» `NULL` является расширением Postgres Pro (см. также Совместимость).

См. также

ALTER DOMAIN, DROP DOMAIN

CREATE EVENT TRIGGER

CREATE EVENT TRIGGER — создать событийный триггер

Синтаксис

```
CREATE EVENT TRIGGER имя
    ON событие
    [ WHEN переменная_фильтра IN (значение_фильтра [, ... ]) [ AND ... ] ]
    EXECUTE { FUNCTION | PROCEDURE } имя_функции()
```

Описание

`CREATE EVENT TRIGGER` создаёт новый событийный триггер. Функция триггера выполняется, когда происходит указанное событие и удовлетворяется связанное с триггером условие `WHEN` (если такое имеется). За вводной информацией по триггерам обратитесь к Главе 38. Владельцем триггера становится пользователь его создавший.

Параметры

имя

Имя, назначаемое новому триггеру. Это имя должно быть уникальным в базе данных.

событие

Имя события, при котором срабатывает триггер и вызывается заданная функция. Подробнее об именах событий можно узнать в Разделе 38.1.

переменная_фильтра

Имя переменной, применяемой для фильтрования событий. Это указание позволяет ограничить срабатывание триггера подмножеством случаев, в которых он поддерживается. В настоящее время единственно возможное значение параметра *переменная_фильтра* — `TAG`.

значение_фильтра

Список значений связанного параметра *переменная_фильтра*, для которых должен срабатывать триггер. Для переменной `TAG` это список меток команд (например, `'DROP FUNCTION'`).

имя_функции

Заданная пользователем функция, объявленная как функция без аргументов и возвращающая тип `event_trigger`.

В синтаксисе `CREATE EVENT TRIGGER` ключевые слова `FUNCTION` и `PROCEDURE` равнозначны, но указываемая функция должна в любом случае быть функцией, а не процедурой. Ключевое слово `PROCEDURE` здесь поддерживается по историческим причинам и считается устаревшим.

Примечания

Создавать событийные триггеры могут только суперпользователи.

Событийные триггеры не вызываются в однопользовательском режиме (см. postgres) и при значении `false` у параметра event_triggers. Если ошибочный событийный триггер заблокировал работу с базой данных так, что даже удалить его нельзя, перезапустите сервер, задав значение `false` для параметра event_triggers, или в однопользовательском режиме, и его можно будет удалить.

Примеры

Триггер, запрещающий выполнение любой команды DDL:

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE FUNCTION abort_any_command();
```

Совместимость

Оператор `CREATE EVENT TRIGGER` отсутствует в стандарте SQL.

См. также

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION

CREATE EXTENSION

CREATE EXTENSION — установить расширение

Синтаксис

```
CREATE EXTENSION [ IF NOT EXISTS ] имя_расширения
    [ WITH ] [ SCHEMA имя_схемы ]
             [ VERSION версия ]
             [ CASCADE ]
```

Описание

`CREATE EXTENSION` загружает в текущую базу данных новое расширение. Расширение с таким именем не должно быть уже загружено.

Загрузка расширения по сути сводится к запуску скрипта расширения. Этот скрипт обычно создаёт новые SQL-объекты, такие как функции, типы данных, операторы и методы поддержки индексов. `CREATE EXTENSION` дополнительно записывает идентификаторы всех добавляемых объектов, так что впоследствии их можно удалить, выполнив команду `DROP EXTENSION`.

Пользователь, выполняющий `CREATE EXTENSION`, становится владельцем самого расширения, что определяет его права в дальнейшем, а также обычно он становится владельцем всех объектов, созданных скриптом расширения.

Для загрузки расширения как правило требуются те же права, что необходимы для создания составляющих его объектов. Для многих расширений это означает, что необходимы права суперпользователя. Однако если расширение помечено как `trusted` (*доверенное*) в своём управляющем файле, его может установить любой пользователь, имеющий в базе данных право `CREATE`. В этом случае сам объект расширения будет принадлежать вызывающему пользователю, но владельцем содержащихся в нём объектов будет начальный суперпользователь (если только вызывающий пользователь не назначается их владельцем в скрипте расширения). Такая схема даёт вызывающему пользователю возможность удалить расширение, но не модифицировать отдельные объекты внутри него.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если расширение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее расширение как-то соотносится с тем, которое могло бы быть создано из указанного скрипта.

`имя_расширения`

Имя устанавливаемого расширения. Postgres Pro создаст расширение, используя инструкции из файла `SHAREDIR/extension/`*`имя_расширения`*`.control`.

`имя_схемы`

Имя схемы, в которую будут установлены объекты расширения (подразумевается, что расширение позволяет управлять размещением своих объектов). Указанная схема должна уже существовать. Если имя не указано и в управляющем файле расширения оно так же не задано, для создания объектов используется текущая схема.

Если в управляющем файле расширения задаётся параметр `schema`, заданную схему нельзя переопределить предложением `SCHEMA`. Обычно при указании предложения `SCHEMA` возникает ошибка, если эта схема конфликтует с параметром `schema` данного расширения. Однако если также задаётся предложение `CASCADE`, в случае конфликта *`имя_схемы`* игнорируется. Заданное *имя_схемы* будет использоваться для установки всех необходимых расширений, в управляющих файлах которых не задаётся `schema`.

Помните, что само расширение не считается принадлежащим какой-либо схеме; имена расширений не дополняются схемой и потому должны быть уникальными во всей базе данных. Однако объекты, принадлежащие расширениям, могут относиться к схемам.

версия

Версия устанавливаемого расширения. Её можно записать в виде идентификатора или строкового значения. По умолчанию версия считывается из управляющего файла расширения.

`CASCADE`

Автоматически устанавливать все расширения, от которого зависит данное, если они ещё не установлены. Их зависимости подобным образом рекурсивно устанавливаются автоматически. Предложение `SCHEMA`, если задано, применяется ко всем расширениям, устанавливаемым таким способом. Другие параметры оператора к автоматически устанавливаемым расширениям не применяются; в частности, всегда выбираются их версии по умолчанию.

Примечания

Прежде чем вы сможете выполнить `CREATE EXTENSION` и загрузить расширение в базу данных, необходимо правильно установить сопутствующие файлы расширения. Информацию об установке расширений, поставляемых в составе Postgres Pro, можно найти по ссылке Дополнительные поставляемые модули.

Расширения, доступные для установки в данный момент, можно найти в системном представлении `pg_available_extensions` или `pg_available_extension_versions`.

Внимание

Устанавливая расширение от имени суперпользователя, важно иметь уверенность в том, что автор расширения написал установочный скрипт безопасным образом. Для злонамеренного пользователя не составит большого труда создать объект типа троянского коня, который впоследствии скомпрометирует выполнение неаккуратно написанного скрипта расширения и позволит этому пользователю стать суперпользователем. Однако такие объекты опасны, только если они находятся в пути `search_path` во время выполнения скрипта, то есть, если они находятся в схеме, в которую устанавливается расширение, или в схеме, где располагается другое расширение, от которого зависит первое. Таким образом, имея дело с расширениями, скрипты которых не были тщательно проверены, рекомендуется устанавливать их только в те схемы, в которых недоверенные пользователи не имеют и не будут иметь права CREATE. То же самое касается их зависимостей, других расширений.

Расширения, поставляемые в составе Postgres Pro, можно считать защищёнными от подобного рода атак времени выполнения, за исключением некоторых, зависящих от других расширений. Как отмечается в документации таких расширений, их следует устанавливать в безопасные схемы и/или в те же схемы, в которые устанавливаются требующиеся им расширения.

За информацией для разработчиков расширений обратитесь к Разделу 36.17.

Примеры

Установка расширения hstore в текущую базу данных (при этом объекты расширения размещаются в схеме `addons`):

```
CREATE EXTENSION hstore SCHEMA addons;
```

Другой способ сделать то же самое:

```
SET search_path = addons;
CREATE EXTENSION hstore;
```

Совместимость

`CREATE EXTENSION` является расширением Postgres Pro.

См. также

ALTER EXTENSION, DROP EXTENSION

CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER — создать новую обёртку сторонних данных

Синтаксис

```
CREATE FOREIGN DATA WRAPPER имя
    [ HANDLER функция_обработчик | NO HANDLER ]
    [ VALIDATOR функция_проверки | NO VALIDATOR ]
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Описание

`CREATE FOREIGN DATA WRAPPER` создаёт обёртку сторонних данных. Владельцем обёртки становится создавший её пользователь.

Имя обёртки сторонних данных должно быть уникальным в базе данных.

Создавать обёртки сторонних данных могут только суперпользователи.

Параметры

`имя`

Имя создаваемой обёртки сторонних данных.

`HANDLER` *`функция_обработчик`*

В аргументе *`функция_обработчик`* указывается имя ранее зарегистрированной функции, которая будет вызываться для получения функций, реализующих обращения к сторонним таблицам. Функция-обработчик не принимает аргументы и возвращает результат типа `fdw_handler`.

Обёртку сторонних таблиц можно создать и без функции-обработчика, но через такую обёртку нельзя будет использовать сторонние таблицы, хотя объявить их вполне возможно.

`VALIDATOR` *`функция_проверки`*

В аргументе *`функция_проверки`* указывается имя ранее зарегистрированной функции, которая будет вызываться для проверки общих параметров, передаваемых обёртке сторонних данных, а также параметров сторонних серверов, сопоставлений пользователей и сторонних таблиц, доступных через эту обёртку. Если функция проверки не задана или указано `NO VALIDATOR`, параметры не будут проверяться во время создания объектов. (Обёртка сторонних данных может игнорировать или не принимать неверные указания параметров во время выполнения, в зависимости от реализации.) Функция проверки должна принимать два аргумента: первый типа `text[]` (в нём содержится массив параметров, хранящихся в системном каталоге), а второй типа `oid` (в нём указывается OID системного каталога с этими параметрами). Возвращаемое значение игнорируется; функция проверки должна сообщать о неверных параметрах, вызывая системную функцию `ereport(ERROR)`.

`OPTIONS (` *`параметр`* `'`*`значение`*`' [, ... ] )`

Это предложение определяет параметры для создаваемой обёртки сторонних данных. Набор допустимых параметров и значений для каждой обёртки свой, контроль их правильности осуществляет функция проверки сторонних данных. Имена параметров должны быть уникальными.

Примечания

Функциональность Postgres Pro по работе со сторонними данными продолжает активно развиваться. На данный момент выполняется только примитивная оптимизация запросов (и по большей части это тоже делает обёртка), так что в этом направлении есть поле для улучшения производительности.

Примеры

Создание бесполезной обёртки сторонних данных `dummy`:

```
CREATE FOREIGN DATA WRAPPER dummy;
```

Создание обёртки сторонних данных `file` с функцией-обработчиком `file_fdw_handler`:

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

Создание обёртки сторонних данных `mywrapper` с параметрами:

```
CREATE FOREIGN DATA WRAPPER mywrapper
    OPTIONS (debug 'true');
```

Совместимость

`CREATE FOREIGN DATA WRAPPER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED), за исключением того, что предложения `HANDLER` и `VALIDATOR` стандартом не предусмотрены, а предложения `LIBRARY` и `LANGUAGE`, напротив, не реализованы в Postgres Pro.

Учтите, однако, что функциональность SQL/MED в целом ещё не обеспечивается.

См. также

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

CREATE FOREIGN TABLE — создать стороннюю таблицу

Синтаксис

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] имя_таблицы ( [
  { имя_столбца тип_данных [ OPTIONS ( параметр 'значение' [, ... ] ) ]
 [ COLLATE правило_сортировки ] [ ограничение_столбца [ ... ] ]
    | ограничение_таблицы }
    [, ... ]
] )
[ INHERITS ( таблица_родитель [, ... ] ) ]
  SERVER имя_сервера
[ OPTIONS ( параметр 'значение' [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] имя_таблицы
  PARTITION OF таблица_родитель [ (
  { имя_столбца [ WITH OPTIONS ] [ ограничение_столбца [ ... ] ]
    | ограничение_таблицы }
    [, ... ]
) ]
{ FOR VALUES указание_границ_секции | DEFAULT }
  SERVER имя_сервера
[ OPTIONS ( параметр 'значение' [, ... ] ) ]

Здесь ограничение_столбца:

[ CONSTRAINT имя_ограничения ]
{ NOT NULL |
  NULL |
  CHECK ( выражение ) [ NO INHERIT ] |
  DEFAULT выражение_по_умолчанию |
  GENERATED ALWAYS AS ( генерирующее_выражение ) STORED }

и ограничение_таблицы:

[ CONSTRAINT имя_ограничения ]
CHECK ( выражение ) [ NO INHERIT ]

и указание_границ_секции:

IN ( выражение_границ_секции [, ...] ) |
FROM ( { выражение_границ_секции | MINVALUE | MAXVALUE } [, ...] )
  TO ( { выражение_границ_секции | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS числовая_константа, REMAINDER числовая_константа )
```

Описание

`CREATE FOREIGN TABLE` создаёт новую стороннюю таблицу в текущей базе данных. Владельцем таблицы будет пользователь, выполнивший эту команду.

Если указано имя схемы (например, `CREATE FOREIGN TABLE myschema.mytable ...`), таблица будет создана в этой схеме. В противном случае она создаётся в текущей схеме. Имя сторонней таблицы должно отличаться от имён других отношений (таблиц, последовательностей, индексов, представлений, материализованных представлений или сторонних таблиц), существующих в этой схеме.

`CREATE FOREIGN TABLE` также автоматически создаёт составной тип данных, соответствующий одной строке сторонней таблицы. Таким образом, имя сторонней таблицы не может совпадать с именем существующего типа в этой же схеме.

Если указано предложение `PARTITION OF`, таблица создаётся в виде секции `parent_table` с указанными границами.

Чтобы создать стороннюю таблицу, необходимо иметь право `USAGE` для стороннего сервера, а также право `USAGE` для всех типов столбцов, содержащихся в таблице.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с тем, которое могло бы быть создано.

`имя_таблицы`

Имя создаваемой таблицы (возможно, дополненное схемой).

`имя_столбца`

Имя столбца, создаваемого в новой таблице.

`тип_данных`

Тип данных столбца (может включать определение массива с этим типом). За дополнительными сведениями о типах данных, которые поддерживает Postgres Pro, обратитесь к Главе 8.

`COLLATE` *`правило_сортировки`*

Предложение `COLLATE` назначает правило сортировки для столбца (который должен иметь тип, поддерживающий сортировку). Если оно отсутствует, используется правило сортировки по умолчанию, установленное для типа данных столбца.

`INHERITS (` *`таблица_родитель`* `[, ... ] )`

Необязательное предложение `INHERITS` определяет список таблиц, от которых новая сторонняя таблица будет автоматически наследовать все столбцы. Родительскими таблицами могут быть обычные или сторонние таблицы. За подробностями обратитесь к описанию подобной формы `CREATE TABLE`.

`PARTITION OF` *`таблица_родитель`* `{ FOR VALUES` *`указание_границ_секции`* `| DEFAULT }`

Эта форма может использоваться для создания сторонней таблицы в виде секции указанной родительской таблицы с заданными граничными значениями. За подробностями обратитесь к описанию подобной формы команды `CREATE TABLE`. Заметьте, что в настоящее время не допускается создание сторонней таблицы в виде секции родительской таблицы, если в этой таблице есть уникальные индексы (`UNIQUE`). (См. также описание `ALTER TABLE ATTACH PARTITION`.)

`CONSTRAINT` *`имя_ограничения`*

Необязательное имя столбца или ограничения таблицы. При нарушении ограничения его имя будет выводиться в сообщении об ошибках, так что имена ограничений вида `столбец должен быть положительным` могут сообщить полезную информацию об ограничении клиентскому приложению. (Имена ограничений, включающие пробелы, необходимо заключать в двойные кавычки.) Если имя ограничения не указано, система генерирует имя автоматически.

`NOT NULL`

Данный столбец не принимает значения NULL.

`NULL`

Данный столбец может содержать значения NULL (по умолчанию).

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

`CHECK (` *`выражение`* `) [ NO INHERIT ]`

В ограничении `CHECK` задаётся выражение, возвращающее логический результат, которому должны удовлетворять все строки в сторонней таблице; то есть это выражение должно выдавать TRUE или UNKNOWN, но никогда FALSE, для всех строк в сторонней таблице. Ограничение-проверка, заданное как ограничение столбца, должно ссылаться только на значение самого столбца, тогда как ограничение на уровне таблицы может ссылаться и на несколько столбцов.

В настоящее время выражения `CHECK` не могут содержать подзапросы или ссылаться на переменные, кроме как на столбцы текущей строки. Также допустима ссылка на системный столбец `tableoid`, но не на другие системные столбцы.

Ограничение с пометкой `NO INHERIT` не будет наследоваться дочерними таблицами.

`DEFAULT` *`выражение_по_умолчанию`*

Предложение `DEFAULT` задаёт значение по умолчанию для столбца, в определении которого оно присутствует. Значение задаётся выражением без переменных (подзапросы и перекрёстные ссылки на другие столбцы текущей таблицы в нём не допускаются). Тип данных выражения, задающего значение по умолчанию, должен соответствовать типу данных столбца.

Это выражение будет использоваться во всех операциях добавления данных, в которых не задаётся значение данного столбца. Если значение по умолчанию не определено, таким значением будет NULL.

`GENERATED ALWAYS AS (` *`генерирующее_выражение`* `) STORED`

Это предложение создаёт столбец как *генерируемый*. В такой столбец нельзя записать данные, а при чтении его возвращается результат указанного выражения.

Ключевое слово `STORED` отмечает, что этот столбец будет вычисляться при записи. (Вычисленное значение будет передаваться обёртке сторонних данных, которая должна сохранить его и затем выдавать при чтении.)

Генерирующее выражение может обращаться к другим столбцам таблицы, но не к другим генерируемым столбцам. Все функции и операторы в нём должны быть постоянными. Обращаться к другим таблицам в таких выражениях нельзя.

`имя_сервера`

Имя существующего стороннего сервера, предоставляющего данную стороннюю таблицу. О создании сервера можно узнать в CREATE SERVER.

`OPTIONS (` *`параметр`* `'`*`значение`*`' [, ...] )`

Параметры, связываемые с новой сторонней таблицей или одним из её столбцов. Допустимые имена и значения параметров у каждой обёртки сторонних данных свои; они контролируются функцией проверки, связанной с этой обёрткой. Имена параметров не должны повторяться (хотя параметр таблицы и параметр столбца вполне могут иметь одно имя).

Примечания

Ограничения сторонних таблиц (например, `CHECK` и `NOT NULL`) не контролируются ядром системы Postgres Pro, как не пытаются их контролировать и большинство обёрток сторонних данных; то есть, система просто предполагает, что ограничение выполняется. Контролировать такое ограничение не имело бы большого смысла, так как оно применялось бы только к строкам, добавляемым или изменяемым через стороннюю таблицу, но не к строкам, модифицируемым другим путём, например, непосредственно на удалённом сервере. Вместо этого, ограничение, связанное со сторонней таблицей, должно представлять ограничение, выполнение которого обеспечивает удалённый сервер.

Некоторые специализированные обёртки сторонних данных могут быть единственным вариантом обращения к доступным через них данным, и в этом случае может быть уместно реализовать контроль ограничений в самой такой обёртке. Но не следует полагать, что какая-либо обёртка ведёт себя так, если об этом не сказано явно в её документации.

Хотя Postgres Pro не пытается контролировать ограничения для сторонних таблиц, он полагает, что они выполняются для целей оптимизации запросов. Если в сторонней таблице будут видны строки, не удовлетворяющие объявленному ограничению, запросы к этой таблице могут выдавать ошибки или некорректные результаты. Ответственность за фактическое выполнение условия ограничения лежит на пользователе.

Внимание

Когда сторонняя таблица используется в качестве секции секционированной таблицы, существует неявное ограничение, согласно которому её содержимое должно отвечать правилу секционирования. Обеспечивать соблюдение этого правила должен пользователь; для этого лучше всего установить соответствующее ограничение на удалённом сервере.

В секционированной таблице, содержащей секции сторонних таблиц, команда `UPDATE`, изменяющая ключ секционирования, может привести к перемещению строки из локальной секции в секцию в сторонней таблице при условии, что обёртка сторонних данных поддерживает перенаправление кортежей. Однако перемещение строки из секции в сторонней таблице в другую секцию в настоящее время невозможно. Команда `UPDATE`, вызывающая такое перемещение, прервётся ошибкой из-за ограничения секции, если на удалённом сервере установлено нужное ограничение.

Подобные соображения распространяются и на генерируемые столбцы. Сохранённые генерируемые столбцы вычисляются при добавлении или изменении данных на локальном сервере Postgres Pro и передаются обёртке сторонних данных, которая должна записать их в стороннее хранилище данных. При этом сервер не требует, чтобы возвращаемые при обращении к сторонней таблице значения генерируемых столбцов согласовывались с генерирующим выражением. Таким образом, в случае несоответствия этих значений результаты запроса могут оказаться некорректными.

Примеры

Создание сторонней таблицы `films`, которая будет доступна через сервер `film_server`:

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
)
SERVER film_server;
```

Создание сторонней таблицы `measurement_y2016m07`, которая будет доступна через сервер `server_07`, в виде секции таблицы `measurement`, секционированной по диапазонам:

```
CREATE FOREIGN TABLE measurement_y2016m07
    PARTITION OF measurement FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')
    SERVER server_07;
```

Совместимость

Команда `CREATE FOREIGN TABLE` в основном соответствует стандарту SQL; однако, как и `CREATE TABLE`, она допускает ограничения `NULL` и сторонние таблицы с нулём столбцов. Возможность задавать значения по умолчанию для столбцов также является расширением Postgres Pro. Наследование таблиц, в форме, определённой в Postgres Pro, стандарту не соответствует.

См. также

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER, IMPORT FOREIGN SCHEMA

CREATE FUNCTION

CREATE FUNCTION — создать функцию

Синтаксис

```
CREATE [ OR REPLACE ] FUNCTION
    имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ { DEFAULT |
 = } выражение_по_умолчанию ] [, ...] ] )
    [ RETURNS тип_результата
      | RETURNS TABLE ( имя_столбца тип_столбца [, ...] ) ]
  { LANGUAGE имя_языка
    | TRANSFORM { FOR TYPE имя_типа } [, ... ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST стоимость_выполнения
    | ROWS строк_в_результате
    | SUPPORT вспомогательная_функция
    | SET параметр_конфигурации { TO значение | = значение | FROM CURRENT }
    | AS 'определение'
    | AS 'объектный_файл', 'объектный_символ'
    | тело_sql
  } ...
```

Описание

Команда `CREATE FUNCTION` определяет новую функцию. `CREATE OR REPLACE FUNCTION` создаёт новую функцию, либо заменяет определение уже существующей. Чтобы определить функцию, необходимо иметь право `USAGE` для соответствующего языка.

Если указано имя схемы, функция создаётся в заданной схеме, в противном случае — в текущей. Имя новой функции должно отличаться от имён существующих функций или процедур с такими же типами аргументов в этой схеме. Однако функции и процедуры с аргументами разных типов могут иметь одно имя (это называется *перегрузкой*).

Чтобы заменить текущее определение существующей функции, используйте команду `CREATE OR REPLACE FUNCTION`. Но учтите, что она не позволяет изменить имя или аргументы функции (если попытаться сделать это, на самом деле будет создана новая, независимая функция). Кроме того, `CREATE OR REPLACE FUNCTION` не позволит изменить тип результата существующей функции. Чтобы сделать это, придётся удалить функцию и создать её заново. (Это означает, что если функция имеет выходные параметры (`OUT`), то изменить типы параметров `OUT` можно, только удалив функцию.)

Когда команда `CREATE OR REPLACE FUNCTION` заменяет существующую функцию, владелец и права доступа к этой функции не меняются. Все другие свойства функции получают значения, задаваемые командой явно или по умолчанию. Чтобы заменить функцию, необходимо быть её владельцем (или быть членом роли-владельца).

Если вы удалите и затем вновь создадите функцию, новая функция станет другой сущностью, отличной от старой; вам потребуется так же удалить существующие правила, представления, триггеры и т. п., ссылающиеся на старую функцию. Поэтому, чтобы изменить определение функции, сохраняя ссылающиеся на неё объекты, следует использовать `CREATE OR REPLACE FUNCTION`. Кроме того, многие дополнительные свойства существующей функции можно изменить с помощью `ALTER FUNCTION`.

Владельцем функции становится создавший её пользователь.

Чтобы создать функцию, необходимо иметь право `USAGE` для типов её аргументов и возвращаемого типа.

За дополнительной информацией о разработке функций обратитесь к Разделу 36.3.

Параметры

`имя`

Имя создаваемой функции (возможно, дополненное схемой).

`режим_аргумента`

Режим аргумента: `IN` (входной), `OUT` (выходной), `INOUT` (входной и выходной) или `VARIADIC` (переменный). По умолчанию подразумевается `IN`. За единственным аргументом `VARIADIC` могут следовать только аргументы `OUT`. Кроме того, аргументы `OUT` и `INOUT` нельзя использовать с предложением `RETURNS TABLE`.

`имя_аргумента`

Имя аргумента. Некоторые языки (включая SQL и PL/pgSQL) позволяют использовать это имя в теле функции. Для других языков это имя служит просто дополнительным описанием, если говорить о самой функции; однако вы можете указывать имена аргументов при вызове функции для улучшения читаемости (см. Раздел 4.3). Имя выходного аргумента в любом случае имеет значение, так как оно определяет имя столбца в типе результата. (Если вы опустите имя выходного аргумента, система выберет для него имя по умолчанию.)

`тип_аргумента`

Тип данных аргумента функции (возможно, дополненный схемой), при наличии аргументов. Тип аргументов может быть базовым, составным или доменным, либо это может быть ссылка на столбец таблицы.

В зависимости от языка реализации также может допускаться указание «псевдотипов», например, `cstring`. Псевдотипы показывают, что фактический тип аргумента либо определён не полностью, либо существует вне множества обычных типов SQL.

Ссылка на тип столбца записывается в виде *`имя_таблицы`*`.`*`имя_столбца`*`%TYPE`. Иногда такое указание бывает полезно, так как позволяет создать функцию, независящую от изменений в определении таблицы.

`выражение_по_умолчанию`

Выражение, используемое для вычисления значения по умолчанию, если параметр не задан явно. Результат выражения должен сводиться к типу соответствующего параметра. Значения по умолчанию могут иметь только входные параметры (включая `INOUT`). Для всех входных параметров, следующих за параметром с определённым значением по умолчанию, также должны быть определены значения по умолчанию.

`тип_результата`

Тип возвращаемых данных (возможно, дополненный схемой). Это может быть базовый, составной или доменный тип, либо ссылка на тип столбца таблицы. В зависимости от языка реализации здесь также могут допускаться «псевдотипы», например `cstring`. Если функция не должна возвращать значение, в качестве типа результата указывается `void`.

В случае наличия параметров `OUT` или `INOUT`, предложение `RETURNS` можно опустить. Если оно присутствует, оно должно согласовываться с типом результата, выводимым из выходных параметров: в качестве возвращаемого типа указывается `RECORD`, если выходных параметров несколько, либо тип единственного выходного параметра.

Указание `SETOF` показывает, что функция возвращает множество, а не единственный элемент.

Ссылка на тип столбца записывается в виде *`имя_таблицы`*`.`*`имя_столбца`*`%TYPE`.

`имя_столбца`

Имя выходного столбца в записи `RETURNS TABLE`. По сути это ещё один способ объявить именованный выходной параметр (`OUT`), но `RETURNS TABLE` также подразумевает и `RETURNS SETOF`.

`тип_столбца`

Тип данных выходного столбца в записи `RETURNS TABLE`.

`имя_языка`

Имя языка, на котором реализована функция. Это может быть `sql`, `c`, `internal`, либо имя процедурного языка, определённого пользователем, например, `plpgsql`. Если присутствует *`тело_sql`*, подразумевается язык `sql`. Вариант написания этого имени в апострофах считается устаревшим и требует точного совпадения регистра.

`TRANSFORM { FOR TYPE` *`имя_типа`* `} [, ... ] }`

Устанавливает список трансформаций, которые должны применяться при вызове функции. Трансформации выполняют преобразования между типами SQL и типами данных, специфичными для языков; см. CREATE TRANSFORM. Преобразования встроенных типов обычно жёстко предопределены в реализациях процедурных языков, так что их здесь указывать не нужно. Если реализация процедурного языка не может обработать тип и трансформация для него отсутствует, будет выполнено преобразование типов по умолчанию, но это зависит от реализации.

`WINDOW`

Указание `WINDOW` показывает, что создаётся не простая, а *оконная функция*. В настоящее время это имеет смысл только для функций, написанных на C. Атрибут `WINDOW` нельзя изменить, модифицируя впоследствии определение функции.

`IMMUTABLE`
`STABLE`
`VOLATILE`

Эти атрибуты информируют оптимизатор запросов о поведении функции. Одновременно можно указать не более одного атрибута. Если никакой атрибут не задан, по умолчанию подразумевается `VOLATILE`.

Характеристика `IMMUTABLE` (постоянная) показывает, что функция не может модифицировать базу данных и всегда возвращает один и тот же результат при определённых значениях аргументов; то есть, она не обращается к базе данных и не использует информацию, не переданную ей явно в списке аргументов. Если функция имеет такую характеристику, любой её вызов с аргументами-константами можно немедленно заменить значением функции.

Характеристика `STABLE` (стабильная) показывает, что функция не может модифицировать базу данных и в рамках одного сканирования таблицы она всегда возвращает один и тот же результат для определённых значений аргументов, но этот результат может быть разным в разных операторах SQL. Это подходящий выбор для функций, результаты которых зависят от содержимого базы данных и настраиваемых параметров (например, текущего часового пояса). (Но этот вариант не подходит для триггеров `AFTER`, желающих прочитать строки, изменённые текущей командой.) Также заметьте, что функции семейства `current_timestamp` также считаются стабильными, так как их результаты не меняются внутри транзакции.

Характеристика `VOLATILE` (изменчивая) показывает, что результат функции может меняться даже в рамках одного сканирования таблицы, так что её вызовы нельзя оптимизировать. Изменчивы в этом смысле относительно немногие функции баз данных, например: `random()`, `currval()` и `timeofday()`. Но заметьте, что любая функция с побочными эффектами должна быть классифицирована как изменчивая, даже если её результат вполне предсказуем, чтобы её вызовы не были соптимизированы; пример такой функции: `setval()`.

За дополнительными подробностями обратитесь к Разделу 36.7.

`LEAKPROOF`

Характеристика `LEAKPROOF` (герметичная) показывает, что функция не имеет побочных эффектов. Она не раскрывает информацию о своих аргументах, кроме как возвращая результат. Например, функция, которая выдаёт сообщение об ошибке с некоторыми, но не всеми значениями аргументов, либо выводит значения аргументов в сообщении об ошибке, не является герметичной. Это влияет на то, как система выполняет запросы к представлениям, созданным с барьером безопасности (с указанием `security_barrier`), или к таблицам с включённой защитой строк. Во избежание неконтролируемой утечки данных система будет проверять условия из политик защиты и определений представлений с барьерами безопасности перед любыми условиями, которые задаёт пользователь в самом запросе и в которых задействуются негерметичные функции. Функции и операторы, помеченные как герметичные, считаются доверенными и могут выполняться перед условиями из политик защиты и представлений с барьерами безопасности. При этом функции, которые не имеют аргументов или которым не передаются никакие аргументы из представления с барьером безопасности или таблицы, не требуется помечать как герметичные, чтобы они выполнялись до условий, связанных с безопасностью. См. CREATE VIEW и Раздел 39.5. Это свойство может установить только суперпользователь.

`CALLED ON NULL INPUT`
`RETURNS NULL ON NULL INPUT`
`STRICT`

`CALLED ON NULL INPUT` (по умолчанию) показывает, что функция будет вызвана как обычно, если среди её аргументов оказываются значения NULL. В этом случае ответственность за проверку значений NULL и соответствующую их обработку ложится на разработчика функции.

Указание `RETURNS NULL ON NULL INPUT` или `STRICT` показывает, что функция всегда возвращает NULL, получив NULL в одном из аргументов. Такая функция не будет вызываться с аргументами NULL, вместо этого автоматически будет полагаться результат NULL.

`[EXTERNAL] SECURITY INVOKER`
`[EXTERNAL] SECURITY DEFINER`

Характеристика `SECURITY INVOKER` (безопасность вызывающего) показывает, что функция будет выполняться с правами пользователя, вызвавшего её. Этот вариант подразумевается по умолчанию. Вариант `SECURITY DEFINER` (безопасность определившего) показывает, что функция выполняется с правами пользователя, владеющего ей. Дополнительную информацию о безопасном создании функций с характеристикой `SECURITY DEFINER` см. ниже.

Ключевое слово `EXTERNAL` (внешняя) допускается для соответствия стандарту SQL, но является необязательным, так как, в отличие от SQL, эта характеристика распространяется на все функции, а не только внешние.

`PARALLEL`

Указание `PARALLEL UNSAFE` означает, что эту функцию нельзя выполнять в параллельном режиме и присутствие такой функции в операторе SQL приводит к выбору последовательного плана выполнения. Это характеристика функции по умолчанию. Указание `PARALLEL RESTRICTED` означает, что функцию можно выполнять в параллельном режиме, но только в ведущем процессе группы. `PARALLEL SAFE` показывает, что функция безопасна для выполнения в параллельном режиме без ограничений, в том числе в параллельных рабочих процессах.

Функции должны помечаться как небезопасные для параллельного выполнения, если они изменяют состояние базы данных, вносят изменения в транзакции (кроме использования подтранзакций для восстановления после сбоя), обращаются к последовательностям (например, используя `currval`) или сохраняют изменения в параметрах. Ограниченно параллельными должны помечаться функции, которые обращаются к временным таблицам, состоянию клиентского подключения, курсорам, подготовленным операторам или разнообразному состоянию обслуживающего процесса, которое система не может синхронизировать в параллельном режиме (например, `setseed` может выполнять только ведущий процесс группы, так как изменения, внесённые другим процессом, не передаются ведущему). В целом, если функция помечена как безопасная, тогда как она является ограниченной или небезопасной, либо если она помечена как ограниченно безопасная, не являясь безопасной, при попытке вызвать её в параллельном запросе она может выдавать ошибки или неверные результаты. Функции на языке C при неправильной пометке теоретически могут проявлять полностью неопределённое поведение, так как система никак не может защититься от произвольного кода на C, но чаще все они будут вести себя не хуже, чем любая другая функция. В случае сомнений функцию следует помечать как небезопасную (`UNSAFE`), что является значением по умолчанию.

`COST` *`стоимость_выполнения`*

Положительное число, задающее примерную стоимость выполнения функции, в единицах cpu_operator_cost. Если функция возвращает множество, это число задаёт стоимость для одной строки. Если стоимость не указана, для функций на C и внутренних функций она считается равной 1 единице, а для функций на всех других языках — 100 единицам. При больших значениях планировщик будет стараться не вызывать эту функцию чаще, чем это необходимо.

`ROWS` *`строк_в_результате`*

Положительное число, задающее примерное число строк, которое будет ожидать планировщик на выходе этой функции. Это указание допустимо, только если функция объявлена как возвращающая множество. Предполагаемое по умолчанию значение — 1000 строк.

`SUPPORT` *`вспомогательная_функция`*

Имя (возможно, дополненное схемой) *вспомогательной функции для планировщика*, которая будет использоваться этой функцией. За подробностями обратитесь к Разделу 36.11. Для использования этого указания нужно быть суперпользователем.

`параметр_конфигурации`
`значение`

Предложение `SET` определяет, что при вызове функции указанный параметр конфигурации должен принять заданное значение, а затем восстановить своё предыдущее значение при завершении функции. Предложение `SET FROM CURRENT` сохраняет в качестве значения, которое будет применено при входе в функцию, значение, действующее в момент выполнения `CREATE FUNCTION`.

Если в определение функции добавлено `SET`, то действие команды `SET LOCAL`, выполняемой внутри функции для того же параметра, ограничивается телом функции: предыдущее значение параметра так же будет восстановлено при завершении функции. Однако обычная команда `SET` (без `LOCAL`) переопределяет предложение `SET`, как и предыдущую команду `SET LOCAL`: действие такой команды будет сохранено и после завершения функции, если только не произойдёт откат транзакции.

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

`определение`

Строковая константа, определяющая реализацию функции; её значение зависит от языка. Это может быть имя внутренней функции, путь к объектному файлу, команда SQL или код функции на процедурном языке.

Часто бывает полезно заключать определение функции в доллары (см. Подраздел 4.1.2.4), а не в традиционные апострофы. Если не использовать доллары, все апострофы и обратные косые черты в определении функции придётся экранировать, дублируя их.

объектный_файл, *объектный_символ*

Эта форма предложения `AS` применяется для динамически загружаемых функций на языке C, когда имя функции в коде C не совпадает с именем функции в SQL. Строка *объектный_файл* задаёт имя файла разделяемой библиотеки, содержащей скомпилированную функцию на C, и воспринимается как параметр команды `LOAD`. Строка *объектный_символ* задаёт символ скомпонованной функции, то есть имя функции в исходном коде на языке C. Если объектный символ опущен, предполагается, что он совпадает с именем определяемой SQL-функции. В C имена всех функций должны быть различными, поэтому перегружаемым функциям, реализованным на C, нужно давать разные имена (например, включать в имена C обозначения типов аргументов).

Если повторные вызовы `CREATE FUNCTION` ссылаются на один и тот же объектный файл, он загружается в рамках сеанса только один раз. Чтобы выгрузить и загрузить этот файл снова (например, в процессе разработки), начните новый сеанс.

тело_sql

Тело функции в стиле `LANGUAGE SQL`. Это может были один оператор

```
RETURN выражение
```

или блок операторов

```
BEGIN ATOMIC
  оператор;
  оператор;
  ...
  оператор;
END
```

Оно определяется подобно телу, задаваемому строковой константой (см. *определение* выше), но есть и некоторые различия. Эта форма работает только с функциями в стиле `LANGUAGE SQL`, тогда как форма со строковой константой поддерживается для всех языков. Она разбирается во время определения функции, тогда как форма со строковой константой — во время выполнения; как следствие, эта форма не поддерживает полиморфные типы аргументов и другие конструкции, которые нельзя обработать во время определения функции. С данной формой отслеживаются зависимости функции от объектов, используемых в её теле, так что команда `DROP ... CASCADE` выполнится корректно, тогда как в случае определения тела в строковой константе после такого удаления могут остаться неполноценные функции. Наконец, данная форма в большей степени соответствует стандарту SQL и совместима с другими реализациями SQL.

Перегрузка

Postgres Pro допускает *перегрузку* функций; то есть, позволяет использовать одно имя для нескольких различных функций, если у них различаются типы входных аргументов. Независимо от того, используете вы эту возможность или нет, она требует предосторожности при вызове функций в базах данных, где одни пользователи не доверяют другим; см. Раздел 10.3.

Две функции считаются совпадающими, если они имеют одинаковые имена и типы *входных* аргументов, параметры `OUT` игнорируются. Таким образом, например, эти объявления вызовут конфликт:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

Функции, имеющие разные типы аргументов, не будут считаться конфликтующими в момент создания, но предоставленные для них значения по умолчанию могут вызвать конфликт в момент использования. Например, рассмотрите следующие определения:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

Вызов `foo(10)` завершится ошибкой из-за неоднозначности в выборе вызываемой функции.

Примечания

В объявлении аргументов функции и возвращаемого значения допускается полный синтаксис описания типа SQL. Однако модификаторы типа в скобках (например, поле точности для типа `numeric`) команда `CREATE FUNCTION` не учитывает. Так что, например, `CREATE FUNCTION foo (varchar(10)) ...` создаст такую же функцию, что и `CREATE FUNCTION foo (varchar) ...`.

При замене существующей функции с помощью `CREATE OR REPLACE FUNCTION` есть ограничения на изменения имён параметров. В частности, нельзя изменить имя, уже назначенное любому входному параметру (хотя можно добавить имена ранее безымянным параметрам). Также, если у функции более одного выходного параметра, нельзя изменять имена выходных параметров, так как это приведёт к изменению имён столбцов анонимного составного типа, описывающего результат функции. Эти ограничения позволяют гарантировать, что существующие вызовы функции не перестанут работать после её замены.

Если функция объявлена как `STRICT` с аргументом `VARIADIC`, при оценивании строгости проверяется, что весь переменный массив *в целом* не NULL. Если же в этом массиве содержатся элементы NULL, функция будет вызываться.

Примеры

SQL-функция, складывающая два целых числа:

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Аналогичная функция, определённая в стиле, соответствующем стандарту SQL, с именами аргументов и без обрамляющих тело кавычек:

```
CREATE FUNCTION add(a integer, b integer) RETURNS integer
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT
    RETURN a + b;
```

Функция увеличения целого числа на 1, использующая именованный аргумент, на языке PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Функция, возвращающая запись с несколькими выходными параметрами:

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

То же самое можно сделать более развёрнуто, явно объявив составной тип:

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
```

```
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Ещё один способ вернуть несколько столбцов — применить функцию `TABLE`:

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Однако пример с `TABLE` отличается от предыдущих, так как в нём функция на самом деле возвращает не одну, а *набор* записей.

Разработка защищённых функций `SECURITY DEFINER`

Так как функция `SECURITY DEFINER` выполняется с правами пользователя, владеющего ей, необходимо позаботиться о том, чтобы её нельзя было использовать не по назначению. В целях безопасности в пути search_path следует исключить любые схемы, доступные на запись недоверенным пользователям. Это не позволит злонамеренным пользователям создать свои объекты (например, таблицы, функции и операторы), которые замаскируют объекты, используемые функцией. Особенно важно в этом отношении исключить схему временных таблиц, которая по умолчанию просматривается первой, а право записи в неё по умолчанию имеют все. Соответствующую защиту можно организовать, поместив временную схему в конец списка поиска. Для этого следует сделать `pg_temp`последней записью в `search_path`. Безопасное использование демонстрирует следующая функция:

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- Установить безопасный путь поиска: сначала доверенная схема(ы), затем 'pg_temp'.
    SET search_path = admin, pg_temp;
```

Эта функция должна обращаться к таблице `admin.pwds`, но без предложения `SET` или с предложением `SET`, включающим только `admin`, её можно «обмануть», создав временную таблицу `pwds`.

Если функция, определяющая контекст безопасности, должна создавать роли, и она вызывается не от имени суперпользователя, необходимо задать для параметра `createrole_self_grant` непустое значение, добавив предложение `SET`.

Также следует помнить о том, что по умолчанию право выполнения для создаваемых функций имеет роль `PUBLIC` (за подробностями обратитесь к Разделу 5.8). Однако часто требуется разрешить доступ к функциям, работающим в контексте определившего, только некоторым пользователям. Для этого необходимо отозвать стандартные права `PUBLIC` и затем дать права на выполнение индивидуально. Чтобы не образовалось окно, в котором новая функция будет доступна всем, создайте её и назначьте права в одной транзакции. Например, так:

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
```

```
COMMIT;
```

Совместимость

Команда `CREATE FUNCTION` определена в стандарте SQL. Её реализация в Postgres Pro может использоваться совместимым образом, но дополнена множеством расширений. И напротив, в стандарте SQL определён ряд дополнительных вещей, не реализованных в Postgres Pro.

Обратите внимание на важные замечания о совместимости:

- Указание `OR REPLACE` является расширением Postgres Pro.
- Для совместимости с другими СУБД *`режим_аргумента`* можно записать после *`имя_аргумента`* или перед ним, но стандарту соответствует только первый вариант.
- Для определения значений по умолчанию для параметров стандарт SQL поддерживает только синтаксис с ключевым словом `DEFAULT`. Синтаксис со знаком `=` используется в T-SQL и Firebird.
- Указание `SETOF` является расширением Postgres Pro.
- В стандарте определён только один язык — `SQL`.
- Все атрибуты, кроме `CALLED ON NULL INPUT` и `RETURNS NULL ON NULL INPUT`, относятся к расширением стандарта.
- Тело функции в стиле `LANGUAGE SQL`, согласно стандарту SQL, может определяться только в форме, задающей *`тело_sql`*.

Простые функции в стиле `LANGUAGE SQL` можно определить так, чтобы их определение соответствовало стандарту и было совместимым с другими реализациями. Более сложные функции, использующие расширенные возможности, атрибуты оптимизации, или написанные на других языках, по понятным причинам должны определяться в специальном синтаксисе Postgres Pro.

См. также

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE

CREATE GROUP

CREATE GROUP — создать роль в базе данных

Синтаксис

```
CREATE GROUP имя [ [ WITH ] параметр [ ... ] ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль' | PASSWORD NULL
    | VALID UNTIL 'дата_время'
    | IN ROLE имя_роли [, ...]
    | IN GROUP имя_роли [, ...]
    | ROLE имя_роли [, ...]
    | ADMIN имя_роли [, ...]
    | USER имя_роли [, ...]
    | SYSID uid
```

Описание

Оператор `CREATE GROUP` теперь является синонимом оператора CREATE ROLE.

Совместимость

Оператор `CREATE GROUP` отсутствует в стандарте SQL.

См. также

CREATE ROLE

CREATE INDEX

CREATE INDEX — создать индекс

Синтаксис

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] имя ] ON
 [ ONLY ] имя_таблицы [ USING метод ]
    ( { имя_столбца | ( выражение ) } [ COLLATE правило_сортировки ] [ класс_операторов
 [ ( параметр_класса_оп = значение [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST |
 LAST } ] [, ...] )
    [ INCLUDE ( имя_столбца [, ...] ) ]
    [ NULLS [ NOT ] DISTINCT ]
    [ WITH ( параметр_хранения [= значение] [, ... ] ) ]
    [ TABLESPACE табл_пространство ]
    [ WHERE предикат ]
```

Описание

`CREATE INDEX` создаёт индексы по указанному столбцу(ам) заданного отношения, которым может быть таблица или материализованное представление. Индексы применяются в первую очередь для оптимизации производительности базы данных (хотя при неправильном использовании возможен и противоположный эффект).

Ключевое поле для индекса задаётся как имя столбца или выражение, заключённое в скобки. Если метод индекса поддерживает составные индексы, допускается указание нескольких полей.

Поле индекса может быть выражением, вычисляемым из значений одного или нескольких столбцов в строке таблицы. Это может быть полезно для получения быстрого доступа к данным по некоторому преобразованию исходных значений. Например, индекс, построенный по выражению `upper(col)`, позволит использовать поиск по индексу в предложении `WHERE upper(col) = 'JIM'`.

Postgres Pro предоставляет следующие методы индексов: B-дерево, хеш, GiST, SP-GiST, GIN, RUM и BRIN. Пользователи могут определить и собственные методы индексов, но это довольно сложная задача.

Если в команде присутствует предложение `WHERE`, она создаёт *частичный индекс*. Такой индекс содержит записи только для части таблицы, обычно более полезной для индексации, чем остальная таблица. Например, если таблица содержит информацию об оплаченных и неоплаченных счетах, при этом последних сравнительно немного, но именно эта часть таблицы наиболее востребована, то увеличить быстродействие можно, создав индекс только по этой части. Ещё одно возможное применение `WHERE` — добавив `UNIQUE`, обеспечить уникальность в подмножестве таблицы. Подробнее это рассматривается в Разделе 11.8.

Выражение в предложении `WHERE` может ссылаться только на столбцы нижележащей таблицы, но не обязательно ограничиваться теми, по которым строится индекс. В настоящее время в `WHERE` также нельзя использовать подзапросы и агрегатные выражения. Это же ограничение распространяется и на выражения в полях индексов.

Все функции и операторы, используемые в определении индекса, должны быть «постоянными», то есть, их результаты должны зависеть только от аргументов, но не от внешних факторов (например, содержимого другой таблицы или текущего времени). Это ограничение обеспечивает определённость поведения индекса. Чтобы использовать в выражении индекса или в предложении `WHERE` собственную функцию, не забудьте пометить её при создании как постоянную (IMMUTABLE).

Параметры

`UNIQUE`

Указывает, что система должна контролировать повторяющиеся значения в таблице при создании индекса (если в таблице уже есть данные) и при каждом добавлении данных. Попытки вставить или изменить данные, при которых будет нарушена уникальность индекса, будут завершаться ошибкой.

Когда уникальные индексы применяются к секционированным таблицам, действуют дополнительные ограничения; см. CREATE TABLE.

`CONCURRENTLY`

С этим указанием Postgres Pro построит индекс, не устанавливая никаких блокировок, которые бы предотвращали добавление, изменение или удаление записей в таблице, тогда как по умолчанию операция построения индекса блокирует запись (но не чтение) данных в таблице до своего завершения. С созданием индекса в этом режиме связан ряд особенностей, о которых следует знать, — см. Неблокирующее построение индексов.

Для временных таблиц `CREATE INDEX` всегда выполняется более простым, неблокирующим способом, так как они не могут использоваться никакими другими сеансами.

`IF NOT EXISTS`

Не считать ошибкой, если индекс с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующий индекс как-то соотносится с тем, который мог бы быть создан. Имя индекса является обязательным, когда указывается `IF NOT EXISTS`.

`INCLUDE`

Необязательное предложение `INCLUDE` позволяет указать список столбцов, которые войдут в индекс как *неключевые* столбцы. Неключевой столбец не может использоваться в условии поиска для сканирования по индексу, и он не учитывается при анализе ограничения уникальности или исключения, устанавливаемого индексом. Однако при сканировании только индекса содержимое неключевых столбцов может быть получено без обращения к целевой таблице, так как оно находится непосредственно в элементе индекса. Таким образом, в результате добавления неключевых столбцов сканирование только индекса может использоваться в тех запросах, где иначе оно было бы неприменимо.

Добавляя в индекс неключевые столбцы, особенно большого размера, есть смысл проявлять консерватизм. Если кортеж в индексе превышает максимально допустимый размер для данного типа индексов, вставить данные в таблицу не удастся. В неключевых столбцах дублируются данные из основной таблицы, что приводит к раздуванию индекса и может повлечь замедление запросов. Кроме того, если неключевой столбец содержится в индексе B-дерево, в таком индексе не будет работать исключение дубликатов.

Для столбцов, перечисленных в `INCLUDE`, не нужны соответствующие классы операторов; это предложение может содержать столбцы, для типов данных которых не определены классы операторов для заданного метода доступа.

Выражения в неключевых столбцах не поддерживаются, так как их нельзя будет использовать при сканировании только индекса.

В настоящее время эту возможность поддерживают методы доступа индексов B-дерево, GiST и SP-GiST. В таких индексах значения столбцов, указанных в предложении `INCLUDE`, включаются в кортежи на уровне листьев, которые соответствуют кортежам кучи, но не включаются в элементы верхних уровней, используемые для навигации в дереве.

`имя`

Имя создаваемого индекса. Указание схемы при этом не допускается; индекс всегда относится к той же схеме, что и родительская таблица. Имя индекса должно отличаться от имени любого другого отношения (таблицы, последовательности, индекса, представления, материализованного представления или сторонней таблицы) в этой схеме. Если имя опущено, Postgres Pro формирует подходящее имя по имени родительской таблицы и именам индексируемых столбцов.

`ONLY`

Указывает, что индексы не должны рекурсивно создаваться в секциях секционированной таблицы. По умолчанию создание выполняется рекурсивно.

`имя_таблицы`

Имя индексируемой таблицы (возможно, дополненное схемой).

`метод`

Имя применяемого метода индекса. Возможные варианты: `btree`, `hash`, `gist`, `spgist`, `gin`, `brin` или установленный пользователем метод, например bloom. По умолчанию подразумевается метод `btree`.

`имя_столбца`

Имя столбца таблицы.

`выражение`

Выражение с одним или несколькими столбцами таблицы. Обычно выражение должно записываться в скобках, как показано в синтаксисе команды. Однако скобки можно опустить, если выражение записано в виде вызова функции.

`правило_сортировки`

Имя правила сортировки, применяемого для индекса. По умолчанию используется правило сортировки, заданное для индексируемого столбца, либо полученное для результата выражения индекса. Индексы с нестандартными правилами сортировки могут быть полезны для запросов, включающих выражения с такими правилами.

`класс_операторов`

Имя класса операторов. Подробнее об этом ниже.

`параметр_класса_оп`

Имя параметра класса операторов. Подробнее об этом ниже.

`ASC`

Указывает порядок сортировки по возрастанию (подразумевается по умолчанию).

`DESC`

Указывает порядок сортировки по убыванию.

`NULLS FIRST`

Указывает, что значения NULL после сортировки оказываются перед остальными. Это поведение по умолчанию с порядком сортировки `DESC`.

`NULLS LAST`

Указывает, что значения NULL после сортировки оказываются после остальных. Это поведение по умолчанию с порядком сортировки `ASC`.

`NULLS DISTINCT`
`NULLS NOT DISTINCT`

Определяет, должны ли значения NULL считаться различными (не равными) для уникального индекса. По умолчанию они считаются различными, поэтому уникальный индекс может содержать в столбце несколько значений NULL.

параметр_хранения

Имя специфичного для индекса параметра хранения. За подробностями обратитесь к разделу Параметры хранения индекса ниже.

табл_пространство

Табличное пространство, в котором будет создан индекс. Если не определено, выбирается default_tablespace, либо temp_tablespaces, при создании индекса временной таблицы.

предикат

Выражение ограничения для частичного индекса.

Параметры хранения индекса

Необязательное предложение `WITH` определяет *параметры хранения* для индекса. У каждого метода индекса есть свой набор допустимых параметров хранения. Следующий параметр принимают методы B-дерево, хеш, GiST и SP-GiST:

`fillfactor` (`integer`)

Фактор заполнения для индекса определяет в процентном отношении, насколько плотно метод индекса будет заполнять страницы индекса. Для B-деревьев концевые страницы заполняются до этого процента при начальном построении индекса и позже, при расширении индекса вправо (добавлении новых наибольших значений ключа). Если страницы впоследствии оказываются заполненными полностью, они будут разделены, что приводит к фрагментации структуры индекса на диске. Для B-деревьев по умолчанию используется фактор заполнения 90, но его можно поменять на любое целое число от 10 до 100.

Для индексов-B-деревьев в таблицах, где ожидается множество добавлений и/или изменений записей, имеет смысл выбрать меньшие значения фактора заполнения в момент выполнения `CREATE INDEX` (до массовой загрузки данных в таблицы). Значения в диапазоне 50 – 90 могут сгладить *всплески* разделений страниц на начальном этапе жизни индекса-B-дерева (при подобном снижении фактора заполнения может даже уменьшиться абсолютное количество разделений страниц, хотя это во многом зависит от нагрузки). Процедура восходящего удаления индексных кортежей, описанная в Подразделе 62.1.4.2, рассчитывает на наличие некоторого «дополнительного» места на страницах, где будут сохраняться «дополнительные» версии кортежей, и следовательно, может зависеть от фактора заполнения (хотя обычно его влияние второстепенно).

В других особых случаях может быть полезно увеличить фактор заполнения до 100 перед выполнением `CREATE INDEX`, чтобы максимально эффективно использовать хранилище. Этот вариант следует рассматривать, только если вы абсолютно уверены, что таблица будет статичной (то есть в ней никогда не будут изменяться или добавляться данные). В противном случае фактор заполнения, равный 100, скорее *повредит* производительности — даже единичные изменения или добавления вызовут резкий скачок активности разделения страниц.

Другие индексные методы используют фактор заполнения по-своему, но примерно в том же ключе; значение по умолчанию для разных методов разное.

Индексы B-дерево дополнительно принимают этот параметр:

`deduplicate_items` (`boolean`)

Этот параметр управляет механизмом исключения дубликатов, описанным в Подразделе 62.1.4.3. Он принимает логическое значение `ON` или `OFF`, соответственно включающее или отключающее это оптимизацию. (Другие возможные написания `ON` и `OFF` перечислены в Разделе 18.1.) Значение по умолчанию — `ON`.

Примечание

После выключения параметра `deduplicate_items` командой `ALTER INDEX` при добавлении в будущем новых элементов дубликаты исключаться не будут, но представление существующих кортежей не поменяется на стандартное.

Индексы GiST дополнительно принимают этот параметр:

`buffering` (enum)

Определяет, будет ли при построении индекса использоваться буферизация, описанная в Подразделе 62.2.4.1. Со значением `OFF` она отключена, с `ON` — включена, а с `AUTO` — отключена вначале, но может затем включиться на ходу, как только размер индекса достигнет значения effective_cache_size. Значение по умолчанию — `AUTO`. Заметьте, что если возможно построить индекс с сортировкой, именно этот метод будет использоваться при любом значении, кроме `buffering=ON`.

Индексы GIN принимают другие параметры:

`fastupdate` (boolean)

Этот параметр управляет механизмом быстрого обновления, описанным в Подразделе 62.4.4.1. Он имеет логическое значение: `ON` включает быстрое обновление, `OFF` отключает его. Значение по умолчанию — `ON`.

Примечание

Выключение `fastupdate` в `ALTER INDEX` предотвращает помещение добавляемых в дальнейшем строк в список записей, ожидающих индексации, но записи, добавленные в этот список ранее, в нём остаются. Чтобы очистить очередь операций, надо затем выполнить `VACUUM` для этой таблицы или вызвать функцию `gin_clean_pending_list`.

`gin_pending_list_limit` (integer)

Пользовательский параметр gin_pending_list_limit. Его значение задаётся в килобайтах.

Индексы RUM принимают другие параметры:

`attach` (text)

Этот параметр задаёт имя столбца, который нужно присоединить в качестве дополнительной информации.

`to` (text)

Этот параметр задаёт имя столбца, по которому добавляется упорядочивание.

`order_by_attach` (boolean)

Этот параметр задаёт упорядочение по `attach` и `itempointer` вместо упорядочения только по `itempointer`.

Индексы BRIN принимают другие параметры:

`pages_per_range` (integer)

Определяет, сколько блоков таблицы образуют зону блоков для каждой записи в индексе BRIN (за подробностями обратитесь к Подразделу 62.5.1). Значение по умолчанию — `128`.

`autosummarize` (`boolean`)

Определяет, будет ли поставлен в очередь расчёт сводного значения для предыдущей зоны страниц, когда происходит добавление в следующую. За подробностями обратитесь к Подразделу 62.5.1.1. По умолчанию `off`.

Неблокирующее построение индексов

Создание индекса может мешать обычной работе с базой данных. Обычно Postgres Pro блокирует запись в индексируемую таблицу и выполняет всю операцию построения индекса за одно сканирование таблицы. Другие транзакции могут продолжать читать таблицу, но при попытке вставить, изменить или удалить строки в таблице они будут заблокированы до завершения построения индекса. Это может оказать нежелательное влияние на работу производственной базы данных. Индексация очень больших таблиц может занимать много часов, и даже для маленьких таблиц построение индекса может заблокировать записывающие процессы на время, неприемлемое для производственной системы.

Postgres Pro поддерживает построение индексов без блокировки записи. Этот метод выбирается указанием `CONCURRENTLY` команды `CREATE INDEX`. Когда он используется, Postgres Pro должен выполнить два сканирования таблицы, а кроме того, должен дождаться завершения всех существующих транзакций, которые потенциально могут модифицировать и использовать этот индекс. Таким образом, эта процедура требует проделать в сумме больше действий и выполняется значительно дольше, чем обычное построение индекса. Однако благодаря тому, что этот метод позволяет продолжать обычную работу с базой во время построения индекса, он оказывается полезным в производственной среде. Хотя разумеется, дополнительная нагрузка на процессор и подсистему ввода-вывода, создаваемая при построении индекса, может привести к замедлению других операций.

При неблокирующем построении индекса он как «нерабочий» добавляется в системный каталог в одной транзакции, а затем в двух других транзакциях выполняются ещё два сканирования таблицы. Перед каждым сканированием таблицы процедура построения индекса должна ждать завершения текущих транзакций, модифицировавших эту таблицу. После второго сканирования также необходимо дожидаться завершения всех транзакций, получивших снимок (см. Главу 13) перед вторым сканированием, включая транзакции, задействованные на любом этапе неблокирующего построения индексов для других таблиц (это касается частичных индексов и индексов, построенных не просто по столбцам). Наконец индекс может быть помечен как «рабочий» и готовый к использованию, после чего команда `CREATE INDEX` завершается. Однако даже тогда индекс может быть не готов немедленно к применению в запросах: в худшем случае он не будет использоваться, пока существуют транзакции, начатые до начала построения индекса.

Если при сканировании таблицы возникает проблема, например взаимоблокировка или нарушение уникальности в уникальном индексе, команда `CREATE INDEX` завершится ошибкой, но оставит после себя «нерабочий» индекс. Этот индекс будет игнорироваться при чтении данных, так как он может быть неполным; однако с ним могут быть связаны дополнительные операции при изменениях. В psql встроенная команда `\d` помечает такой индекс как `INVALID`:

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
Indexes:
    "idx" btree (col) INVALID
```

Рекомендуемый в таких случаях способ исправления ситуации — удалить индекс и затем попытаться снова выполнить `CREATE INDEX CONCURRENTLY`. (Кроме того, можно перестроить его с помощью команды `REINDEX INDEX CONCURRENTLY`.)

Ещё одна сложность, с которой можно столкнуться при неблокирующем построении уникального индекса, заключается в том, что ограничение уникальности уже влияет на другие транзакции, когда второе сканирование таблицы только начинается. Это значит, что нарушения ограничения могут проявляться в других запросах до того, как индекс становится доступным для использования и даже тогда, когда создать индекс в итоге не удаётся. Кроме того, если при втором сканировании происходит ошибка, «нерабочий» индекс оставляет в силе своё ограничение уникальности и дальше.

Метод неблокирующего построения поддерживает также индексы выражений и частичные индексы. Ошибки, произошедшие при вычислении этих выражений, могут привести к такому же поведению, как в вышеописанных случаях с нарушением ограничений уникальности.

Обычное построение индекса допускает одновременное построение других индексов для таблицы обычным методом, но неблокирующее построение для конкретной таблицы в один момент времени допускается только одно. Однако в любом случае никакие другие изменения схемы таблицы в это время не разрешаются. Другое отличие состоит в том, что в блоке транзакции может быть выполнена обычная команда `CREATE INDEX`, но не `CREATE INDEX CONCURRENTLY`.

Метод неблокирующего построения для индексов секционированных таблиц в настоящее время не поддерживается. Однако вы можете сократить время, на которое будет заблокирована секционированная таблица, построив в неблокирующем режиме индекс для каждой секции в отдельности и затем создав секционированный индекс в обычном режиме. В этом случае построение секционированного индекса будет заключаться только в изменении метаданных.

Примечания

Информацию о том, когда могут применяться, и когда не применяются индексы, и в каких конкретных ситуациях они могут быть полезны, можно найти в Главе 11.

В настоящее время индексы с несколькими ключевыми столбцами поддерживаются только методами B-дерево, GiST, GIN и BRIN. Возможность построения индекса по нескольким ключевым столбцам не зависит от поддержки включения в индекс неключевых столбцов (`INCLUDE`). Индексы могут содержать до 32 столбцов, в том числе неключевых. (Этот предел можно изменить, пересобрав Postgres Pro.) Уникальные индексы поддерживает только метод B-дерево.

Для каждого столбца индекса можно задать *класс операторов* и дополнительные параметры класса. Данный класс определяет, какие операторы будут использоваться индексом для этого столбца. Например, индекс-B-дерево по четырёхбайтовым целым будет использовать класс `int4_ops`; этот класс операторов включает функции сравнения для таких значений. На практике обычно достаточно использовать класс операторов по умолчанию для типа данных столбца. Существование классов операторов объясняется в первую очередь тем, что для некоторых типов данных можно предложить более одного осмысленного порядка сортировки. Например, может возникнуть желание отсортировать комплексные числа как по абсолютному значению, так и по вещественной части. Это можно сделать, определив два класса операторов для типа данных и выбрав подходящий класс при создании индекса. За дополнительными сведениями о классах операторов обратитесь к Разделу 11.10 и Разделу 36.16.

Когда команда `CREATE INDEX` вызывается для секционированной таблицы, по умолчанию её действие распространятся рекурсивно на все секции, с тем чтобы в них оказались соответствующие индексы. Сначала каждая секция проверяется на наличие равнозначного индекса, и, если таковой находится, он присоединяется как индекс секции к создаваемому, который таким образом становится родительским индексом. Если нужного индекса не оказывается, новый индекс автоматически создаётся и присоединяется к основному; имя индекса для каждой секции выбирается так же, как и при выполнении этой команды без имени индекса. С указанием `ONLY` рекурсия не производится и индекс помечается как нерабочий. (Команда `ALTER INDEX ... ATTACH PARTITION` пометит его как рабочий, когда он будет представлен во всех секциях). Однако заметьте, что в любой секции, создаваемой в будущем командой `CREATE TABLE ... PARTITION OF`, соответствующий индекс появится автоматически, вне зависимости от данного указания.

Для методов индекса, поддерживающих сканирование по порядку (в настоящее время это поддерживает только B-дерево), можно изменить порядок сортировки индекса, добавив необязательные предложения `ASC`, `DESC`, `NULLS FIRST` или `NULLS LAST`. Так как упорядоченный индекс можно сканировать вперёд или назад, обычно не имеет смысла создавать индекс по убыванию (`DESC`) для одного столбца — этот порядок сортировки можно получить и с обычным индексом. Эти параметры имеют смысл при создании составных индексов так, что они будут соответствовать порядку сортировки, указанному в запросе со смешанным порядком, например `SELECT ... ORDER BY x ASC, y DESC`. Параметры `NULLS` полезны, когда требуется реализовать поведение «NULL внизу», изменив стандартное «NULL вверху», в запросах, зависящих от индексов, чтобы избежать дополнительной сортировки.

Система регулярно собирает статистику по всем столбцам таблицы. Новые индексы, построенные без применения выражений, могут эффективно использовать эту статистику сразу. Однако для создаваемых индексов по выражениям требуется выполнить `ANALYZE` или подождать, пока фоновый процесс автоочистки не проанализирует таблицу и не посчитает статистику для этих индексов.

На время выполнения команды `CREATE INDEX` search_path временно меняется на `pg_catalog, pg_temp`.

Для большинства методов индексов скорость создания индекса зависит от значения maintenance_work_mem. Чем больше это значение, тем меньше времени требуется для создания индекса (если только заданное значение не превышает объём действительно доступной памяти, что влечёт за собой использование подкачки).

Postgres Pro может строить индексы, задействуя несколько процессоров для ускорения обработки строк таблицы. Это называется *параллельным построением индексов*. Для методов индексов, поддерживающих построение в параллельном режиме (в настоящее время это B-дерево и BRIN), параметр `maintenance_work_mem` задаёт максимальный объём памяти, который может быть выделен для одной операции построения индекса в целом, независимо от того, сколько рабочих процессов будет запущено. Целесообразность использования параллельных процессов и их оптимальное количество обычно автоматически определяется моделью стоимости.

Параллельное построение индексов может выиграть от увеличения `maintenance_work_mem` там, где для аналогичного последовательного построения индекса выигрыша не будет или он будет минимальным. Заметьте, что значение `maintenance_work_mem` может влиять на число запрашиваемых рабочих процессов, так как параллельным исполнителям должно быть выделено не менее `32MB` из общего бюджета `maintenance_work_mem`. Кроме того, `32MB` должно остаться для ведущего процесса. Увеличение max_parallel_maintenance_workers позволит создать больше исполнителей, что приведёт к уменьшению времени создания индекса, если только создание индекса уже не упирается в скорость ввода-вывода. Разумеется, для этого должно быть достаточно процессорных ресурсов, которые иначе бы простаивали.

Если в `ALTER TABLE` задаётся значение `parallel_workers`, именно оно определяет, сколько параллельных исполнителей будет запрашивать команда `CREATE INDEX` для данной таблицы. При этом полностью игнорируется модель стоимости и `maintenance_work_mem` не влияет на определение количества параллельных исполнителей. Если параметру `parallel_workers` в `ALTER TABLE` присваивается 0, параллельное построение индексов для этой таблицы полностью отключается.

Подсказка

После изменения параметра `parallel_workers` в ходе оптимизации построения индексов имеет смысл сбросить его. Это предотвратит нежелательные изменения планов запросов, так как `parallel_workers` влияет на *все* параллельные сканирования таблицы.

Хотя `CREATE INDEX` с указанием `CONCURRENTLY` поддерживает параллельное построение без особых ограничений, фактически в параллельном режиме выполняется только первое сканирование таблицы.

Для удаления индекса применяется `DROP INDEX`.

Как и любая длительная транзакция, операция `CREATE INDEX` с таблицей может повлиять на возможность удаления кортежей параллельной операцией `VACUUM` с какой-либо другой таблицей.

В предыдущих выпусках Postgres Pro также поддерживался метод индекса R-дерево. Сейчас он отсутствует, так как он не даёт значительных преимуществ по сравнению с GiST. Указание `USING rtree` команда `CREATE INDEX` будет интерпретировать как `USING gist`, для упрощения перевода старых баз на GiST.

Каждый процесс, выполняющий операцию `CREATE INDEX`, будет выдавать информацию о ходе её выполнения, отображаемую в представлении `pg_stat_progress_create_index`. За подробностями обратитесь к Подразделу 26.4.4.

Примеры

Создание уникального индекса-B-дерева по столбцу `title` в таблице `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

Создание уникального индекса-B-дерева по столбцу `title` с неключевыми столбцами `director` и `rating` в таблице `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);
```

Создание индекса B-дерево без исключения дубликатов:

```
CREATE INDEX title_idx ON films (title) WITH (deduplicate_items = off);
```

Создание индекса по выражению `lower(title)`, позволяющего эффективно выполнять регистронезависимый поиск:

```
CREATE INDEX ON films ((lower(title)));
```

(В этом примере мы решили опустить имя индекса, чтобы имя выбрала система, например `films_lower_idx`.)

Создание индекса с нестандартным правилом сортировки:

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

Создание индекса с нестандартным порядком значений NULL:

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

Создание индекса с нестандартным фактором заполнения:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

Создание индекса GIN с отключённым механизмом быстрого обновления:

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);
```

Создание индекса по столбцу `code` в таблице `films` и размещение его в табличном пространстве `indexspace`:

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

Создание индекса GiST по координатам точек, позволяющего эффективно использовать операторы box с результатом функции преобразования:

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

Создание индекса без блокировки записи в таблицу:

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

Совместимость

`CREATE INDEX` является языковым расширением Postgres Pro. Средства обеспечения индексов в стандарте SQL не описаны.

См. также

ALTER INDEX, DROP INDEX, REINDEX, Подраздел 26.4.4

CREATE LANGUAGE

CREATE LANGUAGE — создать процедурный язык

Синтаксис

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE имя
    HANDLER обработчик_вызова [ INLINE обработчик_внедрённого_кода ]
 [ VALIDATOR функция_проверки ]
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE имя
```

Описание

`CREATE LANGUAGE` регистрирует в базе данных Postgres Pro новый процедурный язык. Впоследствии на этом языке можно будет определять новые функции и процедуры.

`CREATE LANGUAGE` по сути связывает имя языка с функциями-обработчиками, исполняющими код, написанный на этом языке. За дополнительными сведениями о языковых обработчиках обратитесь к Главе 54.

`CREATE OR REPLACE LANGUAGE` создаёт новый язык или заменяет определение существующего. Если язык существует, его параметры изменяются в соответствии с командой, но владелец языка и права доступа к нему не меняются. Все уже написанные на этом языке функции также считаются корректными.

Для того, чтобы зарегистрировать новый язык или изменить параметры существующего, необходимо быть суперпользователем Postgres Pro. Однако после того, как язык создан, владельцем его можно назначить обычного пользователя, который затем сможет удалить или переименовать его, изменить права доступа к нему, или назначить другого владельца. (Однако нельзя назначать обычного пользователя владельцем нижележащих функций на C — это даст ему возможность повысить свои привилегии.)

Форма `CREATE LANGUAGE`, в которой не задаётся никакой обработчик, является устаревшей. Для обратной совместимости со старыми копиями в формате скрипта она воспринимается как `CREATE EXTENSION` и будет работать, если язык упакован в расширение с тем же именем, как обычно и оформляются процедурные языки.

Параметры

`TRUSTED`

Указание `TRUSTED` (доверенный) означает, что язык не даёт пользователю доступ к данным, к которым он не имел бы доступа без него. Если при регистрации языка это ключевое слово опущено, использовать этот язык для создания новых функций смогут только суперпользователи Postgres Pro.

`PROCEDURAL`

Это слово не несёт смысловой нагрузки.

`имя`

Имя процедурного языка. Оно должно быть уникальным среди всех языков в базе данных.

`HANDLER` *`обработчик_вызова`*

Здесь *`обработчик_вызова`* — имя ранее зарегистрированной функции, которая будет вызываться для исполнения процедур (функций) на этом языке. Обработчик вызова для процедурного языка должен быть написан на компилируемом языке, например, на C, с соглашениями о вызовах версии 1 и зарегистрирован в Postgres Pro как функция без аргументов, возвращающая фиктивный тип `language_handler`, который просто показывает, что эта функция — обработчик вызова.

`INLINE` *`обработчик_внедрённого_кода`*

Здесь *`обработчик_внедрённого_кода`* — имя ранее зарегистрированной функции, которая будет вызываться для выполнения анонимного блока кода (команды `DO`) на этом языке. Если *`обработчик_внедрённого_кода`* не определён, данный язык не будет поддерживать анонимные блоки кода. Этот обработчик должен принимать один аргумент типа `internal`, содержащий внутреннее представление команды `DO`, и обычно возвращает тип `void`. Значение, возвращаемое этим обработчиком, игнорируется.

`VALIDATOR` *`функция_проверки`*

Здесь *`функция_проверки`* — имя ранее зарегистрированной функции, которая будет вызываться при создании новой функции на этом языке и проверять её. Если функция проверки не задана, функции на этом языке при создании проверяться не будут. Функция проверки должна принимать один аргумент типа `oid` с идентификатором создаваемой функции и обычно возвращает `void`.

Функция проверки обычно проверяет тело создаваемой функции на синтаксические ошибки, но может также проанализировать и другие характеристики функции, например, поддержку определённых типов аргументов этим языком. Значение, возвращаемое этой функцией, игнорируется, об ошибках она должна сигнализировать, вызывая `ereport()`.

Примечания

Для удаления процедурных языков следует использовать `DROP LANGUAGE`.

В системном каталоге `pg_language` (см. Раздел 51.29) записывается информация о языках, установленных в данный момент. Список установленных языков также показывает команда `\dL` в psql.

Чтобы создавать функции на процедурном языке, пользователь должен иметь право `USAGE` для этого языка. По умолчанию для доверенных языков право `USAGE` даётся роли `PUBLIC` (т. е. всем), однако при желании его можно отозвать.

Процедурные языки являются локальными по отношению к базам данных. Тем не менее, язык можно установить в базу данных `template1`, в результате чего он будет автоматически доступен во всех базах данных, создаваемых из неё впоследствии.

Примеры

Минимальная процедура создания нового процедурного языка выглядит так:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Обычно эти команды записываются в скрипте установки расширения, которое пользователи затем устанавливают так:

```
CREATE EXTENSION plsample;
```

Совместимость

`CREATE LANGUAGE` является расширением Postgres Pro.

См. также

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE

CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW — создать материализованное представление

Синтаксис

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] имя_таблицы
    [ (имя_столбца [, ...] ) ]
    [ USING метод ]
    [ WITH ( параметр_хранения [= значение] [, ... ] ) ]
    [ TABLESPACE табл_пространство ]
    AS query
    [ WITH [ NO ] DATA ]
```

Описание

`CREATE MATERIALIZED VIEW` определяет материализованное представление запроса. Заданный запрос выполняется и наполняет представление в момент вызова команды (если только не указано `WITH NO DATA`). Обновить представление позже можно, выполнив `REFRESH MATERIALIZED VIEW`.

Команда `CREATE MATERIALIZED VIEW` подобна `CREATE TABLE AS`, за исключением того, что она запоминает запрос, порождающий представление, так что это представление можно позже обновить по требованию. Материализованные представления сходны с таблицами во многом, но не во всём; например, не поддерживаются временные материализованные представления.

Команда `CREATE MATERIALIZED VIEW` требует наличия права `CREATE` для схемы, выбранной для создания материализованного представления.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если материализованное представление с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее материализованное представление как-то соотносится с тем, которое могло бы быть создано.

`имя_таблицы`

Имя (возможно, дополненное схемой) создаваемого материализованного представления. Это имя должно отличаться от имени любого другого отношения (таблицы, последовательности, индекса, представления, материализованного представления или сторонней таблицы) в той же схеме.

`имя_столбца`

Имя столбца в создаваемом материализованном представлении. Если имена столбцов не заданы явно, они определяются по именам столбцов результата запроса.

`USING` *`метод`*

Это дополнительное предложение задаёт табличный метод доступа, который будет использоваться для сохранения содержимого нового материализованного представления; этот метод должен быть типа `TABLE`. Подробнее об этом рассказывается в Главе 59. В случае отсутствия этого указания для нового материализованного представления выбирается метод доступа по умолчанию. За подробностями обратитесь к default_table_access_method.

`WITH (` *`параметр_хранения`* `[=` *`значение`*`] [, ... ] )`

Это предложение задаёт дополнительные параметры хранения для создаваемого материализованного представления; подробнее о них можно узнать в разделе Параметры хранения описания CREATE TABLE. Все параметры, которые поддерживает `CREATE TABLE`, поддерживает и `CREATE MATERIALIZED VIEW`. За дополнительной информацией обратитесь к CREATE TABLE.

`TABLESPACE` *`табл_пространство`*

Здесь *`табл_пространство`* — имя табличного пространства, в котором будет создано материализованное представление. Если оно не задано, выбирается default_tablespace.

`query`

Команда `SELECT`, `TABLE` или `VALUES`. Эта команда будет выполняться с ограничениями по безопасности; в частности, будут запрещены вызовы функций, которые сами создают временные таблицы. Также во время выполнения команды search_path временно меняется на `pg_catalog`, `pg_temp`.

`WITH [ NO ] DATA`

Это предложение указывает, будет ли материализованное представление наполняться в момент создания. Если материализованное представление не наполняется, оно помечается как нечитаемое, так что к нему нельзя будет обращаться до выполнения `REFRESH MATERIALIZED VIEW`.

Совместимость

`CREATE MATERIALIZED VIEW` является расширением Postgres Pro.

См. также

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

CREATE OPERATOR

CREATE OPERATOR — создать оператор

Синтаксис

```
CREATE OPERATOR имя (
    {FUNCTION|PROCEDURE} = имя_функции
    [, LEFTARG = тип_слева ] [, RIGHTARG = тип_справа ]
    [, COMMUTATOR = коммут_оператор ] [, NEGATOR = обратный_оператор ]
    [, RESTRICT = процедура_ограничения ] [, JOIN = процедура_соединения ]
    [, HASHES ] [, MERGES ]
)
```

Описание

`CREATE OPERATOR` определяет новый оператор, *имя*. Владельцем оператора становится пользователь, его создавший. Если указано имя схемы, оператор создаётся в ней, в противном случае — в текущей схеме.

Имя оператора образует последовательность из нескольких символов (не более чем `NAMEDATALEN`-1, по умолчанию 63) из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Однако выбор имени ограничен ещё следующими условиями:

- Сочетания символов `--` и `/*` не могут присутствовать в имени оператора, так как они будут обозначать начало комментария.
- Многосимвольное имя оператора не может заканчиваться знаком `+` или `-`, если только оно не содержит также один из этих символов:

  ~ ! @ # % ^ & | ` ?

  Например, `@-` — допустимое имя оператора, а `*-` — нет. Благодаря этому ограничению, Postgres Pro может разбирать корректные SQL-запросы без пробелов между компонентами.
- Символ => зарезервирован для использования в грамматике языка SQL, поэтому его нельзя использовать в качестве имени оператора.

Оператор `!=` отображается в `<>` при вводе, так что эти два имени всегда равнозначны.

Для бинарных операторов должны определяться оба параметра `LEFTARG` и `RIGHTARG`, а для префиксных операторов нужно определить только параметр `RIGHTARG`. Функция *имя_функции* должна быть уже определена с помощью `CREATE FUNCTION` и иметь соответствующее число аргументов (один или два) указанных типов.

В синтаксисе `CREATE OPERATOR` ключевые слова `FUNCTION` и `PROCEDURE` равнозначны, но указываемая функция должна в любом случае быть функцией, а не процедурой. Ключевое слово `PROCEDURE` здесь поддерживается по историческим причинам и считается устаревшим.

Другие предложения определяют дополнительные характеристики оптимизации. Их значение описано в Разделе 36.15.

Чтобы создать оператор, необходимо иметь право `USAGE` для типов аргументов и результата, а также право `EXECUTE` для нижележащей функции. Если указывается коммутирующий или обратный оператор, нужно быть его владельцем.

Параметры

имя

Имя определяемого оператора. Допустимые в нём символы перечислены ниже. Указанное имя может быть дополнено схемой, например так: `CREATE OPERATOR myschema.+ (...)`. Если схема не указана, оператор создаётся в текущей схеме. При этом два оператора в одной схеме могут иметь одно имя, если они работают с разными типами данных. Такое определение операторов называется *перегрузкой*.

имя_функции

Функция, реализующая этот оператор.

тип_слева

Тип данных левого операнда оператора, если он есть. Этот параметр опускается для префиксных операторов.

тип_справа

Тип данных правого операнда оператора.

коммут_оператор

Оператор, коммутирующий для данного.

обратный_оператор

Оператор, обратный для данного.

процедура_ограничения

Функция оценки избирательности ограничения для данного оператора.

процедура_соединения

Функция оценки избирательности соединения для этого оператора.

`HASHES`

Показывает, что этот оператор поддерживает соединение по хешу.

`MERGES`

Показывает, что этот оператор поддерживает соединение слиянием.

Чтобы задать имя оператора с указанием схемы в *коммут_оператор* или другом дополнительном аргументе, применяется синтаксис `OPERATOR()`, например:

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

Примечания

Более подробно информация описана в Разделе 36.14 и Разделе 36.15.

Когда вы определяете оператор, коммутирующий сам для себя, вы делаете именно это. Если же вы определяете пару коммутирующих операторов, возникает небольшое затруднение: как оператор, определяемый первым, может ссылаться на другой, ещё не определённый? Есть три решения этой проблемы:

- Во-первых, можно опустить предложение `COMMUTATOR` для первого оператора, который вы определяете, а затем добавить его в определении второго. Так как Postgres Pro знает, что коммутирующие операторы связаны парами, встречая второе определение, он автоматически возвращается к первому и добавляет в него недостающее предложение `COMMUTATOR`.

- Другой и более очевидный способ — включить предложения `COMMUTATOR` в оба определения. При обработке первого определения Postgres Pro поймёт, что `COMMUTATOR` ссылается на несуществующий оператор и создаст фиктивную запись для этого оператора в системном каталоге. Корректными в такой фиктивной записи будут только имя оператора, тип левого или правого операнда и владелец, поскольку на этом этапе только эти данные может получить Postgres Pro. На эту фиктивную запись будет ссылаться запись каталога первого оператора. При определении второго оператора система дополнит фиктивную запись новой информацией. При попытке использовать фиктивную запись до обновления этой информации будет выдана ошибка.
- Также можно определить оба оператора без предложений `COMMUTATOR`, в таком случае ссылки на коммутирующие операторы можно будет установить командой `ALTER OPERATOR`. Достаточно запустить `ALTER` для одного из операторов.

Во всех трёх случаях необходимо быть владельцем обоих операторов, чтобы назначить их коммутирующими операторами.

Пару обратных операторов можно определить теми же способами, что и пару коммутирующих.

Задать лексический приоритет оператора в команде `CREATE OPERATOR` невозможно, так как обработка приоритетов жёстко зашита в анализаторе. Подробнее приоритеты описаны в Подразделе 4.1.6.

Устаревшие параметры `SORT1`, `SORT2`, `LTCMP` и `GTCMP` ранее использовались для определения имён операторов сортировки, связанных с оператором, применяемым при соединении слиянием. Теперь это не требуется, так как информацию о связанных операторах теперь дают семейства операторов B-дерева. Если в команде отсутствует явное указание `MERGES`, все эти параметры игнорируются.

Для удаления пользовательских операторов из базы данных применяется `DROP OPERATOR`, а для изменения их свойств — ALTER OPERATOR.

Примеры

Следующая команда определяет новый оператор, проверяющий равенство площадей, для типа `box`:

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    FUNCTION = area_equal_function,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_function,
    JOIN = area_join_function,
    HASHES, MERGES
);
```

Совместимость

`CREATE OPERATOR` является языковым расширением Postgres Pro. Средства определения пользовательских операторов в стандарте SQL не описаны.

См. также

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

CREATE OPERATOR CLASS

CREATE OPERATOR CLASS — создать класс операторов

Синтаксис

```
CREATE OPERATOR CLASS имя [ DEFAULT ] FOR TYPE тип_данных
  USING индексный_метод [ FAMILY имя_семейства ] AS
  {  OPERATOR номер_стратегии имя_оператора [ ( тип_операнда, тип_операнда ) ] [ FOR
 SEARCH | FOR ORDER BY семейство_сортировки ]
   | FUNCTION номер_опорной_функции [ ( тип_операнда [ , тип_операнда ] ) ] имя_функции
 ( тип_аргумента [, ...] )
   | STORAGE тип_хранения
  } [, ... ]
```

Описание

`CREATE OPERATOR CLASS` создаёт класс операторов. Класс операторов устанавливает, как данный тип будет использоваться в индексе, определяя, какие операторы исполняют конкретные роли или «стратегии» для этого типа данных и метода индекса. Также класс операторов определяет опорные функции, которые будет задействовать метод индекса в случае выбора данного класса для столбца индекса. Все операторы и функции, используемые классом операторов, должны существовать до создания этого класса.

Если указывается имя схемы, класс операторов создаётся в указанной схеме, в противном случае — в текущей. Два класса операторов в одной схеме могут иметь одинаковые имена, только если они предназначены для разных методов индекса.

Владельцем класса операторов становится пользователь, создавший его. В настоящее время создавать классы операторов могут только суперпользователи. (Это ограничение введено потому, что ошибочное определение класса может вызвать нарушения или даже сбой в работе сервера.)

`CREATE OPERATOR CLASS` в настоящее время не проверяет, включает ли определение класса операторов все операторы и функции, требуемые для метода индекса, и образуют ли они целостный набор. Ответственность за правильность определения класса операторов лежит на пользователе.

Связанные классы операторов могут быть сгруппированы в *семейства операторов*. Чтобы поместить класс в существующее семейство, добавьте параметр `FAMILY` в `CREATE OPERATOR CLASS`. Без этого параметра новый класс помещается в семейство, имеющее то же имя, что и класс (если такое семейство не существует, оно создаётся).

За дополнительными сведениями обратитесь к Разделу 36.16.

Параметры

`имя`

Имя создаваемого класса операторов, возможно, дополненное схемой.

`DEFAULT`

Если присутствует это указание, класс операторов становится классом по умолчанию для своего типа данных. Для определённого типа данных и метода индекса можно определить не больше одного класса операторов по умолчанию.

`тип_данных`

Тип данных столбца, для которого предназначен этот класс операторов.

индексный_метод

Имя индексного метода, для которого предназначен этот класс операторов.

имя_семейства

Имя существующего семейства операторов, в которое будет добавлен этот класс. Если не указано, подразумевается семейство с тем же именем, что и класс (если такое семейство не существует, оно создаётся).

номер_стратегии

Номер стратегии индексного метода для оператора, связанного с данным классом операторов.

имя_оператора

Имя (возможно, дополненное схемой) оператора, связанного с данным классом операторов.

тип_операнда

В предложении `OPERATOR` это тип данных операнда, либо ключевое слово `NONE`, характеризующее префиксный оператор. Типы операндов обычно можно опустить, когда они совпадают с типом данных класса операторов.

В предложении `FUNCTION` это тип данных операнда, который должна поддерживать эта функция, если он отличается от входного типа данных функции (для функций сравнения в B-деревьях и хеш-функций) или от типа данных класса (для опорных функций сортировки и функций равенства образов в B-деревьях, а также для всех функций в классах операторов GiST, SP-GiST, GIN и BRIN). Обычно предполагаемые по умолчанию типы оказываются подходящими, так что *тип_операнда* указывать в `FUNCTION` не нужно (если это не функции сортировки в B-дереве, которые предназначены для сравнения разных типов данных).

семейство_сортировки

Имя (возможно, дополненное схемой) существующего семейства операторов `btree`, описывающего порядок сортировки, связанный с оператором сортировки.

Если не указано ни `FOR SEARCH` (для поиска), ни `FOR ORDER BY` (для сортировки), подразумевается `FOR SEARCH`.

номер_опорной_функции

Номер опорной функции индексного метода для функции, связанной с данным классом операторов.

имя_функции

Имя (возможно, дополненное схемой) функции, которая является опорной функцией индексного метода для данного класса операторов.

тип_аргумента

Тип данных параметра функции.

тип_хранения

Тип данных, фактически сохраняемых в индексе. Обычно это тип данных столбца, но некоторые методы индекса (в настоящее время, GiST, GIN, SP-GiST и BRIN) могут работать с отличным от него типом. Предложение `STORAGE` может присутствовать, только если метод индекса позволяет использовать другой тип данных. Если *тип_данных* столбца задан как `anyarray`, *тип_хранения* может быть объявлен как `anyelement`, чтобы показать, что записи в индексе являются членами типа элемента, принадлежащего к фактическому типу массива, для которого создаётся конкретный индекс.

Предложения `OPERATOR`, `FUNCTION` и `STORAGE` могут указываться в любом порядке.

Примечания

Так как механизмы индексов не проверяют права доступа к функциям, прежде чем вызывать их, включение функций или операторов в класс операторов по сути даёт всем право на выполнение их. Обычно это не проблема для таких функций, какие бывают полезны в классе операторов.

Операторы не должны реализовываться в функциях на языке SQL. SQL-функция вероятнее всего будет встроена в вызывающий запрос, что помешает оптимизатору понять, что этот запрос соответствует индексу.

До PostgreSQL 8.4 предложение `OPERATOR` могло включать указание `RECHECK`. Теперь это не поддерживается, так как оператор индекса может быть «неточным» и это определяется на ходу в момент выполнения. Это позволяет эффективно справляться с ситуациями, когда оператор может быть или не быть неточным.

Примеры

Команда в следующем примере определяет класс операторов индекса GiST для типа данных `_int4` (массива из `int4`). Полный пример приведён в модуле intarray.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, smallint, oid,
 internal),
        FUNCTION        2       g_int_union (internal, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

Совместимость

`CREATE OPERATOR CLASS` является расширением Postgres Pro. Команда `CREATE OPERATOR CLASS` отсутствует в стандарте SQL.

См. также

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

CREATE OPERATOR FAMILY

CREATE OPERATOR FAMILY — создать семейство операторов

Синтаксис

```
CREATE OPERATOR FAMILY имя USING индексный_метод
```

Описание

`CREATE OPERATOR FAMILY` создаёт новое семейство операторов. Семейство операторов определяет коллекцию связанных классов операторов и, возможно, некоторые дополнительные операторы и вспомогательные функции, совместимые с этими классами, но не требующиеся для функционирования каких-либо индексов. (Операторы и функции, требующиеся для работы индексов, следует группировать в соответствующем классе операторов, а не «слабо» связывать в семействе операторов. Обычно операторы с операндами одного типа привязываются к классам операторов, тогда как операторы для двух разных типов могут быть слабо связаны в семействе, содержащем классы операторов для обоих типов.)

Создаваемое семейство операторов изначально не содержит ничего. Оно должно наполняться последующими командами `CREATE OPERATOR CLASS`, добавляющими в него классы операторов, и, возможно, командами `ALTER OPERATOR FAMILY`, добавляющими «слабосвязанные» операторы и соответствующие вспомогательные функции.

Если указывается имя схемы, семейство операторов создаётся в указанной схеме, в противном случае — в текущей. Два семейства операторов в одной схеме могут иметь одинаковые имена, только если они предназначены для разных методов индекса.

Владельцем семейства операторов становится пользователь, создавший его. В настоящее время создавать семейства операторов могут только суперпользователи. (Это ограничение введено потому, что ошибочное определение семейства может вызвать нарушения или даже сбой в работе сервера.)

За дополнительными сведениями обратитесь к Разделу 36.16.

Параметры

`имя`

Имя создаваемого семейства операторов, возможно, дополненное схемой.

`индексный_метод`

Имя индексного метода, для которого предназначено это семейство операторов.

Совместимость

`CREATE OPERATOR FAMILY` является расширением Postgres Pro. Команда `CREATE OPERATOR FAMILY` отсутствует в стандарте SQL.

См. также

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE POLICY

CREATE POLICY — создать новую политику защиты на уровне строк для таблицы

Синтаксис

```
CREATE POLICY имя ON имя_таблицы
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { имя_роли | PUBLIC | CURRENT_ROLE | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( выражение_USING ) ]
    [ WITH CHECK ( выражение_CHECK ) ]
```

Описание

Команда `CREATE POLICY` определяет для таблицы новую политику защиты на уровне строк. Заметьте, что для таблицы должна быть включена защита на уровне строк (используя `ALTER TABLE ... ENABLE ROW LEVEL SECURITY`), чтобы созданные политики действовали.

Политика даёт разрешение на выборку, добавление, изменение или удаление строк, удовлетворяющих соответствующему выражению политики. Существующие строки таблицы проверяются по выражению, указанному в `USING`, тогда как строки, которые могут быть созданы командами `INSERT` или `UPDATE` проверяются по выражению, указанному в `WITH CHECK`. Когда выражение `USING` истинно для заданной строки, эта строка видна пользователю, а если ложно или выдаёт NULL, строка не видна. Когда выражение `WITH CHECK` истинно для заданной строки, эта строка добавляется или изменяется, а если ложно или выдаёт NULL, происходит ошибка.

Для операторов `INSERT`, `UPDATE` и `MERGE` выражения `WITH CHECK` применяются после срабатывания триггеров `BEFORE`, но до того, как будут собственно модифицированы данные. Таким образом, триггер `BEFORE ROW` может изменить данные, подлежащие добавлению, и повлиять на результат условия политики защиты. Выражения `WITH CHECK` обрабатываются до каких-либо других ограничений.

Имена политик задаются на уровне таблицы. Таким образом, одно имя политики можно использовать в нескольких разных таблицах и в каждой дать отдельное, подходящее этой таблице определение политики.

Политики могут применяться для определённых команд или для определённых ролей. По умолчанию создаваемые политики применяются для всех команд и ролей, если явно не задано другое. К одной команде могут применяться несколько политик; подробнее рассказывается ниже. В Таблице 301 показано, как к определённым командам применяются разные типы политик.

Для политик, которые могут иметь и выражения `USING`, и выражения `WITH CHECK` (`ALL` и `UPDATE`), в случае отсутствия выражения `WITH CHECK` выражение `USING` будет использоваться и для определения видимости строк (обычное назначение `USING`) и для определения, какие строки разрешено добавить (назначение `WITH CHECK`).

Если для таблицы включена защита на уровне строк, но применимые политики отсутствуют, предполагается политика «запрета по умолчанию», так что никакие строки нельзя будет увидеть или изменить.

Параметры

`имя`

Имя создаваемой политики. Оно должно отличаться от имён других политик для этой таблицы.

`имя_таблицы`

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется эта политика.

`PERMISSIVE`

Указывает, что создаваемая политика должна быть разрешительной. Все разрешительные политики, которые применяются к данному запросу, будут объединяться вместе логическим оператором «ИЛИ». Создавая разрешительные политики, администраторы могут расширять множество записей, к которым можно обращаться. Политики являются разрешительными по умолчанию.

`RESTRICTIVE`

Указывает, что создаваемая политика должна быть ограничительной. Все ограничительные политики, которые применяются к данному запросу, будут объединяться вместе логическим оператором «И». Создавая ограничительные политики, администраторы могут сократить множество записей, к которым можно обращаться, так как для каждой записи должны удовлетворяться все ограничительные политики.

Заметьте, что для получения доступа к записям должна быть определена минимум одна разрешительная политика, и только в дополнение к ней могут быть определены имеющие смысл ограничительные политики, ограничивающие доступ. Если разрешительные политики отсутствуют, ни к каким записям обращаться нельзя. Когда определены и разрешительные, и ограничительные политики, запись будет доступна, если удовлетворяется минимум одна из разрешительных политик и все ограничительные.

команда

Команда, к которой применяется политика. Допустимые варианты: `ALL`, `SELECT`, `INSERT`, `UPDATE` и `DELETE`. `ALL` (все) подразумевается по умолчанию. Особенности их применения описаны ниже.

имя_роли

Роль (роли), к которой применяется политика. По умолчанию подразумевается `PUBLIC`, то есть политика применяется ко всем ролям.

выражение_USING

Произвольное условное выражение SQL (возвращающее `boolean`). Это условное выражение не может содержать агрегатные или оконные функции. Когда включена защита на уровне строк, оно добавляется в запросы, обращающиеся к данной таблице, и в их результатах оказываются видимыми только те строки, для которых оно выдаёт true. Все строки, для которых это выражение возвращает false или NULL, не будут видны пользователю (в запросе `SELECT`), и не будут доступны для модификации (запросами `UPDATE` или `DELETE`). Такая строка просто пропускается, ошибка при этом не выдаётся.

выражение_CHECK

Произвольное условное выражение SQL (возвращающее `boolean`). Это условное выражение не может содержать агрегатные или оконные функции. Когда включена защита на уровне строк, оно применяется в запросах `INSERT` и `UPDATE` к этой таблице, так что в них принимаются только те строки, для которых оно выдаёт true. Если это выражение выдаёт false или NULL для любой из добавляемых записей или записей, получаемых при изменении, выдаётся ошибка. Заметьте, что *ограничение_проверки* вычисляется для предлагаемого нового содержимого строки, а не для существующих данных.

Политики по командам

`ALL`

Указание `ALL` для политики означает, что она применяется ко всем командам, вне зависимости от типа. Если существует политика `ALL` и другие более детализированные политики, тогда будет применяться и политика `ALL`, и более детализированная политика (или политики). Кроме того, политики `ALL` с выражением `USING` будут применяться и к стороне выборки, и к стороне изменения данных в запросе, если определено только выражение `USING`.

Например, когда выполняется `UPDATE`, политика `ALL` будет фильтровать и строки, которые сможет прочитать `UPDATE` для изменения (применяя выражение `USING`), и окончательные изменённые строки, проверяя, можно ли записать их в таблицу (применяя выражение `WITH CHECK`, если оно определено, или `USING` в противном случае). Если команда `INSERT` или `UPDATE` пытается добавить в таблицу строки, не удовлетворяющие выражению `WITH CHECK` политики `ALL`, вся команда будет прервана.

`SELECT`

Указание `SELECT` для политики означает, что она применяется к запросам `SELECT` и тогда, когда при обращении к отношению, для которого определена политика, задействуется право `SELECT`. В результате запрос `SELECT` выдаст только те записи из отношения, которые удовлетворят политике `SELECT`, и запрос, использующий право `SELECT`, например, запрос `UPDATE`, увидит только записи, разрешённые политикой `SELECT`. Для политики `SELECT` не может задаваться выражение `WITH CHECK`, так как оно действует только когда записи читаются из отношения.

`INSERT`

Указание `INSERT` для политики означает, что она применяется к командам `INSERT`, а также к командам `MERGE` с действиями `INSERT`. Если вставляемые строки не проходят проверку политики, выдаётся ошибка нарушения политики и вся команда `INSERT` прерывается. Для политики `INSERT` не может задаваться выражение `USING`, так как она действует только когда в отношение добавляются записи.

Заметьте, что `INSERT` с указанием `ON CONFLICT DO UPDATE` проверяет выражения `WITH CHECK` политик `INSERT` только для строк, добавляемых в отношение по пути `INSERT`.

`UPDATE`

Выбор типа `UPDATE` для политики означает, что она будет применяться к командам `UPDATE`, `SELECT FOR UPDATE` и `SELECT FOR SHARE`, а также к дополнительным предложениям `ON CONFLICT DO UPDATE` команд `INSERT`. Это также касается команд `MERGE` с действиями `UPDATE`. Так как `UPDATE` подразумевает извлечение существующей записи и замену её новой изменённой записью, политики `UPDATE` принимают как выражение `USING`, так и `WITH CHECK`. Выражение `USING` определяет, какие записи команда `UPDATE` сможет увидеть для последующего изменения, а выражение `WITH CHECK` — какие изменённые строки сохранить в отношении.

Если в какой-либо строке изменённые значения не будут удовлетворять выражению `WITH CHECK`, произойдёт ошибка и вся команда будет прервана. Если указывается только предложение `USING`, его выражение будет применяться и в качестве `USING`, и в качестве выражения `WITH CHECK`.

Обычно команде `UPDATE` также нужно прочитать данные из столбцов подлежащего изменению отношения (например, в предложении `WHERE` или `RETURNING` либо в выражении в правой части предложения `SET`). В этом случае также требуется иметь права `SELECT` в изменяемом отношении и в дополнение к политикам `UPDATE` будут применяться соответствующие политики `SELECT` или `ALL`. Таким образом, помимо того, что пользователю должны разрешать изменение строк политики `UPDATE` или `ALL`, ему также должны разрешать доступ к изменяемым строкам политики `SELECT` или `ALL`.

Когда для команды `INSERT` задано вспомогательное предложение `ON CONFLICT DO UPDATE`, если выбирается путь `UPDATE`, строка, подлежащая изменению, сначала проверяется по выражениям `USING` всех политик `UPDATE`, а затем изменённая строка ещё раз проверяется по выражениям `WITH CHECK`. Заметьте, однако, что в отличие от отдельной команды `UPDATE`, если существующая строка не удовлетворяет выражениям `USING`, будет выдана ошибка (путь `UPDATE` *никогда* не пропускается неявно).

`DELETE`

Указание `DELETE` для политики означает, что она применяется к командам `DELETE`. Команда `DELETE` будет видеть только те строки, которые позволит эта политика. При этом строки могут быть видны через `SELECT`, но удалить их будет нельзя, если они не удовлетворяют выражению `USING` политики `DELETE`.

В большинстве случаев команде `DELETE` также нужно прочитать данные из столбцов в отношении, из которого осуществляется удаление (например, в предложении `WHERE` или `RETURNING`). В таких случаях необходимо также иметь право `SELECT` для этого отношения, и в дополнение к политикам `DELETE` будут применятся соответствующие политики `SELECT` или `ALL`. Таким образом, пользователь должен получить доступ к удаляемым строкам через политики `SELECT` или `ALL`, помимо того что удаление этих строк ему должны разрешить политики `DELETE` или `ALL`.

Для политики `DELETE` не может задаваться выражение `WITH CHECK`, так как она применяется только тогда, когда записи удаляются из отношения, а в этом случае новые строки, подлежащие проверке, отсутствуют.

Таблица 301. Политики, применяемые для разных команд

Команда	Политика `SELECT/ALL`	Политика `INSERT/ALL`	Политика `UPDATE/ALL`		Политика `DELETE/ALL`
	Выражение `USING`	Выражение `WITH CHECK`	Выражение `USING`	Выражение `WITH CHECK`	Выражение `USING`
`SELECT`	Существующая строка	—	—	—	—
`SELECT FOR UPDATE/SHARE`	Существующая строка	—	Существующая строка	—	—
`INSERT` / `MERGE ... THEN INSERT`	—	Новая строка	—	—	—
`INSERT ... RETURNING`	Новая строка [a]	Новая строка	—	—	—
`UPDATE` / `MERGE ... THEN UPDATE`	Существующие и новые строки [a]	—	Существующая строка	Новая строка	—
`DELETE`	Существующая строка [a]	—	—	—	Существующая строка
`ON CONFLICT DO UPDATE`	Существующие и новые строки	—	Существующая строка	Новая строка	—

[a] Если для существующей или новой строки требуется доступ на чтение (например, предложение `WHERE` или `RETURNING`, обращающееся к столбцам отношения).

Применение нескольких политик

Когда к одной команде применяются несколько политик для различных типов команд (как например, политики `SELECT` и `UPDATE` применяются к команде `UPDATE`), пользователь должен иметь разрешения всех этих типов (например, разрешение для выборки строк из отношения, а также разрешение на их изменение). Таким образом, выражения для одного типа политики комбинируются с выражениями для другого типа операцией `И`.

Когда к одной команде применяются несколько политик для одного типа команды, доступ к отношению должна дать как минимум одна разрешительная (`PERMISSIVE`) политика, а также должны удовлетворяться все ограничительные (`RESTRICTIVE`) политики. Таким образом выражения всех политик `PERMISSIVE` объединяются операцией `ИЛИ`, выражения всех политик `RESTRICTIVE` объединяются операцией `И`, а полученные результаты объединяются операцией `И`. Если политики `PERMISSIVE` отсутствуют, доступ запрещается.

Заметьте, что при объединении нескольких политик, политики `ALL` применяются как политики каждого применимого в данном случае типа.

Например, в команде `UPDATE`, требующей разрешений и для `SELECT`, и для `UPDATE`, в случае существования нескольких применимых политик каждого типа они будут объединяться следующим образом:

```
выражение from RESTRICTIVE SELECT/ALL policy 1
AND
выражение from RESTRICTIVE SELECT/ALL policy 2
AND
...
AND
(
  выражение from PERMISSIVE SELECT/ALL policy 1
  OR
  выражение from PERMISSIVE SELECT/ALL policy 2
  OR
  ...
)
AND
выражение from RESTRICTIVE UPDATE/ALL policy 1
AND
выражение from RESTRICTIVE UPDATE/ALL policy 2
AND
...
AND
(
  выражение from PERMISSIVE UPDATE/ALL policy 1
  OR
  выражение from PERMISSIVE UPDATE/ALL policy 2
  OR
  ...
)
```

Примечания

Чтобы создать или изменить политики для таблицы, нужно быть её владельцем.

Хотя политики применяются к явно выполняемым запросам к таблицам БД, они не применяются, когда система выполняет внутренние проверки ссылочной целостности или проверяет ограничения. Это означает, что существуют косвенные пути проверить существование заданного значения. Например, можно попытаться вставить повторяющееся значение в столбец, образующий первичный ключ или имеющую ограничение уникальности. Если при этом произойдёт ошибка, пользователь может заключить, что это значение уже существует. (В данном случае предполагается, что политика разрешает пользователю вставлять записи, которые он может не видеть.) Подобный приём также возможен, если пользователь может вставлять записи в таблицу, которая ссылается на другую, иным образом не видимую. Существование значения можно определить, вставив его в подчинённую таблицу, при этом успешный результат операции будет признаком того, что это значение есть в главной таблице. Эти изъяны можно устранить, либо тщательно разработав политики, которые вовсе не позволят пользователям выполнять операции добавления, изменения и удаления, по результатам которых можно узнать о значениях в таблицах, не видимых иным образом, либо используя генерируемые значения (например, суррогатные ключи).

Вообще система будет применять фильтры, устанавливаемые политиками безопасности, до условий в запросах пользователя, чтобы предотвратить нежелательную утечку защищаемых данных через пользовательские функции, которые могут быть недоверенными. Однако функции и операторы, помеченные системой (или системным администратором) как `LEAKPROOF` (герметичные) могут вычисляться до условий политики, так как они считаются доверенными.

Так как выражения политики добавляются непосредственно в запрос пользователя, они выполняются с правами пользователя, запускающего исходный запрос. Таким образом, пользователи, на которых распространяется заданная политика, должны иметь права для обращения ко всем таблицам и функциям, задействованным в выражении, иначе им просто будет отказано в доступе при попытке обращения к целевой таблице (если для неё включена защита на уровне строк). Однако это не влияет на работу представлений — как и с обычными запросами и представлениями, проверки разрешений и политики для нижележащих таблиц представления будут выполняться с правами владельца представления, и при этом будут действовать политики, распространяющиеся на этого владельца, за исключением случаев, когда представление определяется с характеристикой `security_invoker` (см. `CREATE VIEW`).

Для `MERGE` не существует отдельной политики. Когда выполняется `MERGE`, в зависимости от фактически выполняемых действий применяются политики, определённые для `SELECT`, `INSERT`, `UPDATE` и `DELETE`.

Дополнительное описание и практические примеры можно найти в Разделе 5.9.

Совместимость

`CREATE POLICY` является расширением Postgres Pro.

См. также

ALTER POLICY, DROP POLICY, ALTER TABLE

CREATE PROCEDURE

CREATE PROCEDURE — создать процедуру

Синтаксис

```
CREATE [ OR REPLACE ] PROCEDURE
    имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ { DEFAULT |
 = } выражение_по_умолчанию ] [, ...] ] )
  { LANGUAGE имя_языка
    | TRANSFORM { FOR TYPE имя_типа } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET параметр_конфигурации { TO значение | = значение | FROM CURRENT }
    | AS 'определение'
    | AS 'объектный_файл', 'объектный_символ'
    | тело_sql
  } ...
```

Описание

Команда `CREATE PROCEDURE` определяет новую процедуру. `CREATE OR REPLACE PROCEDURE` создаёт новую процедуру либо заменяет определение уже существующей. Чтобы определить процедуру, необходимо иметь право `USAGE` для соответствующего языка.

Если указано имя схемы, процедура создаётся в заданной схеме, в противном случае — в текущей. Имя новой процедуры должно отличаться от имён существующих процедур и функций с такими же типами аргументов в этой схеме. Однако процедуры и функции с аргументами разных типов могут иметь одно имя (это называется *перегрузкой*).

Команда `CREATE OR REPLACE PROCEDURE` предназначена для изменения текущего определения существующей процедуры. С её помощью нельзя изменить имя или типы аргументов (если попытаться сделать это, будет создана новая отдельная процедура).

Когда команда `CREATE OR REPLACE PROCEDURE` заменяет существующую процедуру, владелец и права доступа к этой процедуре не меняются. Все другие свойства процедуры получают значения, задаваемые командой явно или по умолчанию. Чтобы заменить процедуру, необходимо быть её владельцем (или быть членом роли-владельца).

Владельцем процедуры становится создавший её пользователь.

Чтобы создать процедуру, необходимо иметь право `USAGE` для типов её аргументов.

За дополнительной информацией о разработке процедур обратитесь к Разделу 36.4.

Параметры

имя

Имя создаваемой процедуры (возможно, дополненное схемой).

режим_аргумента

Режим аргумента: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`.

имя_аргумента

Имя аргумента.

тип_аргумента

Тип данных аргумента процедуры (возможно, дополненный схемой), при наличии аргументов. Тип аргументов может быть базовым, составным или доменным, либо это может быть ссылка на столбец таблицы.

В зависимости от языка реализации также может допускаться указание «псевдотипов», например, `cstring`. Псевдотипы показывают, что фактический тип аргумента либо определён не полностью, либо существует вне множества обычных типов SQL.

Ссылка на тип столбца записывается в виде *`имя_таблицы`*`.`*`имя_столбца`*`%TYPE`. Иногда такое указание бывает полезно, так как позволяет создать процедуру, независящую от изменений в определении таблицы.

`выражение_по_умолчанию`

Выражение, используемое для вычисления значения по умолчанию, если параметр не задан явно. Результат выражения должен сводиться к типу соответствующего параметра. Для всех входных параметров, следующих за параметром с определённым значением по умолчанию, также должны быть определены значения по умолчанию.

`имя_языка`

Имя языка, на котором реализована процедура. Это может быть `sql`, `c`, `internal` либо имя процедурного языка, определённого пользователем, например, `plpgsql`. Если присутствует *`тело_sql`*, подразумевается язык `sql`. Вариант написания этого имени в апострофах считается устаревшим и требует точного совпадения регистра.

`TRANSFORM { FOR TYPE` *`имя_типа`* `} [, ... ] }`

Устанавливает список трансформаций, которые должны применяться при вызове процедуры. Трансформации выполняют преобразования между типами SQL и типами данных, специфичными для языков; см. CREATE TRANSFORM. Преобразования встроенных типов обычно жёстко предопределены в реализациях процедурных языков, так что их здесь указывать не нужно. Если реализация процедурного языка не может обработать тип и трансформация для него отсутствует, будет выполнено преобразование типов по умолчанию, но это зависит от реализации.

`[EXTERNAL] SECURITY INVOKER`
`[EXTERNAL] SECURITY DEFINER`

Характеристика `SECURITY INVOKER` (безопасность вызывающего) показывает, что процедура будет выполняться с правами пользователя, вызвавшего её. Этот вариант подразумевается по умолчанию. Вариант `SECURITY DEFINER` (безопасность определившего) обозначает, что процедура выполняется с правами пользователя, владеющего ей.

Ключевое слово `EXTERNAL` (внешняя) допускается для соответствия стандарту SQL, но является необязательным, так как, в отличие от SQL, эта характеристика распространяется на все процедуры, а не только внешние.

В процедуре с характеристикой `SECURITY DEFINER` не могут выполняться операторы управления транзакциями (например, `COMMIT` и `ROLLBACK` в некоторых языках).

`параметр_конфигурации`
`значение`

Предложение `SET` определяет, что при вызове процедуры указанный параметр конфигурации должен принять заданное значение, а затем восстановить своё предыдущее значение при завершении процедуры. Предложение `SET FROM CURRENT` сохраняет в качестве значения, которое будет применено при входе в процедуру, значение, действующее в момент выполнения `CREATE PROCEDURE`.

Если в определение процедуры добавлено `SET`, то действие команды `SET LOCAL`, выполняемой внутри процедуры для того же параметра, ограничивается телом процедуры: предыдущее значение параметра так же будет восстановлено при завершении процедуры. Однако обычная команда `SET` (без `LOCAL`) переопределяет предложение `SET`, как и предыдущую команду `SET LOCAL`: действие такой команды будет сохранено и после завершения процедуры, если только не произойдёт откат транзакции.

Если к определению процедуры добавлено `SET`, то в этой процедуре не могут выполняться операторы управления транзакциями (например, `COMMIT` и `ROLLBACK` в некоторых языках).

За подробными сведениями об именах и значениях параметров обратитесь к SET и Главе 18.

определение

Строковая константа, определяющая реализацию процедуры; её значение зависит от языка. Это может быть имя внутренней процедуры, путь к объектному файлу, команда SQL или код на процедурном языке.

Часто бывает полезно заключать определение процедуры в доллары (см. Подраздел 4.1.2.4), а не в традиционные апострофы. Если не использовать доллары, все апострофы и обратные косые черты в определении процедуры придётся экранировать, дублируя их.

объектный_файл, *объектный_символ*

Эта форма предложения `AS` применяется для динамически загружаемых процедур на языке C, когда имя процедуры в коде C не совпадает с именем процедуры в SQL. Строка *объектный_файл* задаёт имя файла, содержащего скомпилированную процедуру на C (данная команда воспринимает эту строку так же, как и `LOAD`). Строка *объектный_символ* задаёт символ скомпонованной процедуры, то есть имя процедуры в исходном коде на языке C. Если объектный символ не указан, предполагается, что он совпадает с именем определяемой SQL-процедуры.

Если повторные вызовы `CREATE PROCEDURE` ссылаются на один и тот же объектный файл, он загружается в рамках сеанса только один раз. Чтобы выгрузить и загрузить этот файл снова (например, в процессе разработки), начните новый сеанс.

тело_sql

Тело процедуры в стиле `LANGUAGE SQL`. Это должен быть блок вида

```
BEGIN ATOMIC
  оператор;
  оператор;
  ...
  оператор;
END
```

Оно определяется подобно телу, задаваемому строковой константой (см. *определение* выше), но есть и некоторые различия. Эта форма работает только с функциями в стиле `LANGUAGE SQL`, тогда как форма со строковой константой поддерживается для всех языков. Она разбирается во время определения процедуры, тогда как форма со строковой константой — во время выполнения; как следствие, эта форма не поддерживает полиморфные типы аргументов и другие конструкции, которые нельзя обработать во время определения процедуры. С данной формой отслеживаются зависимости процедуры от объектов, используемых в её теле, так что команда `DROP ... CASCADE` выполнится корректно, тогда как в случае определения тела в строковой константе после такого удаления могут остаться неполноценные процедуры. Наконец, данная форма в большей степени соответствует стандарту SQL и совместима с другими реализациями SQL.

Примечания

Дополнительные детали создания функций, которые применимы и к процедурам, описываются в CREATE FUNCTION.

Чтобы выполнить процедуру, воспользуйтесь командой CALL.

Примеры

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
```

```
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;
```

или

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
BEGIN ATOMIC
  INSERT INTO tbl VALUES (a);
  INSERT INTO tbl VALUES (b);
END;
```

и пример вызова:

```
CALL insert_data(1, 2);
```

Совместимость

Команда `CREATE FUNCTION` определена в стандарте SQL. Её реализация в Postgres Pro может использоваться совместимым образом, но дополнена множеством расширений. За подробностями обратитесь также к CREATE FUNCTION.

См. также

ALTER PROCEDURE, DROP PROCEDURE, CALL, CREATE FUNCTION

CREATE PROFILE

CREATE PROFILE — создать профиль

Синтаксис

```
CREATE PROFILE [ IF NOT EXISTS ] имя [ LIMIT параметр значение [ ... ] ]

Здесь параметр:

      FAILED_LOGIN_ATTEMPTS
    | PASSWORD_REUSE_TIME
    | PASSWORD_REUSE_MAX
    | PASSWORD_LIFE_TIME
    | PASSWORD_GRACE_TIME
    | USER_INACTIVE_TIME
    | FAILED_AUTH_KEEP_TIME
    | PASSWORD_MIN_UNIQUE_CHARS
    | PASSWORD_MIN_LEN
    | PASSWORD_REQUIRE_COMPLEX

CREATE PROFILE [ IF NOT EXISTS ] имя FROM существующий_профиль
```

Описание

Команда `CREATE PROFILE` добавляет новый профиль в кластер баз данных Postgres Pro. Выполнять эту команду разрешено только суперпользователям базы данных или пользователям с правами роли `pg_manage_profiles`.

Профиль определяет набор параметров, ограничивающих использование базы данных. В частности, профили Postgres Pro позволяют установить парольную политику для пользователей, которым они назначены. Профили определяются на уровне кластера баз данных, поэтому они распространяются на все базы в кластере.

По умолчанию все параметры нового профиля принимают значения `DEFAULT`, то есть наследуют значения, установленные для встроенного профиля `default`. Изначально профиль `default` не устанавливает никаких ограничений, но это можно изменить, воспользовавшись командой `ALTER PROFILE`. Значение `UNLIMITED` указывает, что данный параметр не накладывает ограничений.

Когда роли назначается профиль, на неё распространяются все установленные в нём ограничения. Всем новым ролям по умолчанию назначается профиль `default`, но вы можете явно задать другой профиль при создании роли.

Параметры

`имя`

Имя нового профиля.

`FAILED_LOGIN_ATTEMPTS` *`значение`*

Задаёт число неудачных попыток входа, при котором роль блокируется. Суперпользователь может разблокировать заблокированную роль, выполнив команду ALTER ROLE с предложением `ACCOUNT UNLOCK`.

Обратите внимание, что при каждой попытке входа со стороны пользователя фактически может предприниматься несколько попыток входа. Например, когда пользователь пытается подключиться к серверу, поддерживающему SSL, клиенты на базе libpq по умолчанию дополнительно предпринимают попытку подключения без SSL, если подключиться через SSL не удаётся.

Возможные значения: положительные целые числа, `DEFAULT` (по умолчанию) и `UNLIMITED` (без ограничения).

`PASSWORD_REUSE_TIME` *значение*

Задаёт период (в днях), в течение которого нельзя будет использовать старый пароль. Возможные значения: неотрицательные действительные числа, значения типа `interval`, `DEFAULT` и `UNLIMITED`.

Этот параметр надо устанавливать вместе с `PASSWORD_REUSE_MAX`, так как они действуют в совокупности. Если они оба имеют значение `UNLIMITED`, повторное использование паролей никак не ограничивается. Если же только один из них равен `UNLIMITED`, повторно использовать старые пароли будет нельзя.

`PASSWORD_REUSE_MAX` *значение*

Задаёт количество смен паролей, после которого можно будет повторно использовать старый пароль. Возможные значения: неотрицательные целые числа, `DEFAULT` и `UNLIMITED`.

Этот параметр надо устанавливать вместе с `PASSWORD_REUSE_TIME`, так как они действуют в совокупности. Если они оба имеют значение `UNLIMITED`, повторное использование паролей никак не ограничивается. Если же только один из них равен `UNLIMITED`, повторно использовать старые пароли будет нельзя.

`PASSWORD_LIFE_TIME` *значение*

Задаёт период (в днях), в течение которого можно использовать пароль на ведущем сервере. Возможные значения: действительные числа, значения типа `interval`, `DEFAULT` и `UNLIMITED`. Пересчитанное в секунды значение должно быть больше `0`. По истечении срока действия пароля последующие попытки подключения соответствующей роли будут отвергнуты. Разблокировать роль и вернуть ей возможность подключения можно с помощью команды ALTER ROLE.

Однако если настроена аутентификация LDAP и эта роль не заблокирована в ней, то она по-прежнему сможет подключаться к резервным серверам.

Если также установлен параметр `PASSWORD_GRACE_TIME`, срок действия пароля продлевается на заданный период. В течение этого периода лояльности пользователю будет предлагаться сменить пароль, но при этом он сможет осуществлять вход.

`PASSWORD_GRACE_TIME` *значение*

Задаёт период (в днях), в течение которого будет выдаваться предупреждение об истечении срока действия пароля, но использование этого пароля будет разрешено. Задать срок действия пароля можно в атрибуте `VALID UNTIL` роли или в параметре `PASSWORD_LIFE_TIME` профиля.

Возможные значения: неотрицательные действительные числа, значения типа `interval`, `DEFAULT` и `UNLIMITED`. Если для параметра `PASSWORD_GRACE_TIME` задано значение `UNLIMITED`, срок действия пароля по сути не ограничен.

`USER_INACTIVE_TIME` *значение*

Задаёт максимальный период (в днях) с момента последнего входа пользователя, в течение которого пользователь должен осуществить очередное подключение. В случае отсутствия подключений по истечении этого периода роль пользователя блокируется. Суперпользователь может разблокировать заблокированную роль командой ALTER ROLE с предложением `ACCOUNT UNLOCK`. Возможные значения: действительные числа, значения типа `interval`, `DEFAULT` и `UNLIMITED`. Пересчитанное в секунды значение должно быть больше `0`.

`FAILED_AUTH_KEEP_TIME` *`значение`*

Задаёт период (в днях), в течение которого хранится информация о первой ошибке аутентификации пользователя. Возможные значения: действительные числа, значения типа `interval`, `DEFAULT` или `UNLIMITED`. Пересчитанное в секунды значение должно быть больше `0`. При попытке пользователя войти в систему по истечении этого периода, счётчик неудачных попыток входа (см. параметр `FAILED_LOGIN_ATTEMPTS`) сбрасывается, и пользователь разблокируется, если он был заблокирован ранее из-за ошибок аутентификации.

`PASSWORD_MIN_UNIQUE_CHARS` *`значение`*

Задаёт минимальное количество уникальных символов в пароле. Возможные значения: положительные целые числа, `DEFAULT` и `UNLIMITED`.

`PASSWORD_MIN_LEN` *`значение`*

Задаёт минимальное количество символов в пароле. Возможные значения: положительные целые числа, `DEFAULT` и `UNLIMITED`.

`PASSWORD_REQUIRE_COMPLEX [значение]`

Определяет, проверяется ли сложность пароля. Если эта проверка включена, пароль должен соответствовать следующим требованиям:

- Пароль содержит как минимум по одному символу из трёх перечисленных групп: строчные буквы, прописные буквы, цифры и специальные символы
- Пароль не содержит имя пользователя

Возможные значения: логические значения или `DEFAULT`. Если значение параметра опущено, подразумевается `true`.

`IF NOT EXISTS`

Не считать ошибкой, если профиль с таким именем уже существует.

`существующий_профиль`

Имя копируемого существующего профиля. Новый профиль получит те же свойства, что и существующий, но будет независимым объектом.

Важно

Все буквы или их аналоги в языках без разделения на регистры (хинди, китайский и т.п.) в кодировке UTF-8 считаются строчными буквами.

Замечания

Изменить значения параметров профиля позволяет команда ALTER PROFILE, а удалить профиль — DROP PROFILE. Все параметры, задаваемые в `CREATE PROFILE`, можно изменить позже с помощью команды `ALTER PROFILE`.

Предупреждение

Параметры `PASSWORD_REUSE_TIME` и `PASSWORD_REUSE_MAX` могут не работать, если при установлении пароля он передаётся в зашифрованном виде. Так, в частности, устанавливает пароль команда `\password` в psql (за подробностями обратитесь к Подразделу «Метакоманды»). Если пароль хешируется с применением MD5, его можно в зашифрованном виде сравнить с предыдущим, и таким образом контролировать его в соответствии с параметрами `PASSWORD_REUSE_TIME` и `PASSWORD_REUSE_MAX`. Однако сравнивать пароли, зашифрованные алгоритмом SCRAM-SHA-256, не представляется возможным.

Предупреждение

Параметры `PASSWORD_MIN_UNIQUE_CHARS`, `PASSWORD_MIN_LEN` и `PASSWORD_REQUIRE_COMPLEX` не работают, если при смене пароля он передаётся в зашифрованном виде.

Примеры

Создание профиля `admin_profile`:

```
CREATE PROFILE admin_profile
    LIMIT PASSWORD_REUSE_MAX 10
          PASSWORD_REUSE_TIME 30;
```

Создание роли с профилем `admin_profile`:

```
CREATE ROLE admin WITH PROFILE admin_profile;
```

Создать новый профиль из существующего:

```
CREATE PROFILE administrator FROM admin_profile;
```

Иногда удобно использовать существующий профиль в качестве шаблона для нового.

См. также

ALTER PROFILE, DROP PROFILE, CREATE ROLE

CREATE PUBLICATION

CREATE PUBLICATION — создать публикацию

Синтаксис

```
CREATE PUBLICATION имя
    [ FOR ALL TABLES
      | FOR объект_публикации [, ... ] ]
    [ WITH ( параметр_публикации [= значение] [, ... ] ) ]

где объект_публикации может быть следующим:

    TABLE [ ONLY ] имя_таблицы [ * ] [ ( имя_столбца [, ... ] ) ] [ WHERE ( выражение
 ) ] [, ... ]
    TABLES IN SCHEMA { имя_схемы | CURRENT_SCHEMA } [, ... ]
```

Описание

`CREATE PUBLICATION` создаёт новую публикацию в текущей базе данных. Имя публикации должно отличаться от имён других существующих публикаций в текущей базе.

Публикация по сути является группой таблиц, изменения в данных которых должны реплицироваться с использованием логической репликации. Подробнее о том, как публикации вписываются в схему логической репликации, рассказывается в Разделе 28.1.

Параметры

имя

Имя новой публикации.

`FOR TABLE`

Задаёт список таблиц, добавляемых в публикацию. Если перед именем таблицы указано `ONLY`, в публикацию добавляется только заданная таблица. Без `ONLY` добавляется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно добавить необязательное указание `*`, чтобы явно обозначить, что должны включаться и все дочерние таблицы. Однако это не распространяется на секционированные таблицы. Секции секционированной таблицы всегда неявно считаются частью публикации, поэтому они никогда не добавляются в публикацию явным образом.

Если указано необязательное предложение `WHERE`, оно определяет выражение *фильтра строк*. Строки, для которых *выражение* выдаёт false или null, не будут публиковаться. Заметьте, что выражение нужно заключать в круглые скобки. Это предложение не действует на команды `TRUNCATE`.

Если указан список столбцов, реплицируются только указанные столбцы. Если список столбцов не задан, через эту публикацию будут реплицироваться все столбцы таблицы, в том числе столбцы, добавленные позже. На команды `TRUNCATE` это ограничение не действуют. Подробнее о списках столбцов рассказывается в Разделе 28.5.

В публикацию могут включаться только постоянные базовые и секционированные таблицы. Временные, нежурналируемые и сторонние таблицы, а также материализованные и обычные представления не могут входить в публикацию.

Указать список столбцов в публикации, которая также публикует все таблицы в схеме (`FOR TABLES IN SCHEMA`), нельзя.

Когда в публикацию добавляется секционированная таблица, все её существующие и будущие секции неявно становятся частью публикации. Поэтому даже операции, выполняемые непосредственно с секцией, также публикуются через публикации, в которые включена её родительская таблица.

`FOR ALL TABLES`

Устанавливает, что данная публикация охватывает изменения во всех таблицах в базе данных, включая таблицы, которые будут созданы позже.

`FOR TABLES IN SCHEMA`

Устанавливает, что данная публикация охватывает изменения во всех таблицах в указанном списке схем, включая таблицы, которые будут созданы позже.

Указать схему, когда публикация также публикует таблицу со списком столбцом, нельзя.

В публикацию будут включены только постоянные базовые и секционированные таблицы, присутствующие в схеме. Временные, нежурналируемые, сторонние таблицы, материализованные представления и обычные представления из этой схемы в публикацию не войдут.

Когда секционированная таблица публикуется посредством публикации схемы, все её существующие и будущие секции неявно становятся частью публикации, независимо от того, относятся ли к опубликованной схеме или нет. Поэтому даже операции, выполняемые непосредственно с секцией, также публикуются через публикации, в которые включена её родительская таблица.

`WITH (` *`параметр_публикации`* `[=` *`значение`*`] [, ... ] )`

В этом предложении могут задаваться следующие необязательные параметры публикации:

`publish` (`string`)

Этот параметр определяет, какие операции DML будет передавать новая публикация её подписчикам. В качестве его значения через запятую задаётся список операций из следующих: `insert`, `update`, `delete` и `truncate`. По умолчанию публикуются все действия, так что этот параметр имеет значение по умолчанию `'insert, update, delete, truncate'`.

Этот параметр влияет только на операции DML. В частности, он не учитывается, когда копируются существующие данные таблиц при начальной синхронизации данных (см. Подраздел 28.8.1) для логической репликации.

`publish_via_partition_root` (`boolean`)

Этот параметр определяет, будут ли изменения в секции секционированной таблицы, включённой в публикацию, публиковаться как произошедшие в секционированной таблице (с её именем и схемой), а не в той секции, где они фактически имели место (это поведение по умолчанию). Включение этого параметра позволяет реплицировать изменения в несекционированную таблицу или в таблицу, состоящую из другого набора секций.

Иногда подписка связана с несколькими публикациями. Когда секционированная таблица публикуется в любой публикации, на которую оформлена подписка с `publish_via_partition_root = true`, изменения в этой секционированной таблице (или в её секциях) публикуются с указанием идентификатора или схемы этой секционированной таблицы, а не отдельной секции.

Этот параметр также определяет, какие фильтры строк и списки столбцов выбираются для секций; подробности описаны ниже.

Когда этот параметр включён, операции `TRUNCATE`, выполняемые непосредственно с секциями, не реплицируются.

Если для параметра типа `boolean` опустить `=` *`значение`*, это равнозначно указанию значения `TRUE`.

Примечания

Если не задано ни `FOR TABLE`, ни `FOR ALL TABLES`, ни `FOR TABLES IN SCHEMA`, публикация создаётся с пустым набором таблиц. Это полезно, если таблицы или схемы будут добавляться позднее.

Создание публикации не влечёт немедленный запуск репликации. Эта операция только определяет логику группирования и фильтрации для будущих подписчиков.

Чтобы создать публикацию, пользователь должен иметь право `CREATE` в текущей базе данных. (Разумеется, на суперпользователей это условие не распространяется.)

Чтобы добавить таблицу в публикацию, пользователь должен иметь права владельца этой таблицы. Использовать предложения `FOR ALL TABLES` и `FOR TABLES IN SCHEMA` могут только суперпользователи.

Таблицы, добавляемые в публикацию, которая охватывает операции `UPDATE` и/или `DELETE`, должны иметь свойство `REPLICA IDENTITY`. В противном случае отслеживание этих операций для таблиц будет запрещено.

Для публикации операций `UPDATE` или `DELETE` необходимо, чтобы список столбцов включал столбцы идентификатора реплики (`REPLICA IDENTITY`). Если публикация создаётся только для операций `INSERT`, ограничений на список столбцов нет.

Для публикации операций `UPDATE` и `DELETE` требуется, чтобы выражение фильтра строк (то есть предложение `WHERE`) содержало только столбцы, входящие в идентификатор реплики (`REPLICA IDENTITY`). Для публикации операции `INSERT` в выражении `WHERE` может использоваться любой столбец. В фильтре строк допускаются простые выражения, не использующие пользовательские функции, пользовательские операторы, пользовательские типы, пользовательские правила сортировки, а также непостоянные встроенные функции и ссылки на системные столбцы.

Фильтр строк таблицы становится избыточным, когда добавляется указание `FOR TABLES IN SCHEMA` и таблица принадлежит указанной схеме.

Для опубликованных секционированных таблиц фильтр строк для каждой секции берётся из опубликованной секционированной таблицы, если параметр публикации `publish_via_partition_root` равен true, или из самой секции, если он равен false (по умолчанию). Подробнее фильтры строк описаны в Разделе 28.4. Аналогично, для опубликованных секционированных таблиц список столбцов для каждой секции берётся из опубликованной секционированной таблицы, если параметр публикации `publish_via_partition_root` равен true, или из самой секции, если он равен false.

Для команды `INSERT ... ON CONFLICT` публикация будет выдавать операцию, к которой сводится команда. В зависимости от исхода команды, она может быть опубликована либо как `INSERT`, либо как `UPDATE`, либо не будет опубликована вовсе.

Для команды `MERGE` публикация будет выдавать `INSERT`, `UPDATE` или `DELETE` для каждой вставляемой, изменяемой или удаляемой строки.

Присоединение (`ATTACH`) таблицы к дереву секционирования, корень которого включён в публикацию со свойством параметра `publish_via_partition_root` равным `true`, не приводит к репликации существующего содержимого таблицы.

Команды `COPY ... FROM` публикуются в виде операций `INSERT`.

Операции DDL не публикуются.

Выражение предложения `WHERE` вычисляется от имени роли, используемой для подключения репликации.

Примеры

Создание публикации, охватывающей изменения в двух таблицах:

```
CREATE PUBLICATION mypublication FOR TABLE users, departments;
```

Создание публикации, охватывающей все изменения, которые относятся к действующим отделам:

```
CREATE PUBLICATION active_departments FOR TABLE departments WHERE (active IS TRUE);
```

Создание публикации, охватывающей все изменения во всех таблицах:

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

Создание публикации, охватывающей только операции `INSERT` в одной таблице:

```
CREATE PUBLICATION insert_only FOR TABLE mydata
    WITH (publish = 'insert');
```

Создание публикации, охватывающей все изменения в таблицах `users`, `departments` и все изменения во всех таблицах, присутствующих в схеме `production`:

```
CREATE PUBLICATION production_publication FOR TABLE users, departments, TABLES IN
 SCHEMA production;
```

Создание публикации, охватывающей все изменения во всех таблицах, присутствующих в схемах `marketing` и `sales`:

```
CREATE PUBLICATION sales_publication FOR TABLES IN SCHEMA marketing, sales;
```

Создание публикации, которая охватывает все изменения для таблицы `users`, но реплицирует только столбцы `user_id` и `firstname`:

```
CREATE PUBLICATION users_filtered FOR TABLE users (user_id, firstname);
```

Совместимость

`CREATE PUBLICATION` является расширением Postgres Pro.

См. также

ALTER PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

CREATE ROLE

CREATE ROLE — создать роль в базе данных

Синтаксис

```
CREATE ROLE имя [ [ WITH ] параметр [ ... ] ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль' | PASSWORD NULL
    | VALID UNTIL 'дата_время'
    | IN ROLE имя_роли [, ...]
    | ROLE имя_роли [, ...]
    | ADMIN имя_роли [, ...]
    | SYSID uid
    | PROFILE имя_профиля
```

Описание

`CREATE ROLE` добавляет новую роль в кластер баз данных Postgres Pro. Роль — это сущность, которая может владеть объектами и иметь определённые права в базе; роль может представлять «пользователя», «группу» или и то, и другое, в зависимости от варианта использования. За информацией об управлении пользователями и проверке подлинности обратитесь к Главе 20 и Главе 19. Чтобы выполнить эту команду, необходимо быть суперпользователем или иметь право `CREATEROLE`.

Учтите, что роли определяются на уровне кластера баз данных, так что они действуют во всех базах в кластере.

Во время создания роли можно сразу назначить создаваемую роль членом существующей роли, а также назначить существующие роли членами создаваемой роли. Правила, по которым включаются начальные параметры членства в роли, описаны ниже в предложениях `IN ROLE`, `ROLE` и `ADMIN`. Команда GRANT позволяет управлять назначением членов ролей и даёт возможность изменять эти параметры после создания новой роли.

Параметры

имя

Имя создаваемой роли.

`SUPERUSER`
`NOSUPERUSER`

Эти предложения определяют, будет ли эта роль «суперпользователем», который может переопределить все ограничения доступа в базе данных. Статус суперпользователя несёт опасность и назначать его следует только в случае необходимости. Создать нового суперпользователя может только суперпользователь. В отсутствие этих предложений по умолчанию подразумевается `NOSUPERUSER`.

`CREATEDB`
`NOCREATEDB`

Эти предложения определяют, сможет ли роль создавать базы данных. Указание `CREATEDB` даёт новой роли это право, а `NOCREATEDB` запрещает роли создавать базы данных. По умолчанию подразумевается `NOCREATEDB`. Только суперпользователи или роли с правом `CREATEDB` могут указывать `CREATEDB`.

`CREATEROLE`
`NOCREATEROLE`

Эти предложения определяют, сможет ли роль создавать, изменять и удалять другие роли, добавлять для них комментарии и изменять метку безопасности в любых ролях. Подробнее возможности, предоставляемые этим правом, описаны в Создание роли. Если предложение не указано, по умолчанию используется `NOCREATEROLE`.

`INHERIT`
`NOINHERIT`

Эти предложения влияют на статус наследования членства, когда данная роль добавляется как член другой роли, как в текущей, так и в будущих командах. В частности, они контролируют статус наследования членства, добавленного этой командой с помощью предложения `IN ROLE`, а в более поздних командах — с помощью предложения `ROLE`. Они также используются по умолчанию для статуса наследования при добавлении данной роли в качестве члена с помощью команды `GRANT`. Если предложение не указано, по умолчанию используется `INHERIT`.

В версиях Postgres Pro до 16 наследование было атрибутом на уровне роли, который управлял для этой роли всеми проверками членства во время выполнения.

`LOGIN`
`NOLOGIN`

Эти предложения определяют, разрешается ли новой роли вход на сервер; то есть, может ли эта роль стать начальным авторизованным именем при подключении клиента. Можно считать, что роль с атрибутом `LOGIN` соответствует пользователю. Роли без этого атрибута бывают полезны для управления доступом в базе данных, но это не пользователи в обычном понимании. По умолчанию подразумевается вариант `NOLOGIN`, за исключением вызова `CREATE ROLE` в виде `CREATE USER`.

`REPLICATION`
`NOREPLICATION`

Эти предложения определяют, будет ли роль ролью репликации. Чтобы роль могла подключаться к серверу в режиме репликации (в режиме физической или логической репликации) и создавать/удалять слоты репликации, у неё должен быть этот атрибут (либо это должна быть роль суперпользователя). Роль, имеющая атрибут `REPLICATION`, обладает очень большими привилегиями и поэтому этот атрибут должны иметь только роли, фактически используемые для репликации. По умолчанию подразумевается вариант `NOREPLICATION`. Выдавать атрибут `REPLICATION` могут только суперпользователи или роли с `REPLICATION`.

`BYPASSRLS`
`NOBYPASSRLS`

Эти предложения определяют, будут ли для роли игнорироваться все политики защиты на уровне строк (RLS). По умолчанию подразумевается вариант `NOBYPASSRLS`. Выдавать атрибут `BYPASSRLS` могут только суперпользователи или роли с `BYPASSRLS`.

Заметьте, что pg_dump по умолчанию отключает `row_security` (устанавливает значение `OFF`), чтобы гарантированно было выгружено всё содержимое таблицы. Если пользователь, запускающий pg_dump, не будет иметь необходимых прав, он получит ошибку. Однако суперпользователи и владелец выгружаемой таблицы всегда обходят защиту RLS.

`CONNECTION LIMIT` *предел_подключений*

Если роли разрешён вход, этот параметр определяет, сколько параллельных подключений может установить роль. Значение -1 (по умолчанию) снимает ограничение. Заметьте, что под это ограничение подпадают только обычные подключения. Ни подготовленные транзакции, ни соединения фоновых рабочих процессов в расчёт не берутся.

[`ENCRYPTED`] `PASSWORD` '*пароль*'
`PASSWORD NULL`

Задаёт пароль роли. (Пароль полезен только для ролей с атрибутом `LOGIN`, но задать его можно и для ролей без такого атрибута.) Если проверка подлинности по паролю не будет использоваться, этот параметр можно опустить. При указании пустого значения будет задан пароль NULL, что не позволит данному пользователю пройти проверку подлинности по паролю. При желании пароль NULL можно установить явно, указав `PASSWORD NULL`.

Примечание

Если задана пустая строка, пароль также будет сброшен в NULL, но в Postgres Pro до версии 10 было не так. В ранних версиях пустая строка могла приниматься или нет, в зависимости от метода аутентификации и версии сервера, а libpq не давала использовать такой пароль в любом случае. Во избежание неоднозначности указывать пустую строку в качестве пароля не следует.

Пароль всегда хранится в системных каталогах в зашифрованном виде. Ключевое слово `ENCRYPTED` не имеет эффекта, но принимается для обратной совместимости. Метод шифрования определяется параметром конфигурации password_encryption. Если представленная строка пароля уже зашифрована с применением MD5 или SCRAM, она сохраняется как есть вне зависимости от значения `password_encryption` (так как система не может расшифровать переданный зашифрованной пароль, чтобы зашифровать его по другому алгоритму). Это позволяет пересохранять зашифрованные пароли при выгрузке/восстановлении.

`VALID UNTIL` '*дата_время*'

Предложение `VALID UNTIL` устанавливает дату и время, после которого пароль роли перестаёт действовать. Если это предложение отсутствует, срок действия пароля будет неограниченным.

`IN ROLE` *имя_роли*

С предложением `IN ROLE` новая роль автоматически становится членом указанных существующих ролей. Для нового члена будет включён атрибут `SET` и отключён атрибут `ADMIN`. Атрибут `INHERIT` будет включён, если не указан атрибут `NOINHERIT`.

`ROLE` *имя_роли*

С предложением `ROLE` одна или нескольких указанных существующих ролей автоматически добавляются как члены новой роли с включённым атрибутом `SET`. Фактически это делает новую роль «группой». Для ролей, указанных в этом предложении с атрибутом `INHERIT`, будет включён атрибут `INHERIT` в новом членстве. Для новых участников атрибут `ADMIN` будет отключён.

`ADMIN` *имя_роли*

Предложение `ADMIN` работает подобно `ROLE`, но перечисленные в нём роли добавляются как члены новой роли с включённым атрибутом `ADMIN`, что даёт им право включать в новою роль другие роли.

`SYSID` *uid*

Предложение `SYSID` игнорируется, но принимается для обратной совместимости.

`PROFILE` *имя_профиля*

Задаёт профиль роли. Если это предложение опущено, роли назначается профиль `default` (за подробностями обратитесь к описанию CREATE PROFILE).

Примечания

Для изменения атрибутов роли применяется `ALTER ROLE`, а для удаления роли — `DROP ROLE`. Все атрибуты, заданные в `CREATE ROLE`, могут быть изменены позднее командами `ALTER ROLE`.

Для добавления и удаления членов ролей, используемых в качестве групп, рекомендуется использовать `GRANT` и `REVOKE`.

Предложение `VALID UNTIL` определяет срок действия только пароля, но не роли как таковой. В частности, ограничение срока пароля не действует при входе пользователя без проверки подлинности по паролю.

Определённые здесь атрибуты роли не наследуются, т. е. членство в роли, например, с правом `CREATEDB`, не позволит её участнику создавать новые базы данных, даже если членство было выдано с атрибутом `INHERIT`. Разумеется, если в выдаче членства есть атрибут `SET`, роль участника сможет выполнить команду `SET ROLE` для роли createdb, а затем создать новую базу данных.

При выдаче членства с помощью предложений `IN ROLE`, `ROLE` и `ADMIN` праводателем является роль, выполняющая эту команду.

Атрибут `INHERIT` устанавливается по умолчанию в целях обратной совместимости: в предыдущих выпусках Postgres Pro пользователи всегда обладали всеми правами групп, в которые они были включены. Однако `NOINHERIT` по смыслу ближе к тому, что описано в стандарте SQL.

Postgres Pro включает программу createuser, которая предоставляет ту же функциональность, что и команда `CREATE ROLE` (на самом деле она вызывает эту команду), но может запускаться в командной оболочке.

Ограничение `CONNECTION LIMIT` действует только приблизительно; если одновременно запускаются два сеанса, тогда как для этой роли остаётся только одно «свободное место», может так случиться, что будут отклонены оба подключения. Кроме того, это ограничение не распространяется на суперпользователей.

Указывая в этой команде незашифрованный пароль, следует проявлять осторожность. Пароль будет передаваться на сервер открытым текстом и может также записываться в историю команд клиента или в протокол работы сервера. Команда createuser, однако, передаёт пароль зашифрованным. Кроме того, в psql есть команда `\password`, с помощью которой можно безопасно сменить пароль позже.

Примеры

Создание роли, для которой разрешён вход, но не задан пароль:

```
CREATE ROLE jonathan LOGIN;
```

Создание роли с паролем:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

(`CREATE USER` действует так же, как `CREATE ROLE`, но подразумевает ещё и атрибут `LOGIN`.)

Создание роли с профилем:

```
CREATE ROLE davide WITH PROFILE admin_profile;
```

Создание роли с паролем, действующим до конца 2004 г., то есть пароль перестаёт действовать в первую же секунду 2005 г.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Создание роли, которая может создавать базы данных и управлять ролями:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Совместимость

Оператор `CREATE ROLE` описан в стандарте SQL, но стандарт требует поддержки только следующего синтаксиса:

```
CREATE ROLE имя [ WITH ADMIN имя_роли ]
```

Возможность создавать множество начальных администраторов и все другие параметры `CREATE ROLE` относятся к расширениям Postgres Pro.

В стандарте SQL определяются концепции пользователей и ролей, но в нём они рассматриваются как отдельные сущности, а все команды создания пользователей считаются внутренней спецификой СУБД. В Postgres Pro мы решили объединить пользователей и роли в единую сущность, так что роли получили дополнительные атрибуты, не описанные в стандарте.

Наиболее близкое к описанному в стандарте SQL поведение можно получить, если создавать пользователей по стандарту SQL как роли Postgres Pro с атрибутом `NOINHERIT`, а роли по стандарту SQL как роли Postgres Pro с атрибутом `INHERIT`.

Предложение `USER` является устаревшим аналогом предложения `ROLE`:

```
USER имя_роли [, ...]
```

Предложение `IN GROUP` является устаревшим аналогом предложения `IN ROLE`:

```
IN GROUP имя_роли [, ...]
```

См. также

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser, createrole_self_grant, CREATE PROFILE

CREATE RULE

CREATE RULE — создать правило перезаписи

Синтаксис

```
CREATE [ OR REPLACE ] RULE имя AS ON событие
    TO имя_таблицы [ WHERE условие ]
    DO [ ALSO | INSTEAD ] { NOTHING | команда | ( команда ; команда ... ) }

Здесь допускается событие:

    SELECT | INSERT | UPDATE | DELETE
```

Описание

`CREATE RULE` создаёт правило, применяемое к указанной таблице или представлению. `CREATE OR REPLACE RULE` либо создаёт новое правило, либо заменяет существующее с тем же именем для той же таблицы.

Система правил Postgres Pro позволяет определить альтернативное действие, заменяющее операции добавления, изменения или удаления данных в таблицах базы данных. Грубо говоря, правило описывает дополнительные команды, которые будут выполняться при вызове определённой команды для определённой таблицы. Кроме того, правило `INSTEAD` может заменить заданную команду другой, либо сделать, чтобы она не выполнялась вовсе. Правила также применяются для реализации SQL-запросов. Важно понимать, что правило это фактически механизм преобразования команд (макрос). Заданное преобразование имеет место до начала выполнения команды. Когда требуется выполнить некоторую операцию независимо для каждой физической строки, скорее всего, для этого нужно применять триггер, а не правило. Более подробно о системе правил можно узнать в Главе 39.

В настоящее время правила `ON SELECT` можно применять только к представлениям. Такое правило должно называться `"_RETURN"`, быть безусловным, с характеристикой `INSTEAD` (вместо исходного), и его действия должны состоять из единственной команды `SELECT`, определяющей видимое содержимое представления. (Само представление по сути является фиктивной таблицей без хранилища.) Лучше всего рассматривать такое правило как деталь реализации. Хотя представление можно переопределить с помощью `CREATE OR REPLACE RULE "_RETURN" AS ...`, лучше использовать `CREATE OR REPLACE VIEW`.

С помощью правил можно создать иллюзию изменяемого представления, определив правила `ON INSERT`, `ON UPDATE` и `ON DELETE` (либо только те, которых достаточно для решения поставленной задачи) и заменив операции изменения данных в представлении соответствующими действиями с другими таблицами. Если требуется поддерживать оператор `INSERT RETURNING` и подобные ему, в каждое из этих правило обязательно нужно поместить подходящее предложение `RETURNING`.

Использование правил с условиями для сложных изменений представлений связано с одним ограничением: для каждого действия, которое вы хотите разрешить для представления, *необходимо* определить безусловное правило `INSTEAD`. Если определено только условное правило, или правило не типа `INSTEAD`, система отвергнет попытки выполнить изменения, предполагая, что в некоторых случаях изменения могут свестись к операциям с фиктивной нижележащей таблицей. При желании обработать все полезные случаи изменений в условных правилах, добавьте безусловное правило `DO INSTEAD NOTHING`, чтобы система понимала, что ей никогда не придётся изменять нижележащую таблицу. Затем создайте условные правила без свойства `INSTEAD`; в тех случаях, когда они будут применяться, их действия будут добавлены к действию по умолчанию `INSTEAD NOTHING`. (Однако этот способ в настоящее время не подходит для реализации запросов `RETURNING`.)

Примечание

Представления, достаточно простые для реализации автоматического обновления (см. CREATE VIEW), могут быть изменяемыми без пользовательских правил. Хотя вы тем не менее можете создать явное правило, обычно автоматическое преобразование будет работать лучше такого правила.

Другая, заслуживающая рассмотрения, альтернатива правилам — триггеры `INSTEAD OF` (см. CREATE TRIGGER).

Параметры

`имя`

Имя создаваемого правила. Оно должно отличаться от имён любых других правил для той же таблицы. При наличии нескольких правил для одной таблицы и одного типа события они применяются в алфавитном порядке.

`событие`

Тип события: `SELECT`, `INSERT`, `UPDATE` или `DELETE`. Заметьте, что команду `INSERT` с предложением `ON CONFLICT` нельзя использовать с таблицами, для которых определены правила `INSERT` или `UPDATE`. В этом случае подумайте о применении изменяемых представлений.

`имя_таблицы`

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

`условие`

Любое выражение условия SQL (возвращающее `boolean`). Это выражение не может ссылаться на какие-либо таблицы, кроме как на `NEW` и `OLD`, и не может содержать агрегатные функции.

`INSTEAD`

`INSTEAD` указывает, что заданные команды должны выполняться *вместо* исходной команды.

`ALSO`

`ALSO` указывает, что заданные команды должны выполняться *в дополнение к* исходной команде.

Если ни `ALSO`, ни `INSTEAD` не указано, по умолчанию подразумевается `ALSO`.

`команда`

Команда или команды, составляющие действие правила. Здесь допустимы команды: `SELECT`, `INSERT`, `UPDATE`, `DELETE` и `NOTIFY`.

В параметрах *`условие`* и *`команда`* можно использовать имена специальных таблиц `NEW` и `OLD` для обращения к значениям в соответствующей таблице. `NEW` (новая) принимается в правилах `ON INSERT` и `ON UPDATE` и обозначает ссылку на новую строку, добавляемую или изменяемую. `OLD` (старая) принимается в правилах `ON UPDATE` и `ON DELETE` и обозначает ссылку на существующую строку, изменяемую или удаляемую.

Примечания

Чтобы создать или изменить правила для таблицы, нужно быть её владельцем.

В правило для `INSERT`, `UPDATE` или `DELETE` для представления можно добавить предложение `RETURNING`, выдающее столбцы представления. Это предложение будет генерировать результат, если правило сработает при выполнении команды `INSERT RETURNING`, `UPDATE RETURNING` или `DELETE RETURNING`. Когда правило срабатывает при выполнении команды без `RETURNING`, предложение `RETURNING` этого правила игнорируется. В текущей реализации только безусловные правила `INSTEAD` могут содержать `RETURNING`; более того, допускается максимум одно предложение `RETURNING` среди всех правил для некоторого события. (Благодаря этому ограничению, только одно предложение `RETURNING` может быть выбрано для вычисления результатов.) Запросы с `RETURNING` к данному представлению не будут выполняться, если ни одно из определённых для него правил не содержит предложение `RETURNING`.

Очень важно следить за тем, чтобы правила не зацикливались. Например, два следующих определения правил будут приняты Postgres Pro, но при попытке выполнить команду `SELECT` Postgres Pro сообщит об ошибке из-за рекурсивного расширения правила:

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

В настоящее время, если действие правила содержит команду `NOTIFY`, эта команда будет выполняться безусловно, то есть, `NOTIFY` будет выдаваться, даже если не найдётся никаких строк, к которым бы применялось правило. Например, в следующем примере:

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

одно событие `NOTIFY` будет отправлено при выполнении команды `UPDATE`, даже если никакие строки не соответствуют условию `id = 42`. Это недостаток текущей реализации, который может быть исправлен в будущих версиях.

Совместимость

Оператор `CREATE RULE` является языковым расширением Postgres Pro, как и вся система перезаписи запросов.

См. также

ALTER RULE, DROP RULE

CREATE SCHEMA

CREATE SCHEMA — создать схему

Синтаксис

```
CREATE SCHEMA имя_схемы [ AUTHORIZATION указание_роли ] [ элемент_схемы [ ... ] ]
CREATE SCHEMA AUTHORIZATION указание_роли [ элемент_схемы [ ... ] ]
CREATE SCHEMA IF NOT EXISTS имя_схемы [ AUTHORIZATION указание_роли ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION указание_роли

Здесь указание_роли:

    имя_пользователя
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Описание

`CREATE SCHEMA` создаёт новую схему в текущей базе данных. Имя схемы должно отличаться от имён других существующих схем в текущей базе данных.

Схема по сути представляет собой пространство имён: она содержит именованные объекты (таблицы, типы данных, функции и операторы), имена которых могут совпадать с именами других объектов, существующих в других схемах. Для обращения к объекту нужно либо «дополнить» его имя именем схемы в виде префикса, либо установить путь поиска, включающий требуемую схему. Команда `CREATE`, в которой указывается неполное имя объекта, создаёт объект в текущей схеме (схеме, стоящей первой в пути поиска; узнать её позволяет функция `current_schema`).

Команда `CREATE SCHEMA` может дополнительно включать подкоманды, создающие объекты в новой схеме. Эти подкоманды по сути воспринимаются как отдельные команды, выполняемые после создания схемы, за исключением того, что с предложением `AUTHORIZATION` все создаваемые объекты будут принадлежать указанному в нём пользователю.

Параметры

`имя_схемы`

Имя создаваемой схемы. Если оно опущено, именем схемы будет *`имя_пользователя`*. Это имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных схем.

`имя_пользователя`

Имя пользователя (роли), назначаемого владельцем новой схемы. Если опущено, по умолчанию владельцем будет пользователь, выполняющий команды. Чтобы назначить владельцем создаваемой схемы другую роль, необходимо иметь право `SET ROLE` для этой роли.

`элемент_схемы`

Оператор SQL, определяющий объект, создаваемый в новой схеме. В настоящее время `CREATE SCHEMA` может содержать только подкоманды `CREATE TABLE`, `CREATE VIEW`, `CREATE INDEX`, `CREATE SEQUENCE`, `CREATE TRIGGER` и `GRANT`. Создать объекты других типов можно отдельными командами после создания схемы.

`IF NOT EXISTS`

Не делать ничего (только выдать замечание), если схема с таким именем уже существует. Когда используется это указание, эта команда не может содержать подкоманды *`элемент_схемы`*.

Примечания

Чтобы создать схему, пользователь должен иметь право `CREATE` в текущей базе данных. (Разумеется, на суперпользователей это условие не распространяется.)

Примеры

Создание схемы:

```
CREATE SCHEMA myschema;
```

Создание схемы для пользователя `joe`; схема так же получит имя `joe`:

```
CREATE SCHEMA AUTHORIZATION joe;
```

Создание схемы с именем `test`, владельцем которой будет пользователь `joe`, если только схема `test` ещё не существует. (Является ли владельцем существующей схемы пользователь `joe`, значения не имеет.)

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

Создание схемы, в которой сразу создаются таблица и представление:

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

Заметьте, что отдельные подкоманды не завершаются точкой с запятой.

Следующие команды приводят к тому же результату другим способом:

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

Совместимость

Стандарт SQL также допускает в команде `CREATE SCHEMA` предложение `DEFAULT CHARACTER SET` и дополнительные типы подкоманд, которые Postgres Pro в настоящее время не принимает.

В стандарте SQL говорится, что подкоманды в `CREATE SCHEMA` могут следовать в любом порядке. Однако текущая реализация в Postgres Pro не воспринимает все возможные варианты ссылок вперёд в подкомандах, поэтому иногда возникает необходимость переупорядочить подкоманды, чтобы исключить такие ссылки.

Согласно стандарту SQL, владелец схемы всегда владеет всеми объектами в ней, но Postgres Pro допускает размещение в схемах объектов, принадлежащих не владельцу схемы. Такая ситуация возможна, только если владелец схемы даст право `CREATE` в этой схеме кому-либо другому, либо объекты в ней будет создавать суперпользователь.

Указание `IF NOT EXISTS` является расширением Postgres Pro.

См. также

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

CREATE SEQUENCE — создать генератор последовательности

Синтаксис

```
CREATE [ { TEMPORARY | TEMP } | UNLOGGED ] SEQUENCE [ IF NOT EXISTS ] имя
    [ AS тип_данных ]
    [ INCREMENT [ BY ] шаг ]
    [ MINVALUE мин_значение | NO MINVALUE ] [ MAXVALUE макс_значение | NO MAXVALUE ]
    [ START [ WITH ] начало ] [ CACHE кеш ] [ [ NO ] CYCLE ]
    [ OWNED BY { имя_таблицы.имя_столбца | NONE } ]
```

Описание

`CREATE SEQUENCE` создаёт генератор последовательности. Эта операция включает создание и инициализацию специальной таблицы *имя*, содержащей одну строку. Владельцем генератора будет пользователь, выполняющий эту команду.

Если указано имя схемы, последовательность создаётся в заданной схеме, в противном случае — в текущей. Временные последовательности существуют в специальной схеме, так что при создании таких последовательностей имя схемы задать нельзя. Имя последовательности должно отличаться от имён других отношений (таблиц, последовательностей, индексов, представлений, материализованных представлений или сторонних таблиц), уже существующих в этой схеме.

После создания последовательности работать с ней можно, вызывая функции `nextval`, `currval` и `setval`. Эти функции документированы в Разделе 9.17.

Хотя непосредственно изменить значение последовательности нельзя, получить её параметры и текущее состояние можно таким запросом:

```
SELECT * FROM name;
```

В частности, поле `last_value` последовательности будет содержать последнее значение, выделенное для какого-либо сеанса. (Конечно, ко времени вывода это значение может стать неактуальным, если другие сеансы активно вызывают `nextval`.)

Параметры

`TEMPORARY` или `TEMP`

Если указано, объект последовательности создаётся только для данного сеанса и автоматически удаляется при завершении сеанса. Существующая постоянная последовательность с тем же именем не будут видна (в этом сеансе), пока существует временная, однако к ней можно обратиться, дополнив имя указанием схемы.

`UNLOGGED`

С этим указанием последовательность создаётся как нежурналируемая. Изменения нежурналируемых последовательностей не проходят через журнал предзаписи. Такие последовательности не защищены от сбоев: нежурналируемая последовательность автоматически сбрасывается в исходное состояние после сбоя или аварийного отключения сервера. Кроме того, нежурналируемые последовательности не реплицируются на ведомые серверы.

В отличие от нежурналируемых таблиц, нежурналируемые последовательности не дают значительного выигрыша в производительности. Это указание полезно, в первую очередь, когда последовательность связана через столбцы идентификаторов или столбцы serial с нежурналируемыми таблицами. В таких случаях обычно не имеет смысла журналировать и реплицировать саму последовательность, но не таблицу, с которой она связана.

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с последовательностью, которая могла бы быть создана — это может быть даже не последовательность.

`имя`

Имя создаваемой последовательности (возможно, дополненное схемой).

`тип_данных`

Необязательное предложение `AS` *`тип_данных`* задаёт тип данных для последовательности. Допустимые типы: `smallint`, `integer` и `bigint`. По умолчанию устанавливается тип `bigint`. От типа данных зависят принимаемые по умолчанию минимальное и максимальное значения последовательности.

`шаг`

Необязательное предложение `INCREMENT BY` *`шаг`* определяет, какое число будет добавляться к текущему значению последовательности для получения нового значения. С положительным шагом последовательность будет возрастающей, а с отрицательным — убывающей. Значение по умолчанию: 1.

`мин_значение`
`NO MINVALUE`

Необязательное предложение `MINVALUE` *`мин_значение`* определяет наименьшее число, которое будет генерировать последовательность. Если это предложение опущено либо указано `NO MINVALUE`, используется значение по умолчанию: 1 для возрастающей последовательности или минимальное значение типа данных — для убывающей.

`макс_значение`
`NO MAXVALUE`

Необязательное предложение `MAXVALUE` *`макс_значение`* определяет наибольшее число, которое будет генерировать последовательность. Если это предложение опущено либо указано `NO MAXVALUE`, используется значение по умолчанию: максимальное значение типа данных для возрастающей последовательности или -1 — для убывающей.

`начало`

Необязательное предложение `START WITH` *`начало`* позволяет запустить последовательность с любого значения. По умолчанию началом считается *`мин_значение`* для возрастающих последовательностей и *`макс_значение`* для убывающих.

`кеш`

Необязательное предложение `CACHE` *`кеш`* определяет, сколько чисел последовательности будет выделяться и сохраняться в памяти для ускорения доступа к ним. Минимальное значение равно 1 (за один раз генерируется только одно значение, т. е. кеширования нет), и оно же предполагается по умолчанию.

`CYCLE`
`NO CYCLE`

Параметр `CYCLE` позволяет зациклить последовательность при достижении *`макс_значения`* или *`мин_значения`* для возрастающей или убывающей последовательности, соответственно. Когда этот предел достигается, следующим числом этих последовательностей будет соответственно *`мин_значение`* или *`макс_значение`*.

Если указывается `NO CYCLE`, при каждом вызове `nextval` после достижения предельного значения будет возникать ошибка. Если указания `CYCLE` и `NO CYCLE` отсутствуют, по умолчанию предполагается `NO CYCLE`.

`OWNED BY` *`имя_таблицы.имя_столбца`*
`OWNED BY NONE`

Предложение `OWNED BY` позволяет связать последовательность с определённым столбцом таблицы так, чтобы при удалении этого столбца (или всей таблицы) последовательность удалялась автоматически. Указанная таблица должна иметь того же владельца и находиться в той же схеме, что и последовательность. Подразумеваемое по умолчанию предложение `OWNED BY NONE` указывает, что такая связь не устанавливается.

Примечания

Для удаления последовательности применяется команда `DROP SEQUENCE`.

Последовательности основаны на арифметике `bigint`, так что их значения не могут выходить за диапазон восьмибайтовых целых (-9223372036854775808 .. 9223372036854775807).

Так как вызовы `nextval` и `setval` никогда не откатываются, объекты последовательностей не подходят, если требуется обеспечить непрерывное назначение номеров последовательностей. Непрерывное назначение можно организовать, используя исключительную блокировку таблицы со счётчиком; однако это решение будет гораздо дороже, чем применение объектов последовательностей, особенно когда последовательные номера будут затребоваться сразу многими транзакциями.

Если значение параметра *`кеш`* больше единицы, и объект последовательности используется параллельно в нескольких сеансах, результат может оказаться не вполне ожидаемым. Каждый сеанс будет выделять и кешировать несколько очередных значений последовательности при одном обращении к объекту последовательности и соответственно увеличивать `последнее_значение` этого объекта. Затем при следующих *`кеш`*-1 вызовах `nextval` в этом сеансе будет просто возвращать заготовленные значения, не касаясь объекта последовательности. В результате, все числа, выделенные, но не использованные в сеансе, будут потеряны при завершении сеанса, что приведёт к образовании «дырок» в последовательности.

Более того, хотя разным сеансам гарантированно выделяются различные значения последовательности, если рассмотреть все сеансы в целом, порядок этих значений может быть нарушен. Например, при значении *`кеш`*, равном 10, сеанс A может зарезервировать значения 1..10 и получить `nextval`=1, затем сеанс B может зарезервировать значения 11..20 и получить `nextval`=11 до того, как в сеансе A сгенерируется `nextval`=2. Таким образом, при значении *`кеш`*, равном одному, можно быть уверенными в том, что `nextval` генерирует последовательные значения; но если *`кеш`* больше одного, рассчитывать можно только на то, что все значения `nextval` различны; их порядок может быть непоследовательным. Кроме того, `last_value` возвращает последнее зарезервированное значение для всех сеансов, вне зависимости от того, было ли оно уже возвращено функцией `nextval`.

Также следует учитывать, что действие функции `setval`, выполненной для такой последовательности, проявится в других сеансах только после того, как в них будут использованы все предварительно закешированные значения.

Примеры

Создание возрастающей последовательности с именем `serial`, с начальным значением 101:

```
CREATE SEQUENCE serial START 101;
```

Получение следующего номера этой последовательности:

```
SELECT nextval('serial');

 nextval
---------
     101
```

Получение следующего номера этой последовательности:

```
SELECT nextval('serial');
```

```
 nextval
---------
     102
```

Использование этой последовательности в команде `INSERT`:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Изменение значения последовательности после `COPY FROM`:

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

Совместимость

Команда `CREATE SEQUENCE` соответствует стандарту SQL, со следующими исключениями:

- Для получения следующего значения применяется функция `nextval()`, а не выражение `NEXT VALUE FOR`, как того требует стандарт.
- Предложение `OWNED BY` является расширением Postgres Pro.

См. также

ALTER SEQUENCE, DROP SEQUENCE

CREATE SERVER

CREATE SERVER — создать сторонний сервер

Синтаксис

```
CREATE SERVER [ IF NOT EXISTS ] имя_сервера [ TYPE 'тип_сервера' ] [ VERSION
 'server_version' ]
    FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Описание

`CREATE SERVER` создаёт сторонний сервер. Владельцем сервера становится создавший его пользователь.

Определение стороннего сервера обычно включает информацию о подключении, которую использует обёртка сторонних данных для доступа к внешнему ресурсу. Определяя сопоставления пользователей, можно установить и другие параметры подключения, связанные с пользователями.

Имя сервера должно быть уникальным в базе данных.

Для создания сервера требуется право `USAGE` для обёртки сторонних данных.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если сервер с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующий сервер как-то соотносится с тем, который мог бы быть создан.

`имя_сервера`

Имя создаваемого стороннего сервера.

`тип_сервера`

Необязательный тип сервера, может быть полезен для обёрток сторонних данных.

`server_version`

Необязательная версия сервера, может быть полезна для обёрток сторонних данных.

`имя_обёртки_сторонних_данных`

Имя обёртки сторонних данных, управляющей сервером.

`OPTIONS ( параметр 'значение' [, ... ] )`

Это предложение определяет параметры сервера. Эти параметры обычно задают свойства подключения к серверу; их конкретные имена и значения зависят от обёртки сторонних данных.

Примечания

При использовании модуля dblink имя стороннего сервера может служить аргументом функции dblink_connect, определяющим параметры подключения. Для такого варианта использования необходимо иметь право `USAGE` для стороннего сервера.

Если сторонний сервер поддерживает вынос сортировки наружу, необходимо, чтобы он имел тот же порядок сортировки, что и локальный сервер.

Примеры

Создание сервера `myserver`, доступного через обёртку `postgres_fdw`:

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'foo', dbname
 'foodb', port '5432');
```

За подробностями обратитесь к postgres_fdw.

Совместимость

`CREATE SERVER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED).

См. также

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE, CREATE USER MAPPING

CREATE STATISTICS

CREATE STATISTICS — создать расширенную статистику

Синтаксис

```
CREATE STATISTICS [ [ IF NOT EXISTS ] имя_статистики ]
    ON ( выражение )
    FROM имя_таблицы

CREATE STATISTICS [ [ IF NOT EXISTS ] имя_статистики ]
    [ ( вид_статистики [, ... ] ) ]
    ON { имя_столбца | ( выражение ) }, { имя_столбца | ( выражение ) } [, ...]
    FROM имя_таблицы
```

Описание

Команда `CREATE STATISTICS` создаст новый объект расширенной статистики, отслеживающий данные определённой таблицы, сторонней таблицы или материализованного представления. Объект статистики будет создан в текущей базе данных, и его владельцем станет пользователь, выполняющий команду.

Команда `CREATE STATISTICS` имеет две основные формы. Первая форма позволяет собрать одномерную статистику для одного выражения, что даёт примерно тот же положительный эффект, что и индекс по выражению, но не требует поддержания индекса. Данная форма не позволяет указать вид статистики, так как различные виды существуют только для многомерных статистик. Вторая форма позволяет собрать многомерные статистики по нескольким столбцам и/или выражениям с возможностью выбора определённых видов статистики. Эта форма также автоматически собирает одномерные статистики по всем отдельным выражениям в списке.

Если задано имя схемы (например, `CREATE STATISTICS myschema.mystat ...`), объект статистики создаётся в указанной схеме, в противном случае — в текущей. Если имя объекта статистики задано, оно должно отличаться от имён других объектов статистики в этой схеме.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если объект статистики с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что при этом проверяется только имя объекта, а не характеристики его определения. Имя объекта статистики является обязательным, когда указывается `IF NOT EXISTS`.

`имя_статистики`

Имя создаваемого объекта статистики (возможно, дополненное схемой). Если имя опущено, Postgres Pro формирует подходящее имя по имени родительской таблицы и именам столбцов и/или выражений.

`вид_статистики`

Вид многовариантной статистики, который будет вычисляться в этом объекте. В настоящее время поддерживаются следующие виды: `ndistinct` (подсчёт числа различных значений), `dependencies` (определение функциональных зависимостей) и `mcv` (списки самых частых значений). Если это предложение опущено, в объект статистики включаются все поддерживаемые виды статистики. Одномерная статистика по выражениям собирается автоматически, если в определении статистики присутствуют сложные выражения, а не просто ссылки на столбцы. За подробностями обратитесь к Подразделу 14.2.2 и Разделу 65.2.

имя_столбца

Имя столбца таблицы, который будет покрываться вычисляемой статистикой. Это указание принимается только формой построения многомерной статистики. Необходимо указать минимум два выражения или столбца; их порядок не имеет значения.

выражение

Выражение, которое будет покрываться вычисляемой статистикой. Это указание может задаваться для построения многомерной статистики или быть частью списка столбцов и/или выражений, для которых строится многомерная статистика. В последнем случае для каждого выражения в списке автоматически строятся отдельные одномерные статистики.

имя_таблицы

Имя (возможно, дополненное схемой) таблицы, содержащей столбцы, по которым создаётся статистика; об особенностях, связанных с наследованием и секционированием, рассказывается в описании ANALYZE.

Примечания

Чтобы создать объект статистики, читающий таблицу, необходимо быть владельцем этой таблицы. После создания объекта статистики его владелец может определяться независимо от нижележащей таблицы.

Статистики выражений рассчитываются по отдельным выражениям подобно тому, как строится индекс по выражению, но не требуют таких накладных расходов, как индексы. Статистики по выражениям строятся автоматически для каждого выражения, входящего в определение объекта статистики.

В настоящее время планировщик не использует расширенную статистику для оценок избирательности, выполненных для соединений таблиц. Это ограничение, скорее всего, будет снято в одной из следующих версий Postgres Pro.

Примеры

Создайте таблицу `t1` с двумя функционально зависимыми столбцами; то есть знания значения одного столбца достаточно, чтобы определить значение другого. Затем для этих столбцов постройте статистику функциональной зависимости:

```
CREATE TABLE t1 (
    a   int,
    b   int
);

INSERT INTO t1 SELECT i/100, i/500
                 FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- число совпадающих строк будет катастрофически недооценено:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;

ANALYZE t1;

-- теперь оценка числа строк стала точнее:
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

Без статистики функциональной зависимости планировщик предположил бы, что два условия `WHERE` независимы друг от друга, и перемножил бы их оценки избирательности, что дало бы слишком заниженную оценку числа строк. Однако с созданной статистикой планировщик понимает, что условия `WHERE` избыточны, и не ошибается с этой оценкой.

Создайте таблицу `t2` с двумя идеально коррелирующими столбцами (содержащими одинаковые данные), а затем по этим столбцам постройте статистику MCV:

```
CREATE TABLE t2 (
    a   int,
    b   int
);

INSERT INTO t2 SELECT mod(i,100), mod(i,100)
                 FROM generate_series(1,1000000) s(i);

CREATE STATISTICS s2 (mcv) ON a, b FROM t2;

ANALYZE t2;

-- подходящая комбинация (входит в MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 1);

-- неподходящая комбинация (не входит в MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 2);
```

Список значений MCV даёт планировщику более точное представление о самых частых значениях в таблице, а также верхнюю границу избирательности для комбинаций, отсутствующих в ней, благодаря чему он может выработать более точные оценки в обоих случаях.

Создайте таблицу `t3` с одним полем timestamp и выполните запросы, содержащие выражения с этим столбцом. Без расширенной статистики планировщик не имеет информации о распределении данных для этих выражений и использует оценки по умолчанию. Планировщик также не понимает, что значение даты, усечённое до месяцев, полностью определяется значением даты, усечённой до дней. Затем постройте статистики выражений и статистику ndistinct по этим выражениям:

```
CREATE TABLE t3 (
    a   timestamp
);

INSERT INTO t3 SELECT i FROM generate_series('2020-01-01'::timestamp,
                                             '2020-12-31'::timestamp,
                                             '1 minute'::interval) s(i);

ANALYZE t3;

-- оценки количества совпадающих строк будут существенно занижены:
EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('month', a) = '2020-01-01'::timestamp;

EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('day', a) BETWEEN '2020-01-01'::timestamp
                                 AND '2020-06-30'::timestamp;

EXPLAIN ANALYZE SELECT date_trunc('month', a), date_trunc('day', a)
   FROM t3 GROUP BY 1, 2;

-- построить статистику ndistinct по двум выражениям (статистика по
-- отдельным выражениям строится автоматически)
CREATE STATISTICS s3 (ndistinct) ON date_trunc('month', a), date_trunc('day', a) FROM
 t3;
```

```
ANALYZE t3;

-- теперь оценки количества стали более точными:
EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('month', a) = '2020-01-01'::timestamp;

EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('day', a) BETWEEN '2020-01-01'::timestamp
                                 AND '2020-06-30'::timestamp;

EXPLAIN ANALYZE SELECT date_trunc('month', a), date_trunc('day', a)
   FROM t3 GROUP BY 1, 2;
```

Без статистики выражений и ndistinct планировщик не имеет сведений о количестве различных значений по выражениям и поэтому должен полагаться на оценки по умолчанию. При этом предполагается, что условия равенства и вхождения в диапазон имеют избирательность 0.5%, а число различных значений в выражении предполагается таким же, как и в столбце (то есть предполагаются уникальные значения). Вследствие этого оценка количества строк в первых двух строках оказывается существенно заниженной. Более того, планировщик не имеет информации о связанности выражений и поэтому он предполагает, что условия `WHERE` и `GROUP BY` независимы, и перемножает их избирательности, что даёт значительно завышенную оценку количества групп в запросе с агрегированием. Это ещё больше усугубляется нехваткой точной статистики для выражений, вследствие чего планировщик вынужден использовать для выражения оценку ndistinct по умолчанию, полученную из статистики ndistinct для столбца. Однако с новой статистикой планировщик понимает, что условия связаны, и даёт гораздо более точные оценки.

Совместимость

Команда `CREATE STATISTICS` отсутствует в стандарте SQL.

См. также

ALTER STATISTICS, DROP STATISTICS

CREATE SUBSCRIPTION

CREATE SUBSCRIPTION — создать подписку

Синтаксис

```
CREATE SUBSCRIPTION имя_подписки
    CONNECTION 'строка_подключения'
    PUBLICATION имя_публикации [, ...]
    [ WITH ( параметр_подписки [= значение] [, ... ] ) ]
```

Описание

`CREATE SUBSCRIPTION` создаёт подписку логической репликации. Пользователь, создающий подписку, становится её владельцем. Имя подписки должно отличаться от имён других существующих подписок в текущей базе.

Подписка представляет собой реплицирующее подключение к публикующему серверу. Поэтому данная команда не только добавляет определения подписки в локальные каталоги, но также создаёт слот репликации на удалённом сервере.

В момент фиксации транзакции, в рамках которой выполняется эта команда, будет запущен рабочий процесс логической репликации, если подписка создаётся не в отключённом состоянии.

Чтобы создать подписку, необходимо иметь права роли `pg_create_subscription`, а также право `CREATE` для текущей базы данных.

Дополнительные сведения о подписках и логической репликации в целом можно найти в Разделе 28.2 и Главе 28.

Параметры

`имя_подписки`

Имя новой подписки.

`CONNECTION 'строка_подключения'`

Строка подключения libpq, определяющая, как подключаться к базе данных публикации. За подробностями обратитесь к Подразделу 32.1.1.

`PUBLICATION имя_публикации [, ...]`

Имена публикаций на публикующем сервере, на которые оформляется подписка.

`WITH ( параметр_подписки [= значение] [, ... ] )`

В этом предложении задаются необязательные параметры подписки.

Следующие параметры определяют, как будет создаваться подписка:

`connect` (`boolean`)

Определяет, нужно ли при выполнении `CREATE SUBSCRIPTION` подключаться к публикующему серверу. Значение по умолчанию — `true`. Если равняется `false`, параметры `create_slot`, `enabled` и `copy_data` тоже принимают значение `false`. (Значение `false` параметра `connect` несовместимо со значением `true` параметров `create_slot`, `enabled` и `copy_data`.)

Так как со значением `false` соединение не устанавливается, подписка на таблицы не оформляется. Чтобы начать репликацию, потребуется вручную создать слот репликации, при необходимости включить `failover`, включить подписку и обновить её. Примеры можно найти в Подразделе 28.2.3.

`create_slot` (boolean)

Определяет, должна ли команда создавать слот репликации на публикующем сервере. Значение по умолчанию — `true`.

При значении `false` потребуется создать слот на публикующем сервере другим способом. Примеры можно найти в Подразделе 28.2.3.

`enabled` (boolean)

Определяет, активировать ли репликацию в подписке, или её нужно только настроить, но не запускать сразу. Значение по умолчанию — `true`.

`slot_name` (string)

Имя слота репликации на публикующем сервере, которое должно использоваться. По умолчанию в качестве имени слота используется имя подписки.

Когда в качестве `slot_name` задаётся `NONE`, с подпиской не будет связан слот репликации. У таких подписок также должны быть равны `false` свойства `enabled` и `create_slot`. Используйте это значение, если будете создавать слот репликации позднее вручную. Примеры можно найти в Подразделе 28.2.3.

Если задать в качестве `slot_name` корректное имя, а для `create_slot` указать `false`, значение `failover` указанного слота может отличаться от значения `failover` в подписке. При создании слота убедитесь, что значение `failover` слота совпадает с указанным в подписке, иначе слот на публикующем сервере может вести себя неожиданно: например, слот на публикующем сервере может синхронизироваться с резервными серверами даже при выключенном в подписке параметре `failover` и не синхронизироваться даже при включённом в подписке параметре `failover`.

Следующие параметры управляют поведением репликации подписки после её создания:

`binary` (boolean)

Определяет, будет ли для данной подписки запрашиваться передача данных в двоичном (а не в текстовом) виде. Значение по умолчанию — `false`. Данные для начальной синхронизации таблиц копируются (см. `copy_data`) в том же формате. Двоичный формат может обрабатываться быстрее, чем текстовый, но он может оказаться непереносимым между разными машинными архитектурами и версиями Postgres Pro. Кроме того, двоичный формат сильно зависит от типов данных: например, он не позволяет копировать данные из столбца с типом `smallint` в `integer`, хотя с текстовым форматом это вполне возможно. Даже если этот параметр включён, в двоичном виде будут передаваться только те типы данных, для которых определены функции двоичного получения/отправки. Обратите внимание, что для начальной синхронизации требуется, чтобы для всех типов данных были определены функции двоичного получения/отправки, в противном случае синхронизация прервётся ошибкой (более подробно функции получения/отправки описаны в CREATE TYPE).

При репликации между серверами разных версий возможна ситуация, когда для некоторого типа на стороне публикации будет определена функция двоичной отправки, а на стороне подписки не будет соответствующей функции двоичного получения. В такой ситуации передача данных будет невозможна, и параметр `binary` использовать нельзя.

Если версия публикующего сервера ниже Postgres Pro 16, данные для начальной синхронизации таблиц будут передаваться в текстовом формате, даже если задать `binary = true`.

`copy_data` (boolean)

Определяет, должны ли копироваться уже существующие данные в публикациях, на которые оформляется подписка, сразу после начала репликации. Значение по умолчанию — `true`.

Предложения `WHERE`, содержащиеся в публикациях, влияют на то, какие данные будут скопированы. За подробностями обратитесь к Замечания.

В Замечания описано поведение при копировании данных (`copy_data = true`) с параметром `origin`.

`streaming` (`enum`)

Определяет, включать ли для данной подписки передачу изменений ещё выполняющихся транзакций. Значение по умолчанию — `off`, то есть каждая транзакция сначала целиком декодируется на стороне публикации и только затем передаётся подписчику как единое целое.

Если этот параметр включён (`on`), поступающие изменения записываются во временные файлы и применяются после фиксирования транзакции на публикующем сервере и получения данных подписчиком.

Если задано значение `parallel`, доступный параллельный рабочий процесс применяет поступающие изменения напрямую. Если нет свободного параллельного рабочего процесса для применения изменений передаваемых в потоке транзакций, изменения записываются во временные файлы и применяются после фиксирования транзакции. Обратите внимание, что в случае сбоя параллельного рабочего процесса LSN завершения удалённой транзакции может быть не записан в журнал сервера.

`synchronous_commit` (`enum`)

Значение этого параметра переопределяет свойство synchronous_commit для рабочих процессов, применяющих изменения данной подписки. По умолчанию — `off`.

Значение `off` безопасно для логической репликации: если подписчик потеряет транзакции из-за нарушения синхронизации, данные будут повторно переданы с публикующего сервера.

При выполнении синхронной логической репликации может быть уместно другое значение. Рабочие процессы логической репликации передают позиции записанных и сохранённых на диске данных публикующему серверу, так что при синхронной репликации он будет ждать завершения сохранения. Это значит, что значение `off` параметра `synchronous_commit` на подписчике может увеличить задержку при выполнении `COMMIT` на сервере публикации. При таком сценарии может быть выгоднее задать для `synchronous_commit` значение `local` или выше.

`two_phase` (`boolean`)

Определяет, включается ли двухфазная фиксация для этой подписки. Значение по умолчанию — `false`.

Когда двухфазная фиксация включена, подготовленные транзакции отправляются подписчику во время `PREPARE TRANSACTION` и также обрабатываются подписчиком как двухфазные транзакции. В противном случае подготовленные транзакции отправляются подписчику только после фиксации, а затем немедленно обрабатываются подписчиком.

Реализация двухфазной фиксации требует, чтобы процедура репликации успешно завершила начальную фазу синхронизации таблиц. Таким образом, даже если для подписки включён параметр `two_phase`, внутренним состоянием двухфазной фиксации временно остаётся «в ожидании», пока не завершится фаза инициализации. Фактическое состояние двухфазной фиксации можно увидеть в столбце `subtwophasestate` в `pg_subscription`.

`disable_on_error` (`boolean`)

Определяет, следует ли автоматически отключать подписку, если рабочие процессы подписчика обнаружат какие-либо ошибки при репликации данных публикации. Значение по умолчанию — `false`.

`password_required` (`boolean`)

Если задано значение `true`, подключения к публикующему серверу, создаваемые в результате подписки, должны требовать аутентификации по паролю, при этом пароль должен указываться как часть строки подключения. Этот параметр игнорируется, если владельцем подписки является суперпользователь. Значение по умолчанию — `true`. Задать значение `false` могут только суперпользователи.

`run_as_owner` (`boolean`)

Если true, все действия репликации выполняются от имени владельца подписки. Если false (по умолчанию), рабочие процессы репликации выполняют операции с таблицами от имени владельца таблицы, что в целом намного более безопасно (за подробностями обратитесь к Разделу 28.10).

`origin` (`string`)

Определяет, будет ли для данной подписки запрашиваться передача изменений, не отмеченных источником репликации, или всех изменений независимо от источника. Если задать для `origin` значение `none`, для подписки будет запрашиваться передача изменений, не отмеченных источником репликации. Если задать для `origin` значение `any`, публикующий сервер отправляет все изменения независимо от их источника. По умолчанию — `any`.

В Замечания описано поведение при копировании данных (`copy_data = true`) с параметром `origin`.

`failover` (`boolean`)

Определяет, должны ли слоты репликации, связанные с подпиской, синхронизироваться с ведомым сервером, чтобы логическую репликацию можно было возобновить на новом ведущем сервере сразу после переключения. Значение по умолчанию — `false`.

Если для параметра типа `boolean` опустить `=` *`значение`*, это равнозначно указанию значения `TRUE`.

Замечания

Подробнее о том, как организовать управление доступом подписчиков к публикующему серверу, рассказывается в Разделе 28.10.

При создании слота репликации (поведение по умолчанию) `CREATE SUBSCRIPTION` нельзя выполнять внутри блока транзакции.

Создание подписки с подключением к тому же кластеру баз данных (например, для организации репликации между базами данных в одном кластере или в одной базе данных) будет успешным, только если слот репликации не создаётся той же командой. В противном случае команда `CREATE SUBSCRIPTION` зависнет. Чтобы оформить такую подписку, слот репликации нужно создать отдельно (воспользовавшись функцией `pg_create_logical_replication_slot` и передав ей имя модуля `pgoutput`) и создать подписку с параметром `create_slot = false`. Примеры можно найти в Подразделе 28.2.3. Это ограничение реализации может быть устранено в будущем выпуске.

Если для какой-либо таблицы в публикации указано предложение `WHERE`, строки, для которых *`выражение`* выдаёт значение `false` или `null`, не будут публиковаться. Когда в подписке есть несколько публикаций, в которых одна и та же таблица опубликована с разными предложениями `WHERE`, строка будет опубликована, если выполняется любое из выражений (относящееся к этой операции публикации). В случае неодинаковых предложений `WHERE`, если в одной из публикаций нет предложения `WHERE` (относящегося к этой операции публикации) или публикация объявлена как `FOR ALL TABLES` или `FOR TABLES IN SCHEMA`, строки всегда публикуются независимо от определения других выражений. Если на стороне подписчика используется версия Postgres Pro ниже 15, на этапе начальной синхронизации данных все фильтры строк игнорируются. В этом случае следует рассмотреть возможность удалить все изначально скопированные данные, которые будут несовместимы с фильтрами впоследствии. Так как при первоначальной синхронизации данных параметр публикации `publish` не учитывается, при копировании существующих данных могут быть скопированы строки, которые не были бы реплицированы при выполнении операций DML. Примеры приведены в Подразделе 28.2.2.

Создать подписки, связанные с несколькими публикациями, в которых одна и та же таблица опубликована с разными списками столбцов, нельзя.

При создании подписок допускается указание несуществующих публикаций, что позволяет создавать нужные публикации позже. Таким образом, в `pg_subscription` могут фигурировать несуществующие публикации.

Когда для подписки заданы параметры `copy_data = true` и `origin = NONE`, данные для начальной синхронизации копируются напрямую с публикующего сервера, то есть исходный источник данных определить невозможно. Если публикующий сервер сам является подписчиком, копируемые данные могли быть получены из другого источника. При выявлении такой ситуации пользователю выдаётся предупреждение, которое только указывает на возможную проблему, — пользователю необходимо самостоятельно убедиться в необходимости копировать данные из третьего источника.

Чтобы найти таблицы, данные в которых могли быть получены из третьего источника (из-за подписок на публикующем сервере), воспользуйтесь таким SQL-запросом:

```
# замените <pub-names> именами подписок для запроса
SELECT DISTINCT PT.schemaname, PT.tablename
FROM pg_publication_tables PT
     JOIN pg_class C ON (C.relname = PT.tablename)
     JOIN pg_namespace N ON (N.nspname = PT.schemaname),
     pg_subscription_rel PS
WHERE C.relnamespace = N.oid AND
      (PS.srrelid = C.oid OR
      C.oid IN (SELECT relid FROM pg_partition_ancestors(PS.srrelid) UNION
                SELECT relid FROM pg_partition_tree(PS.srrelid))) AND
      PT.pubname IN (<pub-names>);
```

Примеры

Создание подписки на репликации `mypublication` и `insert_only` на удалённом сервере с немедленным запуском репликации при фиксировании транзакции:

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION mypublication, insert_only;
```

Создание подписки на публикацию `insert_only` на удалённом сервере с отключением репликации для запуска в будущем.

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
        PUBLICATION insert_only
               WITH (enabled = false);
```

Совместимость

`CREATE SUBSCRIPTION` является расширением Postgres Pro.

См. также

ALTER SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

CREATE TABLE

CREATE TABLE — создать таблицу

Синтаксис

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] имя_таблицы ( [
  { имя_столбца тип_данных [ STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT } ]
 [ COMPRESSION метод_сжатия ] [ COLLATE правило_сортировки ] [ ограничение_столбца
 [ ... ] ]
    | ограничение_таблицы
    | LIKE таблица_источник [ параметр_LIKE ... ] }
    [, ... ]
] )
[ INHERITS ( таблица_родитель [, ... ] ) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { имя_столбца | ( выражение ) }
 [ COLLATE правило_сортировки ] [ класс_операторов ] [, ... ] ) ]
[ USING метод ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE табл_пространство ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] имя_таблицы
    OF имя_типа [ (
  { имя_столбца [ WITH OPTIONS ] [ ограничение_столбца [ ... ] ]
    | ограничение_таблицы }
    [, ... ]
) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { имя_столбца | ( выражение ) }
 [ COLLATE правило_сортировки ] [ класс_операторов ] [, ... ] ) ]
[ USING метод ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE табл_пространство ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] имя_таблицы
    PARTITION OF таблица_родитель [ (
  { имя_столбца [ WITH OPTIONS ] [ ограничение_столбца [ ... ] ]
    | ограничение_таблицы }
    [, ... ]
) ] { FOR VALUES указание_границ_секции | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( { имя_столбца | ( выражение ) }
 [ COLLATE правило_сортировки ] [ класс_операторов ] [, ... ] ) ]
[ USING метод ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE табл_пространство ]

Здесь ограничение_столбца:

[ CONSTRAINT имя_ограничения ]
{ NOT NULL |
  NULL |
```

```
  CHECK ( выражение ) [ NO INHERIT ] |
  DEFAULT выражение_по_умолчанию |
  GENERATED ALWAYS AS ( генерирующее_выражение ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( параметры_последовательности ) ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] параметры_индекса |
  PRIMARY KEY параметры_индекса |
  REFERENCES целевая_таблица [ ( целевой_столбец ) ] [ MATCH FULL | MATCH PARTIAL |
 MATCH SIMPLE ]
    [ ON DELETE ссылочное_действие ] [ ON UPDATE ссылочное_действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

и ограничение_таблицы:

[ CONSTRAINT имя_ограничения ]
{ CHECK ( выражение ) [ NO INHERIT ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] ( имя_столбца [, ... ] ) параметры_индекса |
  PRIMARY KEY ( имя_столбца [, ... ] ) параметры_индекса |
  EXCLUDE [ USING индексный_метод ] ( элемент_исключения WITH оператор
 [, ... ] ) параметры_индекса [ WHERE ( предикат ) ] |
  FOREIGN KEY ( имя_столбца [, ... ] ) REFERENCES целевая_таблица [ ( целевой_столбец
 [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE ссылочное_действие ] [ ON
 UPDATE ссылочное_действие ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

и параметр_LIKE:

{ INCLUDING | EXCLUDING } { COMMENTS | COMPRESSION | CONSTRAINTS | DEFAULTS | GENERATED
 | IDENTITY | INDEXES | STATISTICS | STORAGE | ALL }

и указание_границ_секции:

IN ( выражение_границ_секции [, ...] ) |
FROM ( { выражение_границ_секции | MINVALUE | MAXVALUE } [, ...] )
  TO ( { выражение_границ_секции | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS числовая_константа, REMAINDER числовая_константа )

параметры_индекса в ограничениях UNIQUE, PRIMARY KEY и EXCLUDE:

[ INCLUDE ( имя_столбца [, ... ] ) ]
[ WITH ( параметр_хранения [= значение] [, ... ] ) ]
[ USING INDEX TABLESPACE табл_пространство ]

элемент_исключения в ограничении EXCLUDE:

{ имя_столбца | ( выражение ) } [ COLLATE правило_сортировки ] [ класс_операторов
 [ ( параметр_класса_оп = значение [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST |
 LAST } ]

ссылочное_действие в ограничении FOREIGN KEY/REFERENCES:

{ NO ACTION | RESTRICT | CASCADE | SET NULL [ ( имя_столбца [, ... ] ) ] | SET DEFAULT
 [ ( имя_столбца [, ... ] ) ] }
```

Описание

`CREATE TABLE` создаёт новую, изначально пустую таблицу в текущей базе данных. Владельцем таблицы будет пользователь, выполнивший эту команду.

Если задано имя схемы (например, `CREATE TABLE myschema.mytable ...`), таблица создаётся в указанной схеме, в противном случае — в текущей. Временные таблицы существуют в специальной схеме, так что при создании таких таблиц имя схемы задать нельзя. Имя таблицы должно отличаться от имён других отношений (таблиц, последовательностей, индексов, представлений, материализованных представлений или сторонних таблиц) в этой схеме.

`CREATE TABLE` также автоматически создаёт составной тип данных, соответствующий одной строке таблицы. Таким образом, имя таблицы не может совпадать с именем существующего типа в этой же схеме.

Необязательные предложения ограничений задают ограничения (проверки), которым должны удовлетворять добавляемые или изменяемые строки, чтобы операция добавления или изменения была выполнена успешно. Ограничение представляет собой SQL-объект, помогающий некоторым способом определить множество допустимых значений в таблице.

Определить ограничения можно двумя способами: в виде ограничений таблицы и в виде ограничений столбца. Ограничение столбца определяется как часть определения столбца, а ограничение таблицы не привязывается к конкретному столбцу и может задействовать несколько столбцов. Любые ограничения столбцов можно также записать в виде ограничения таблицы, они введены просто для удобства записи в случаях, когда ограничение затрагивает только один столбец.

Чтобы создать таблицу, необходимо иметь право `USAGE` для типов всех столбцов или типа в предложении `OF`, соответственно.

Параметры

`TEMPORARY` или `TEMP`

С таким указанием таблица создаётся как временная. Временные таблицы автоматически удаляются в конце сеанса или могут удаляться в конце текущей транзакции (см. описание `ON COMMIT` ниже). В путь поиска по умолчанию первой включается временная схема, поэтому постоянная таблица с тем же именем не будет выбираться для новых планов запросов, пока существует временная (если имя задаётся без указания схемы). Все индексы, создаваемые для временной таблицы, автоматически также становятся временными.

Демон автоочистки не может прочитать и, как следствие, сжимать и анализировать временные таблицы. По этой причине соответствующие операции очистки и анализа следует выполнять, вызывая SQL-команды в рамках сеанса. Например, если временную таблицу планируется использовать в сложных запросах, будет разумным выполнить для неё `ANALYZE` после того, как она будет наполнена.

По желанию можно добавить указание `GLOBAL` или `LOCAL` перед `TEMPORARY` или `TEMP`. В настоящее время это не имеет значения для Postgres Pro и считается устаревшей возможностью; см. раздел Совместимость ниже.

`UNLOGGED`

С этим указанием таблица создаётся как нежурналируемая. Данные, записываемые в нежурналируемые таблицы, не проходят через журнал предзаписи (см. Главу 27), в результате чего такие таблицы работают гораздо быстрее обычных. Однако они не защищены от сбоя; при сбое или аварийном отключении сервера нежурналируемая таблица автоматически усекается. Кроме того, содержимое нежурналируемой таблицы не реплицируется на ведомые серверы. Любые индексы, создаваемые для нежурналируемой таблицы, автоматически становятся нежурналируемыми.

С этим указанием все последовательности, создаваемые вместе с нежурналируемой таблицей (для столбцов идентификаторов или столбцов serial), также создаются нежурналируемыми.

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее отношение как-то соотносится с тем, которое могло бы быть создано.

`имя_таблицы`

Имя создаваемой таблицы (возможно, дополненное схемой).

`OF` *`имя_типа`*

Создаёт *типизированную таблицу*, структура которой определяется указанным составным типом (его имя может быть дополнено схемой). Типизированная таблица привязана к породившему её типу; например, при удалении типа (командой `DROP TYPE ... CASCADE`) будет удалена и эта таблица.

Когда создаётся типизированная таблица, типы данных её столбцов определяются нижележащим составным типом, а не задаются командой `CREATE TABLE`. Но `CREATE TABLE` может добавить в таблицу значения по умолчанию и ограничения, а также задать параметры её хранения.

`имя_столбца`

Имя столбца, создаваемого в новой таблице.

`тип_данных`

Тип данных столбца (может включать определение массива с этим типом). За дополнительными сведениями о типах данных, которые поддерживает Postgres Pro, обратитесь к Главе 8.

`COLLATE` *`правило_сортировки`*

Предложение `COLLATE` назначает правило сортировки для столбца (который должен иметь тип, поддерживающий сортировку). Если оно отсутствует, используется правило сортировки по умолчанию, установленное для типа данных столбца.

`STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }`

Эта форма устанавливает режим хранения столбца. Она определяет, хранятся ли данные внутри таблицы или в отдельной таблице TOAST и сжимаются ли они. Режим `PLAIN` должен применяться для значений фиксированной длины, таких как `integer`; это вариант хранения внутри, без сжатия. Режим `MAIN` применяется для хранения внутри, но сжатых данных, `EXTERNAL` — для внешнего хранения несжатых данных, а `EXTENDED` — для внешнего хранения сжатых данных. С указанием `DEFAULT` будет использоваться режим по умолчанию для типа данных столбца. `EXTENDED` используется по умолчанию для большинства типов данных, поддерживающих хранилище не `PLAIN`. Применение `EXTERNAL` позволяет ускорить операции с подстроками на очень больших значениях `text` и `bytea`, за счёт проигрыша в объёме хранилища. За дополнительными сведениями обратитесь к Разделу 63.2.

`COMPRESSION` *`метод_сжатия`*

Указание `COMPRESSION` задаёт метод сжатия для столбца. Сжатие поддерживается только для типов данных переменной длины и применяется только для столбцов с режимом хранения `main` или `extended`. (О режимах хранения столбцов рассказывается в ALTER TABLE.) Заданное для секционированной таблицы это свойство не оказывает прямого действия, так как такие таблицы сами по себе не содержат данные, но установленное значение будут наследовать её новые секции. Поддерживаются следующие методы сжатия: `pglz` и `lz4`. (`lz4` поддерживается, только если Postgres Pro был собран с ключом `--with-lz4`.) Кроме того, *`метод_сжатия`* может принимать значение `default`, явно выбирающее поведение по умолчанию — когда применяемый метод определяется значением default_toast_compression во время добавления данных.

`INHERITS (` *`таблица_родитель`* `[, ... ] )`

Необязательное предложение `INHERITS` определяет список таблиц, от которых новая таблица будет автоматически наследовать все столбцы. Родительские таблицы могут быть обычными или сторонними таблицами.

При использовании `INHERITS` создаётся постоянная связь дочерней таблицы с родительскими. Изменения схемы в родительских таблицах обычно также отражаются в дочерних, и по умолчанию при чтении родительских таблиц в результат включаются данные дочерней таблицы.

Когда в нескольких родительских таблицах оказываются столбцы с одним именем, происходит ошибка, за исключением случая, когда типы данных всех этих столбцов в таблицах совпадают. В этом случае одноимённые столбцы объединяются и формируют один столбец в новой таблице. Если имя столбца новой таблицы совпадает с именем одного из унаследованных столбцов, их типы так же должны совпадать, и в этом случае определения столбцов тоже сливаются в одну. Если в новой таблице явно указывается значение по умолчанию для нового столбца, это значение переопределяет любые значения по умолчанию, унаследованные этим столбцом. В противном случае, если значения по умолчанию определяются в разных родительских таблицах, эти определения должны совпадать, иначе произойдёт ошибка.

Ограничения `CHECK` объединяются вместе по сути так же, как и столбцы: если несколько родительских таблиц и/или определение новой таблицы содержат одноимённые ограничения `CHECK`, этим ограничениям должны соответствовать одинаковые выражения проверки, в противном случае произойдёт ошибка. В случае совпадения выражения, эти ограничения с данным выражением объединяются в одно. При этом ограничения со свойством `NO INHERIT` в родительской таблице исключаются из рассмотрения. Заметьте, что безымянное ограничение `CHECK` в новой таблице никогда не сливается с другими, так как для него всегда выбирается уникальное имя.

Параметры `STORAGE` для столбца так же копируются из родительских таблиц.

Если столбец в родительской таблице является столбцом идентификации, это свойство не наследуется. Если требуется, в дочерней таблице этот столбец можно объявить столбцом идентификации.

`PARTITION BY { RANGE | LIST | HASH } ( {` *`имя_столбца`* `| (` *`выражение`* `) } [` *`класс_операторов`* `] [, ...] )`

Необязательное предложение `PARTITION BY` задаёт стратегию секционирования таблицы. Таблица, созданная с этим указанием, называется *секционируемой* таблицей. Задаваемый в скобках список столбцов или выражений формирует *ключ секционирования* таблицы. Для секционирования по диапазонам или по хешу ключ секционирования может включать несколько столбцов или выражений (до 32, но этот предел можно изменить при сборке Postgres Pro), но для секционирования по спискам ключ должен состоять из одного столбца или выражения.

Для секционирования по диапазонам и по спискам нужен класс операторов B-дерева, тогда как для секционирования по хешу требуется класс операторов хеширования. Если класс операторов не задан явно, будет применён класс операторов по умолчанию для соответствующего типа; в случае отсутствия такого класса выдаётся ошибка. Для секционирования по хешу применяемый класс операторов должен реализовывать опорную функцию 2 (см. Подраздел 36.16.3).

Секционируемая таблица разделяется на подтаблицы (называемые секциями), которые создаются отдельными командами `CREATE TABLE`. Сама по себе секционируемая таблица не содержит данных. Строка данных, вставляемая в эту таблицу, перенаправляется в секцию в зависимости от значений столбцов или выражений в ключе секционирования. Если значениям в новой строке не соответствует ни одна из существующих секций, возникает ошибка.

Узнать больше о секционировании таблиц можно в Разделе 5.12.

`PARTITION OF` *`таблица_родитель`* `{ FOR VALUES` *`указание_границ_секции`* `| DEFAULT }`

Создаёт таблицу в виде *секции* указанной родительской таблицы. Таблицу можно создать либо как секцию для определённых значений (используя `FOR VALUES`), либо как секцию по умолчанию (используя `DEFAULT`). Это указание неприемлемо для таблиц, секционируемых по хешу. В создаваемую секцию копируются все индексы, ограничения и пользовательские триггеры уровня строк.

Здесь *`указание_границ_секции`* должно соответствовать методу и ключу секционирования родительской таблицы и не должно конфликтовать с любой существующей секцией того же родителя. Вариант указания с `IN` используется для секционирования по спискам, тогда как вариант с `FROM` и `TO` для секционирования по диапазонам, а с `WITH` — для секционирования по хешу.

выражение_границ_секции — любое выражение без переменных (подзапросы, оконные, агрегатные и возвращающие множества функции в нём не допускаются). Его тип данных должен подходить для соответствующего столбца в ключе секционирования. Это выражение вычисляется единожды во время создания таблицы, поэтому в нём могут вызываться даже изменчивые функции, например `CURRENT_TIMESTAMP`.

При создании секции с секционированием по спискам возможно указать, что столбец ключа секционирования может содержать NULL, включив в список секции `NULL`. Однако у отдельно взятой родительской таблицы может быть не больше одной такой секции. Для диапазонных секций `NULL` задать нельзя.

При создании диапазонной секции нижняя граница, задаваемая во `FROM`, включается в диапазон, а верхняя граница, задаваемая в `TO` — исключается. То есть, значения, задаваемые в списке `FROM`, являются допустимыми значениями соответствующих столбцов ключа секционирования для этой секции, а значения в списке `TO` — нет. Заметьте, что это утверждение должно восприниматься с учётом правил сравнения строк таблицы (см. Подраздел 9.25.5). Например, с секционированием `PARTITION BY RANGE (x,y)`, секция с границами `FROM (1, 2) TO (3, 4)` примет `x=1` с любым значением `y>=2`, `x=2` с любым `y`, отличным от NULL, и `x=3` с любым `y<4`.

Специальные значения `MINVALUE` и `MAXVALUE` могут использоваться при создании диапазонной секции для указания, что нижняя или верхняя граница для значений столбца отсутствует. Например, секция, определённая с указанием `FROM (MINVALUE) TO (10)`, будет принимать любые значения меньше 10, а секция, определённая с указанием `FROM (10) TO (MAXVALUE)`, — любые значения, которые больше или равны 10.

При создании диапазонной секции с более чем одним столбцом может также иметь смысл использовать `MAXVALUE` в определении нижней границы, а `MINVALUE` — верхней. Например, секция, определённая с указанием `FROM (0, MAXVALUE) TO (10, MAXVALUE)`, будет принимать любые строки, в которых первый столбец ключа секционирования больше 0 и меньше или равен 10. Подобным образом, секция, определённая с указанием `FROM ('a', MINVALUE) TO ('b', MINVALUE)`, будет принимать строки, в которых первый столбец ключа секционирования начинается с «a».

Заметьте, что если для одного столбца в границе секции задаётся `MINVALUE` или `MAXVALUE`, то же значение должно применяться и для всех последующих столбцов. Например, граница `(10, MINVALUE, 0)` будет некорректной; допустимая граница: `(10, MINVALUE, MINVALUE)`.

Также заметьте, что для некоторых типов элементов, таких как `timestamp`, наряду с другими значениями допускается значение "infinity" (бесконечность). Оно отличается от вариантов `MINVALUE` и `MAXVALUE`, которые на самом деле не обозначают значения, которые можно сохранить, а просто говорят о том, это значение не ограничено. `MAXVALUE` можно воспринимать как значение, которое больше любого другого, включая "бесконечность", а `MINVALUE` меньше любого другого значения, включая "минус бесконечность". Таким образом, диапазон `FROM ('infinity') TO (MAXVALUE)` не будет пустым, а будет принимать ровно одно значение — «infinity».

С указанием `DEFAULT` таблица присоединяется к родительской таблице как секция по умолчанию. Для таблиц, секционируемых по хешу, это указание не поддерживается. Ключ секционирования, не попадающий ни в одну из секций данного родителя, будет отправлен в секцию по умолчанию.

Когда у таблицы есть секция по умолчанию (`DEFAULT`) и к ней добавляется новая секция, требуется просканировать секцию по умолчанию и убедиться в том, что она не содержит строки, которые должны относиться к новой секции. Если она содержит большое количество строк, это сканирование может быть длительным. Сканирование не будет выполняться, если секция по умолчанию является сторонней таблицей или в ней есть ограничение, гарантирующее отсутствие в этой секции строк, подлежащих перемещению в новую секцию.

При создании секции с секционированием по хешу должен задаваться модуль и остаток. Модулем должно быть положительное число, а остатком неотрицательное число, меньшее модуля. Обычно при начальной настройке таблицы с секционированием по хешу нужно выбрать модуль, равный количеству секций, и назначить каждой секции этот модуль и разные остатки (см. примеры ниже). Однако секциям можно назначить и разные модули, с условием, что модули, назначенные секциям таблицы, секционируемой по хешу, являются делителями следующих больших модулей. Это позволяет постепенно увеличивать число секций, не производя полное перемещение всех данных. Например, предположим, что у вас есть таблица, секционируемая по хешу на 8 секций, для каждой из которых назначен модуль 8, и возникла необходимость увеличить число секций до 16. Вы можете отсоединить одну из секций по модулю 8, создать две новые секции по модулю 16, покрывающих ту же часть пространства ключа (одну с остатком, равным остатку отсоединённой секции, а вторую с остатком, равным тому же остатку плюс 8), и вновь наполнить их данными. Вы можете повторять эту операцию (возможно, позже) для остальных секций по модулю 8, пока все они не будут заменены. Хотя и при таком подходе может потребоваться перемещать большие объёмы данных на каждом этапе, это всё же лучше, чем создавать абсолютно новую таблицу и перемещать все данные сразу.

В секции должны содержаться столбцы с теми же именами и типами, что и в секционированной таблице, к которой она относится. Изменение имён и типов столбцов в секционируемой таблице будет автоматически распространяться во все секции. Ограничения `CHECK` будут наследоваться автоматически всеми секциями, но для отдельных секций могут быть заданы дополнительные ограничения `CHECK`; дополнительные ограничения с теми же именами и условиями, как в родительской таблице, будут объединены с родительским ограничением. Также независимо для каждой секции могут быть заданы значения по умолчанию. Но заметьте, что значение по умолчанию, заданное на уровне секции, не будет действовать при добавлении строк через секционированную таблицу.

Строки, добавляемые в секционированную таблицу, будут автоматически перенаправляться в соответствующую секцию. Если подходящей секции не найдётся, произойдёт ошибка.

Такие операции, как `TRUNCATE`, обычно затрагивают и саму таблицу, и каскадно распространяются на все дочерние секции, но могут также выполняться в отдельных секциях.

Обратите внимание, что для создания секции с использованием `PARTITION OF` требуется применение блокировки `ACCESS EXCLUSIVE` для родительской секционированной таблицы. Аналогично для удаления секции с помощью `DROP TABLE` требуется установить блокировку `ACCESS EXCLUSIVE` в родительской таблице. Для выполнения этих операций с менее строгой блокировкой можно использовать команду `ALTER TABLE ATTACH/DETACH PARTITION`, тем самым снизив влияние на параллельные операции с секционированной таблицей.

`LIKE` *`таблица_источник`* `[` *`параметр_LIKE`* `... ]`

Предложение `LIKE` определяет таблицу, из которой в новую таблицу будут автоматически скопированы все имена столбцов, их типы данных и их ограничения на NULL.

В отличие от `INHERITS`, новая и исходная таблица становятся полностью независимыми после завершения создания. Изменения в исходной таблице не отражаются в новой, а данные новой таблицы не включаются в результат чтения исходной.

Кроме того, в отличие от `INHERITS`, столбцы и ограничения, копируемые командой `LIKE`, не объединяются с одноимёнными столбцами и ограничениями. Если дублирующееся имя указывается явно или возникает в другом предложении `LIKE`, происходит ошибка.

Необязательные предложения *`параметр_LIKE`* указывают, какие дополнительные свойства исходной таблицы будут копироваться. Указание `INCLUDING` копирует заданное свойство, а `EXCLUDING` исключает его. По умолчанию подразумевается `EXCLUDING`. Если к одному типу объекта относятся несколько указаний, будет применяться последнее. Допустимые указания:

`INCLUDING COMMENTS`

Копировать комментарии для скопированных столбцов, ограничений и индексов. По умолчанию комментарии не копируются, вследствие чего скопированные столбцы и ограничения в новой таблице оказываются без комментариев.

`INCLUDING COMPRESSION`

Копировать метод сжатия для столбцов. По умолчанию указания метода сжатия не копируются, вследствие чего для новых столбцов выбирается метод сжатия по умолчанию.

`INCLUDING CONSTRAINTS`

Копировать ограничения-проверки (`CHECK`). В данном контексте ограничения на уровне столбцов и на уровне таблицы не различаются. Ограничения NOT NULL копируются в новую таблицу всегда.

`INCLUDING DEFAULTS`

Копировать выражения значений по умолчанию в определениях копируемых столбцов. Без этого указания выражения по умолчанию не копируются, вследствие чего в новой таблице скопированные столбцы получают значения по умолчанию NULL. Заметьте, что при копировании подобных выражений, в которых вызываются функции, модифицирующие БД, как например `nextval`, может образовываться функциональная связь исходной таблицы с новой.

`INCLUDING GENERATED`

Копировать выражения, генерирующие значения для копируемых столбцов. По умолчанию все новые столбцы будут обычными базовыми столбцами.

`INCLUDING IDENTITY`

Копировать характеристики идентификации в определениях копируемых столбцов. Для каждого столбца идентификации в новой таблице создаётся новая последовательность, не зависящая от последовательностей, связанных со старой таблицей.

`INCLUDING INDEXES`

Создавать в новой таблице индексы, ограничения `PRIMARY KEY`, `UNIQUE` и `EXCLUDE`, существующие в исходной таблице. Имена для новых индексов и ограничений выбираются согласно стандартным правилам, независимо от того, как назывались исходные. (Это позволяет избежать потенциальных конфликтов с именами новых индексов.)

`INCLUDING STATISTICS`

Копировать в новую таблицу расширенную статистику.

`INCLUDING STORAGE`

Копировать параметры `STORAGE` в определениях копируемых столбцов. По умолчанию параметры `STORAGE` исключаются, вследствие чего скопированные столбцы в новой таблице получают параметры по умолчанию, определённые соответствующим типом. Подробнее параметры `STORAGE` описаны в Разделе 63.2.

`INCLUDING ALL`

Указание `INCLUDING ALL` является сокращённым вариантом выбора всех имеющихся отдельных параметров. (После `INCLUDING ALL` можно дополнительно добавить предложения `EXCLUDING`, чтобы выбрать все параметры, за исключением некоторых.)

Предложение `LIKE` может также применяться для копирования определений столбцов из представлений, сторонних таблиц и составных типов. Неприменимые параметры (например, `INCLUDING INDEXES` для представления) при этом игнорируются.

`CONSTRAINT` *имя_ограничения*

Необязательное имя столбца или ограничения таблицы. При нарушении ограничения его имя будет выводиться в сообщении об ошибках, так что имена ограничений вида `столбец должен быть положительным` могут сообщить полезную информацию об ограничении клиентскому приложению. (Имена ограничений, включающие пробелы, необходимо заключать в двойные кавычки.) Если имя ограничения не указано, система генерирует имя автоматически.

`NOT NULL`

Данный столбец не принимает значения NULL.

`NULL`

Данный столбец может содержать значения NULL (по умолчанию).

Это предложение предназначено только для совместимости с нестандартными базами данных SQL. Использовать его в новых приложениях не рекомендуется.

`CHECK ( ` *`выражение`* ` ) [ NO INHERIT ]`

В ограничении `CHECK` задаётся выражение, возвращающее логический результат, по которому определяется, будет ли успешна операция добавления или изменения для конкретных строк. Операция выполняется успешно, если результат выражения равен TRUE или UNKNOWN. Если же для какой-нибудь строки, задействованной в операции добавления или изменения, будет получен результат FALSE, возникает ошибка, и эта операция не меняет ничего в базе данных. Ограничение-проверка, заданное как ограничение столбца, должно ссылаться только на значение самого столбца, тогда как ограничение на уровне таблицы может ссылаться и на несколько столбцов.

В настоящее время выражения `CHECK` не могут содержать подзапросы или ссылаться на какие-либо переменные, кроме как на столбцы текущей строки (см. Подраздел 5.5.1). Также допустима ссылка на системный столбец `tableoid`, но не на другие системные столбцы.

Ограничение с пометкой `NO INHERIT` не будет наследоваться дочерними таблицами.

Когда для таблицы задано несколько ограничений `CHECK`, они будут проверяться для каждой строки в алфавитном порядке имён после проверки ограничений `NOT NULL`. (До версии 9.5 в PostgreSQL не было установлено никакого определённого порядка обработки ограничений `CHECK`.)

`DEFAULT ` *`выражение_по_умолчанию`*

Предложение `DEFAULT` задаёт значение по умолчанию для столбца, в определении которого оно присутствует. Значение задаётся выражением без переменных (в частности, перекрёстные ссылки на другие столбцы текущей таблицы в нём не допускаются). Также не допускаются подзапросы. Тип данных выражения, задающего значение по умолчанию, должен соответствовать типу данных столбца.

Это выражение будет использоваться во всех операциях добавления данных, в которых не задаётся значение данного столбца. Если значение по умолчанию не определено, таким значением будет NULL.

`GENERATED ALWAYS AS ( ` *`генерирующее_выражение`* ` ) STORED`

Это предложение создаёт столбец как *генерируемый*. В такой столбец нельзя записать данные, а при чтении его возвращается результат указанного выражения.

Ключевое слово `STORED` отмечает, что этот столбец будет вычисляться при записи и сохраняться на диске.

Генерирующее выражение может обращаться к другим столбцам таблицы, но не к другим генерируемым столбцам. Все функции и операторы в нём должны быть постоянными. Обращаться к другим таблицам в таких выражениях нельзя.

`GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( ` *`параметры_последовательности`* ` ) ]`

С этим предложением столбец создаётся как *столбец идентификации*. С ним будет связана неявная последовательность, из которой этот столбец будет автоматически получать значения для новых строк. Такому столбцу неявно назначается свойство `NOT NULL`.

Предложения `ALWAYS` и `BY DEFAULT` определяют, как явно заданные пользователем значения будут обрабатываться командами `INSERT` и `UPDATE`.

В команде `INSERT`, в случае выбора `ALWAYS`, пользовательское значение используется, только если в этой команде указано `OVERRIDING SYSTEM VALUE`. С предложением `BY DEFAULT` пользовательскому значению отдаётся предпочтение. За подробностями обратитесь к описанию INSERT. (В команде `COPY` пользовательские значения используются всегда, вне зависимости от выбранного здесь варианта.)

В команде `UPDATE`, в случае выбора `ALWAYS`, попытка поместить в столбец любое значение, отличное от `DEFAULT`, будет отвергнута. Если выбран вариант `BY DEFAULT`, столбец может быть изменён обычным образом. (Предложение `OVERRIDING` для команды `UPDATE` отсутствует.)

Используя необязательное предложение *параметры_последовательности*, можно переопределить свойства последовательности. Среди доступных параметров: параметры для CREATE SEQUENCE, а также `SEQUENCE NAME` *имя*, `LOGGED` и `UNLOGGED`. Они позволяют выбрать имя и режим журналирования для последовательности. Без `SEQUENCE NAME` система проставит последовательности ещё не использованное имя. Без `LOGGED` или `UNLOGGED` у последовательности будет тот же уровень журналирования, что и у таблицы.

`UNIQUE [ NULLS [ NOT ] DISTINCT ]` (ограничение столбца)
`UNIQUE [ NULLS [ NOT ] DISTINCT ] ( имя_столбца [, ... ] )` [`INCLUDE ( имя_столбца [, ...])`] (ограничение таблицы)

Ограничение `UNIQUE` определяет, что группа из одного или нескольких столбцов таблицы может содержать только уникальные значения. Ограничение уникальности для таблицы ведёт себя точно так же, как ограничение для столбца, но может охватывать несколько столбцов. Таким образом, ограничение уникальности гарантирует, что любые две строки таблицы различаются как минимум в этих столбцах.

При проверке ограничения уникальности значения NULL не считаются равными, если не указан параметр `NULLS NOT DISTINCT`.

В каждом ограничении уникальности должен задаваться набор столбцов, отличный от набора любого другого ограничения уникальности или первичного ключа в данной таблице. (Избыточные ограничения уникальности будут просто игнорироваться.)

При установлении ограничения уникальности в многоуровневой иерархии секционирования в определение ограничения должны включаться все столбцы ключа секционирования целевой секционированной таблицы, а также столбцы всех подчинённых секционированных таблиц.

При добавлении ограничения уникальности автоматически будет создан уникальный индекс-B-дерево по столбцу или группе столбцов, перечисленных в ограничении. Созданный индекс наследует имя ограничения уникальности.

Необязательное предложение `INCLUDE` добавляет к этому индексу один или несколько столбцов, составляющих просто «дополнительную нагрузку»: для них уникальность не будет требоваться, и искать значения в них по данному индексу нельзя. Однако их содержимое может быть получено при сканировании только индекса. Заметьте, что хотя ограничение по неключевым столбцам не контролируется, оно всё же зависит от них. Как следствие, некоторые операции с этими столбцами (например, `DROP COLUMN`) могут повлечь каскадное удаление индекса и ограничения.

`PRIMARY KEY` (ограничение столбца)
`PRIMARY KEY ( имя_столбца [, ... ] )` [`INCLUDE ( имя_столбца [, ...])`] (ограничение таблицы)

Ограничение `PRIMARY KEY` определяет, что столбец или столбцы таблицы могут содержать только уникальные (без повторений) значения, отличные от NULL. Для таблицы может быть задан только один первичный ключ, будь то ограничение столбца или ограничение таблицы.

В определении первичного ключа должен задаваться набор столбцов, отличный от набора любого другого ограничения уникальности, установленного для данной таблицы. (В противном случае уникальное ограничение оказывается избыточным и будет отброшено.)

`PRIMARY KEY` устанавливает для данных те же ограничения, что и сочетание `UNIQUE` и `NOT NULL`, но созданный по набору столбцов первичный ключ также даёт метаинформацию о конструкции схемы, так как он подразумевает, что другие таблицы могут ссылаться на этот набор столбцов как на уникальный идентификатор строк.

Определяемые для секционированной таблицы ограничения `PRIMARY KEY` подчиняются тем же требованиям, что и ограничения `UNIQUE`.

При добавлении ограничения `PRIMARY KEY` автоматически будет создан уникальный индекс-B-дерево по столбцу или группе столбцов, перечисленных в ограничении. Созданный индекс наследует имя ограничения первичного ключа.

Необязательное предложение `INCLUDE` добавляет к этому индексу один или несколько столбцов, составляющих просто «дополнительную нагрузку»: для них уникальность не будет требоваться, и искать значения в них по данному индексу нельзя. Однако их содержимое может быть получено при сканировании только индекса. Заметьте, что хотя ограничение по неключевым столбцам не контролируется, оно всё же зависит от них. Как следствие, некоторые операции с этими столбцами (например, `DROP COLUMN`) могут повлечь каскадное удаление индекса и ограничения.

`EXCLUDE [ USING` *`индексный_метод`* `] (` *`элемент_исключения`* `WITH` *`оператор`* `[, ... ] )` *`параметры_индекса`* `[ WHERE (` *`предикат`* `) ]`

Предложение `EXCLUDE` определяет ограничение-исключение, которое гарантирует, что для любых двух строк, сравниваемых по указанным столбцам или выражениям с указанными операторами, результат не будет равен `TRUE` для всех сравнений. Если все указанные операторы проверяют равенство, это ограничение равносильно ограничению `UNIQUE`, хотя обычное ограничение уникальности будет работать быстрее. С другой стороны, в ограничениях-исключениях можно задавать более общие условия, чем простое условие на равенство. Например, можно задать ограничение, требующее, чтобы никакие две строки в таблице не содержали пересекающихся кругов (см. Раздел 8.8), применив оператор `&&`. Кроме того, операторы должны быть коммутативными.

Ограничения-исключения реализуются с помощью одноимённых индексов, так что каждый указанный в них оператор должен быть связан с соответствующим классом операторов (см. Раздел 11.10) для *`индексного_метода`*. В каждом *`элементе_исключения`* указывается столбец индекса, чтобы можно было дополнительно указать правило сортировки, класс оператора, параметры класса оператора и/или параметры сортировки, подробно описанные в CREATE INDEX.

Индексный метод доступа должен поддерживать `amgettuple` (см. Главу 60); в настоящее время это означает, что индексы GIN для этого не подходят. Хотя в ограничении-исключении можно использовать B-деревья и хеш-индексы, в этом мало смысла, так как такой подход ничем не лучше обычного ограничения уникальности. Так что на практике методом доступа всегда будет GiST или SP-GiST.

Параметр *`предикат`* позволяет указать ограничение-исключение для подмножества таблицы; внутри при этом создаётся частичный индекс. Заметьте, что предикат необходимо заключить в скобки.

При установлении ограничения исключения в многоуровневой иерархии секционирования в определение ограничения должны включаться все столбцы ключа секционирования целевой секционированной таблицы, а также столбцы всех подчинённых секционированных таблиц. Кроме того, эти столбцы должны сравниваться через оператор равенства. Данные ограничения обеспечивают существование в одной секции потенциально конфликтующих строк. Ограничение может также относиться к другим столбцам, не являющихся частью ключа секционирования, которые можно сравнить с помощью любого подходящего оператора.

`REFERENCES` *`внешняя_таблица`* `[ (` *`внешний_столбец`* `) ] [ MATCH` *`тип_совпадения`* `] [ ON DELETE` *`ссылочное_действие`* `] [ ON UPDATE` *`ссылочное_действие`* `]` (ограничение столбца)
`FOREIGN KEY (` *`имя_столбца`* `[, ... ] ) REFERENCES` *`внешняя_таблица`* `[ (` *`внешний_столбец`* `[, ... ] ) ] [ MATCH` *`тип_совпадения`* `] [ ON DELETE` *`ссылочное_действие`* `] [ ON UPDATE` *`ссылочное_действие`* `]` (ограничение таблицы)

Эти предложения определяют ограничение внешнего ключа, требующее, чтобы группа из одного или нескольких столбцов новой таблицы содержала только такие значения, которым соответствуют значения в заданных столбцах некоторой строки во внешней таблице. Если список *`целевых_столбцов`* опущен, в качестве него используется первичный ключ *`целевой_таблицы`*. Если же он задан, в качестве целевых столбцов должны указываться столбцы неоткладываемого уникального ограничения или первичного ключа во внешней таблице, либо столбцы нечастичного уникального индекса. При этом пользователь должен иметь право `REFERENCES` во внешней таблице (либо для всей таблицы, либо только для целевых столбцов). Для добавления ограничения внешнего ключа требуется блокировка `SHARE ROW EXCLUSIVE` в целевой таблице. Обратите внимание, что нельзя определить ограничения внешнего ключа, связывающие временные и постоянные таблицы.

Значения, вставляемые в ссылающиеся столбцы, сверяются со значениями во внешних столбцах внешней таблицы с учётом заданного типа совпадения. Возможны три типа совпадения: `MATCH FULL` (полное совпадение), `MATCH PARTIAL` (частичное совпадение) и тип по умолчанию, `MATCH SIMPLE` (простое совпадение). С `MATCH FULL` ни один из столбцов составного внешнего ключа не может содержать NULL, кроме случая, когда все внешние столбцы NULL; в этом случае строка может не иметь соответствия во внешней таблице. С `MATCH SIMPLE` любой из столбцов внешнего ключа может содержать NULL; при этом строка с NULL в одном из таких столбцов может не иметь соответствия во внешней таблице. Тип `MATCH PARTIAL` ещё не реализован. (Разумеется, чтобы вопросы со сравнением NULL не возникали, к столбцам, ссылающимся на внешние, можно применить ограничения `NOT NULL`.)

Кроме того, при изменении значений во внешних столбцах с данными в столбцах этой таблицы могут производиться определённые действия. Предложение `ON DELETE` задаёт действие, производимое при удалении некоторой строки во внешней таблице. Предложение `ON UPDATE` подобным образом задаёт действие, производимое при изменении значения в целевых столбцах внешней таблицы. Если строка изменена, но это изменение не затронуло целевые столбцы, никакое действие не производится. Ссылочные действия, кроме `NO ACTION`, нельзя сделать откладываемыми, даже если ограничение объявлено как откладываемое. Для каждого предложения возможные следующие варианты действий:

`NO ACTION`

Выдать ошибку, показывающую, что при удалении или изменении записи произойдёт нарушение ограничения внешнего ключа. Для отложенных ограничений ошибка произойдёт в момент проверки ограничения, если строки, ссылающиеся на эту запись, по-прежнему будут существовать. Этот вариант действия подразумевается по умолчанию.

`RESTRICT`

Выдать ошибку, показывающую, что при удалении или изменении записи произойдёт нарушение ограничения внешнего ключа. Этот вариант подобен `NO ACTION`, но эта проверка будет неоткладываемой.

`CASCADE`

Удалить все строки, ссылающиеся на удаляемую запись, либо поменять значения в ссылающихся столбцах на новые значения во внешних столбцах, в соответствии с операцией.

`SET NULL [ (` *`имя_столбца`* `[, ... ] ) ]`

Установить во всех ссылающихся столбцах или в указанном подмножестве ссылающихся столбцов значения null. Подмножество столбцов можно указать только для действий `ON DELETE`.

`SET DEFAULT [ ( *имя_столбца* [, ... ] ) ]`

Установить во всех ссылающихся столбцах или в указанном подмножестве ссылающихся столбцов значения по умолчанию. Подмножество столбцов можно указать только для действий `ON DELETE`. (Если значения по умолчанию отличны от NULL, во внешней таблице должна быть строка, соответствующая набору этих значений; в противном случае операция завершится ошибкой.)

Если внешние столбцы меняются часто, будет разумным добавить индекс для ссылающихся столбцов, чтобы действия по обеспечению ссылочной целостности, связанные с ограничением внешнего ключа, выполнялись более эффективно.

`DEFERRABLE`
`NOT DEFERRABLE`

Это предложение определяет, может ли ограничение быть отложенным. Неоткладываемое ограничение будет проверяться немедленно после каждой команды. Проверка откладываемых ограничений может быть отложена до завершения транзакции (обычно с помощью команды `SET CONSTRAINTS`). По умолчанию подразумевается вариант `NOT DEFERRABLE`. В настоящее время это предложение принимают только ограничения `UNIQUE`, `PRIMARY KEY`, `EXCLUDE` и `REFERENCES` (внешний ключ). Ограничения `NOT NULL` и `CHECK` не могут быть отложенными. Заметьте, что откладываемые ограничения не могут применяться в качестве решающих при конфликте в операторе `INSERT` с предложением `ON CONFLICT DO UPDATE`.

`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

Для откладываемых ограничений это предложение определяет, когда ограничение должно проверяться по умолчанию. Ограничение с характеристикой `INITIALLY IMMEDIATE` (подразумеваемой по умолчанию) проверяется после каждого оператора. Ограничение `INITIALLY DEFERRED`, напротив, проверяется только в конце транзакции. Время проверки ограничения можно изменить явно с помощью команды `SET CONSTRAINTS`.

`USING *метод*`

Это дополнительное предложение задаёт табличный метод доступа, который будет использоваться для сохранения содержимого новой таблицы; типом этого метода доступа должен быть `TABLE`. Подробнее об этом рассказывается в Главе 59. В случае отсутствия этого указания для новой таблицы выбирается метод доступа по умолчанию. За подробностями обратитесь к default_table_access_method.

При создании секции табличный метод доступа наследуется от секционированной таблицы, если он был задан.

`WITH ( *параметр_хранения* [= *значение*] [, ... ] )`

Это предложение определяет дополнительные параметры хранения для таблицы или индекса; за подробностями обратитесь к разделу Параметры хранения ниже. В целях обратной совместимости предложение `WITH` для таблицы также может содержать указание `OIDS=FALSE`, отмечающее, что строки новой таблицы не должны содержать OID (идентификатор объекта); указание `OIDS=TRUE` более не поддерживается

`WITHOUT OIDS`

Обеспечивающий обратную совместимость синтаксис создания таблицы с характеристикой `WITHOUT OIDS`; создание таблицы с указанием `WITH OIDS` более не поддерживается.

`ON COMMIT`

Поведением временных таблиц в конце блока транзакции позволяет управлять предложение `ON COMMIT`, которое принимает три параметра:

`PRESERVE ROWS`

Никакое специальное действие в конце транзакции не выполняется. Это поведение по умолчанию.

`DELETE ROWS`

Все строки в этой временной таблице будут удаляться в конце каждого блока транзакции. По сути, при каждой фиксации транзакции будет автоматически выполняться `TRUNCATE`. В случае секционированной таблицы это действие не распространяется на её секции.

`DROP`

Временная таблица будет удалена в конце текущего блока транзакции. Если это секционированная таблица, будут удалены и все её секции. Если у таблицы есть потомки в иерархии наследования, они также будут удалены.

`TABLESPACE` *`табл_пространство`*

Здесь *`табл_пространство`* — имя табличного пространства, в котором будет создаваться новая таблица. Если оно не указано, выбирается default_tablespace для обычных или temp_tablespaces для временных таблиц. Если же этот параметр задаётся для секционируемой таблицы, то ввиду того, что ей самой табличное пространство не требуется, заданное имя переопределяет значение `default_tablespace` (табличное пространство по умолчанию), которое применяется для всех создаваемых секций в случае отсутствия явного указания.

`USING INDEX TABLESPACE` *`табл_пространство`*

Это предложение позволяет выбрать табличное пространство, в котором будут создаваться индексы, связанные с ограничениями `UNIQUE`, `PRIMARY KEY` или `EXCLUDE`. Если оно не указано, выбирается default_tablespace или temp_tablespaces, если таблица временная.

Параметры хранения

Предложение `WITH` позволяет установить *параметры хранения* для таблиц и индексов, связанных с ограничениями `UNIQUE`, `PRIMARY KEY` и `EXCLUDE`. Параметры хранения для индексов документированы в CREATE INDEX. Поддерживаемые в настоящее время параметры хранения для таблиц перечислены ниже. Как показано, для многих параметров существует дополнительный параметр с тем же именем и префиксом `toast.`, который управляет поведением вторичной таблицы TOAST, если она есть (за дополнительными сведениями о TOAST обратитесь к Разделу 63.2). Если значение некоторого параметра задано для таблицы, а значение равнозначного параметра `toast.` не определено, для таблицы TOAST будет применяться значение параметра основной таблицы. Возможность задания этих параметров для секционированных таблиц не поддерживается, но вы можете задать их для отдельных конечных секций.

`fillfactor` (`integer`)

Фактор заполнения для таблицы, задаваемый в процентах, от 10 до 100. Значение по умолчанию — 100 (плотное заполнение). При меньшем факторе заполнения операции `INSERT` упаковывают данные в страницы только до заданного процента; оставшееся место резервируется для изменения строк на этой странице. В результате `UPDATE` получает шанс поместить изменённую копию строки в ту же страницу, что и исходную, что гораздо эффективнее, чем размещать её на другой странице, и увеличивает возможности для внесения изменений по схеме кортежи только в куче. Для таблиц, записи в которых никогда не меняются, лучшим выбором будет плотное заполнение, но для активно изменяемых таблиц лучше выбрать меньший фактор заполнения. Этот параметр нельзя задать для таблиц TOAST.

`toast_tuple_target` (`integer`)

Параметр toast_tuple_target задаёт минимальную длину кортежа, после превышения которой мы будем пытаться сжимать и/или переносить значения больших столбцов в таблицы TOAST и до которой мы будем пытаться сократить размер кортежа после перехода к TOAST. Это затрагивает столбцы с пометкой External (внешние, которые могут переноситься), Main (основные, которые могут сжиматься) или Extended (расширенные, которые могут и сжиматься, и просто переноситься) и касается только новых кортежей. На существующие кортежи это не влияет. По умолчанию этот параметр имеет значение, позволяющее разместить минимум 4 кортежа в блоке, что при стандартном размере блока составляет 2040 байт. Допустимые значения лежат в интервале от 128 байт до (размер_блока - заголовок), по умолчанию 8160 байт. Изменение этого значения может не отражаться на очень коротких и очень длинных кортежах. Заметьте, что выбранное по умолчанию значение часто близко к оптимальному, и весьма вероятно, что изменение этого параметра в некоторых случаях будет иметь отрицательный эффект. Для таблиц TOAST этот параметр задать нельзя.

`parallel_workers` (`integer`)

Данный параметр задаёт число рабочих процессов, которые должны задействоваться при параллельном сканировании таблицы. Если это значение не задано, система будет определять его, исходя из размера отношения. Фактическое число рабочих процессов, выбранное планировщиком или служебными операторами, выполняющими параллельное сканирование, может быть меньше, например, вследствие ограничения max_worker_processes.

`autovacuum_enabled`, `toast.autovacuum_enabled` (`boolean`)

Включает или отключает демон автоочистки для определённой таблицы. Со значением true демон автоочистки будет автоматически выполнять операции `VACUUM` и/или `ANALYZE` в этой таблице, согласно правилам, описанным в Подразделе 23.1.6. Со значением false эта таблица не будет подвергаться автоочистке, если только это не потребуется для предотвращения зацикливания идентификаторов транзакций. Более подробно предотвращение зацикливания описывается в Подразделе 23.1.5. Заметьте, что демон автоочистки не будет запускаться вовсе (если только это не потребуется для предотвращения зацикливания), если параметр autovacuum имеет значение false; это нельзя переопределить, установив параметры хранения для отдельных таблиц. Таким образом, явно устанавливать для этого параметра значение `true` практически не имеет смысла — полезно только значение `false`.

`vacuum_index_cleanup`, `toast.vacuum_index_cleanup` (`enum`)

Включает или отключает очистку индекса при выполнении в этой таблице операции `VACUUM`. Значение по умолчанию — `AUTO`. Значение `OFF` отключаёт очистку индекса, `ON` — включает, а со значением `AUTO` решение об очистке принимается динамически при каждом выполнении `VACUUM`. Динамический выбор позволяет избежать во время `VACUUM` ненужного сканирования индексов, в результате которого удаляются лишь несколько мёртвых кортежей. Принудительное безусловное отключение очистки индексов может очень существенно ускорить `VACUUM`, но также может привести к значительному раздуванию индексов при частых изменениях данных в таблице. Если для команды `VACUUM` задаётся параметр `INDEX_CLEANUP`, он переопределяет значение `vacuum_index_cleanup`.

`vacuum_truncate`, `toast.vacuum_truncate` (`boolean`)

Включает или отключает процедуру отсечения пустых страниц в конце таблицы в процессе очистки. Значение по умолчанию — `true` (вкл.). Когда этот параметр включён, операции `VACUUM` и автоочистка пытаются отсечь пустые страницы, чтобы освободившееся место возвратилось операционной системе. Заметьте, что для этого отсечения требуется блокировка таблицы на уровне `ACCESS EXCLUSIVE`. Явно заданный параметр `TRUNCATE` команды `VACUUM` переопределяет значение данного параметра.

`autovacuum_vacuum_threshold`, `toast.autovacuum_vacuum_threshold` (`integer`)

Значение параметра autovacuum_vacuum_threshold для таблицы.

`autovacuum_vacuum_scale_factor`, `toast.autovacuum_vacuum_scale_factor` (`floating point`)

Значение параметра autovacuum_vacuum_scale_factor для таблицы.

`autovacuum_vacuum_insert_threshold`, `toast.autovacuum_vacuum_insert_threshold` (`integer`)

Значение параметра autovacuum_vacuum_insert_threshold для таблицы. Особое значение -1 отключает очистку, вызываемую добавлением данных.

`autovacuum_vacuum_insert_scale_factor`, `toast.autovacuum_vacuum_insert_scale_factor` (`floating point`)

Значение параметра autovacuum_vacuum_insert_scale_factor для таблицы.

`autovacuum_analyze_threshold` (`integer`)

Значение параметра autovacuum_analyze_threshold для таблицы.

`autovacuum_analyze_scale_factor` (`floating point`)

Значение параметра autovacuum_analyze_scale_factor для таблицы.

`autovacuum_vacuum_cost_delay`, `toast.autovacuum_vacuum_cost_delay` (`floating point`)

Значение параметра autovacuum_vacuum_cost_delay для таблицы.

`autovacuum_vacuum_cost_limit`, `toast.autovacuum_vacuum_cost_limit` (`integer`)

Значение параметра autovacuum_vacuum_cost_limit для таблицы.

`autovacuum_freeze_min_age`, `toast.autovacuum_freeze_min_age` (`integer`)

Значение параметра vacuum_freeze_min_age для таблицы. Учтите, что система будет игнорировать установленные для таблиц значения `autovacuum_freeze_min_age`, превышающие половину системного autovacuum_freeze_max_age.

`autovacuum_freeze_max_age`, `toast.autovacuum_freeze_max_age` (`integer`)

Значение параметра autovacuum_freeze_max_age для таблицы. Учтите, что система будет игнорировать установленные для таблиц значения `autovacuum_freeze_max_age`, превышающие значение системного параметра (они могут быть только меньше).

`autovacuum_freeze_table_age`, `toast.autovacuum_freeze_table_age` (`integer`)

Значение параметра vacuum_freeze_table_age для таблицы.

`autovacuum_multixact_freeze_min_age`, `toast.autovacuum_multixact_freeze_min_age` (`integer`)

Значение параметра vacuum_multixact_freeze_min_age для таблицы. Учтите, что демон автоочистки будет игнорировать установленные для таблиц значения `autovacuum_multixact_freeze_min_age`, превышающие половину значения системного параметра autovacuum_multixact_freeze_max_age.

`autovacuum_multixact_freeze_max_age`, `toast.autovacuum_multixact_freeze_max_age` (`integer`)

Значение параметра autovacuum_multixact_freeze_max_age для таблицы. Учтите, что система автоочистки будет игнорировать установленные для таблиц параметры `autovacuum_multixact_freeze_max_age`, превышающие системный параметр (они могут быть только меньше).

`autovacuum_multixact_freeze_table_age`, `toast.autovacuum_multixact_freeze_table_age` (`integer`)

Значения параметра vacuum_multixact_freeze_table_age для таблицы.

`log_autovacuum_min_duration`, `toast.log_autovacuum_min_duration` (`integer`)

Значения параметра log_autovacuum_min_duration для таблицы.

`user_catalog_table` (`boolean`)

Объявляет таблицу как дополнительную таблицу каталога, например для целей логической репликации. За подробностями обратитесь к Подразделу 47.6.2. Для таблиц TOAST этот параметр задать нельзя.

Примечания

Postgres Pro автоматически создаёт индекс, гарантирующий уникальность, для каждого ограничения уникальности и ограничения первичного ключа. Поэтому явно создавать индекс для столбцов первичного ключа не требуется. (За дополнительными сведениями обратитесь к CREATE INDEX.)

Ограничения уникальности и первичные ключи в текущей реализации не наследуются. Вследствие этого ограничения уникальности довольно плохо сочетаются с наследованием.

В таблице не может быть больше 1600 столбцов. (На практике фактический предел обычно ниже из-за ограничения на длину записи.)

Примеры

Создание таблицы `films` и таблицы `distributors`:

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

Создание таблицы с двумерным массивом:

```
CREATE TABLE array_int (
    vector  int[][]
);
```

Определение ограничения уникальности для таблицы `films`. Ограничения уникальности могут быть определены для одного или нескольких столбцов таблицы:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Определение ограничения-проверки для столбца:

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

Определение ограничения-проверки для таблицы:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
```

```
);
```

Определение ограничения первичного ключа для таблицы `films`:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Определение ограничения первичного ключа для таблицы `distributors`. Следующие два примера равнозначны, но в первом используется синтаксис ограничений для таблицы, а во втором — для столбца:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

Определение значений по умолчанию: для столбца `name` значением по умолчанию будет строка, для столбца `did` — следующее значение объекта последовательности, а для `modtime` — время, когда была вставлена запись:

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

Определение двух ограничений `NOT NULL` для столбцов таблицы `distributors`, при этом одному ограничению даётся явное имя:

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

Определение ограничения уникальности для столбца `name`:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

То же самое условие, но в виде ограничения таблицы:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

Создание такой же таблицы с фактором заполнения 70% для таблицы и её уникального индекса:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

Создание таблицы `circles` с ограничением-исключением, не допускающим пересечения двух кругов:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Создание таблицы `cinemas` в табличном пространстве `diskvol1`:

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

Создание составного типа и типизированной таблицы:

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

Создание таблицы, секционируемой по диапазонам:

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

Создание таблицы, секционируемой по диапазонам, с ключом секционирования, включающим несколько столбцов:

```
CREATE TABLE measurement_year_month (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH FROM logdate));
```

Создание таблицы, секционируемой по спискам:

```
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));
```

Создание таблицы, секционируемой по хешу:

```
CREATE TABLE orders (
    order_id     bigint not null,
    cust_id      bigint not null,
```

```
    status       text
) PARTITION BY HASH (order_id);
```

Создание секции таблицы, секционируемой по диапазонам:

```
CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
    unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

Создание нескольких секций для таблицы, секционируемой по диапазонам, с ключом секционирования, включающим несколько столбцов:

```
CREATE TABLE measurement_ym_older
    PARTITION OF measurement_year_month
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

CREATE TABLE measurement_ym_y2016m11
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 11) TO (2016, 12);

CREATE TABLE measurement_ym_y2016m12
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 12) TO (2017, 01);

CREATE TABLE measurement_ym_y2017m01
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2017, 01) TO (2017, 02);
```

Создание секции таблицы, секционируемой по спискам:

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

Создание секции таблицы, секционируемой по спискам (при этом сама секция также создаётся секционируемой), и добавление секции в неё:

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);

CREATE TABLE cities_ab_10000_to_100000
    PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

Создание секций таблицы, секционируемой по хешу:

```
CREATE TABLE orders_p1 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Создание секции по умолчанию:

```
CREATE TABLE cities_partdef
    PARTITION OF cities DEFAULT;
```

Совместимость

Команда `CREATE TABLE` соответствует стандарту SQL, с описанными ниже исключениями.

Временные таблицы

Хотя синтаксис `CREATE TEMPORARY TABLE` подобен аналогичному в стандарте SQL, результат получается другим. В стандарте временные таблицы определяются только один раз и существуют (изначально пустые) в каждом сеансе, в котором они используются. Postgres Pro вместо этого требует, чтобы каждый сеанс выполнял собственную команду `CREATE TEMPORARY TABLE` для каждой временной таблицы, которая будет использоваться. Это позволяет использовать в разных сеансах таблицы с одинаковыми именами для разных целей, тогда как при подходе, регламентированном стандартом, все экземпляры временной таблицы с одним именем должны иметь одинаковую табличную структуру.

Поведение временных таблиц, описанное в стандарте, в большинстве своём игнорируют и другие СУБД, так что в этом отношении Postgres Pro ведёт себя так же, как и ряд других СУБД.

В стандарте SQL также разделяются глобальные и локальные временные таблицы — в локальной временной таблице содержится отдельный набор данных для каждого модуля SQL в отдельном сеансе, хотя её определение так же разделяется между ними. Так как в Postgres Pro модули SQL не поддерживаются, это различие в Postgres Pro не существует.

Совместимости ради, Postgres Pro принимает ключевые слова `GLOBAL` и `LOCAL` в объявлении временной таблицы, но в настоящее время они никак не действуют. Использовать их не рекомендуется, так как в будущих версиях Postgres Pro может быть принята их интерпретация, более близкая к стандарту.

Предложение `ON COMMIT` для временных таблиц тоже подобно описанному в стандарте SQL, но есть некоторые отличия. Если предложение `ON COMMIT` опущено, в SQL подразумевается поведение `ON COMMIT DELETE ROWS`. Однако в Postgres Pro по умолчанию действует `ON COMMIT PRESERVE ROWS`. Параметр `ON COMMIT DROP` в стандарте SQL отсутствует.

Неотложенные ограничения уникальности

Когда ограничение `UNIQUE` или `PRIMARY KEY` не является отложенным, Postgres Pro проверяет уникальность непосредственно в момент добавления или изменения строки. Стандарт SQL говорит, что уникальность должна обеспечиваться только в конце оператора; это различие проявляется, например когда одна команда изменяет множество ключевых значений. Чтобы получить поведение, оговоренное стандартом, объявите ограничение как откладываемое (`DEFERRABLE`), но не отложенное (т. е., `INITIALLY IMMEDIATE`). Учтите, что этот вариант может быть значительно медленнее, чем немедленная проверка ограничений.

Ограничения-проверки для столбцов

Стандарт SQL говорит, что ограничение `CHECK`, определяемое для столбца, может ссылаться только на столбец, с которым оно связано; только ограничения `CHECK` для таблиц могут ссылаться на несколько столбцов. В Postgres Pro этого ограничения нет; он воспринимает ограничения-проверки для столбцов и таблиц одинаково.

Ограничение `EXCLUDE`

Ограничения `EXCLUDE` являются расширением Postgres Pro.

Ограничения внешнего ключа

Возможность указывать списки столбцов в действиях внешнего ключа `SET DEFAULT` и `SET NULL` является расширением Postgres Pro.

Возможность ограничения внешнего ключа ссылаться на столбцы уникального индекса вместо столбцов первичного ключа или ограничения уникальности является расширением Postgres Pro.

`NULL` «Ограничение»

«Ограничение» `NULL` (на самом деле это не ограничение) является расширением Postgres Pro стандарта SQL, которое реализовано для совместимости с некоторыми другими СУБД (и для симметрии с ограничением `NOT NULL`). Так как это поведение по умолчанию для любого столбца, его присутствие не несёт смысловой нагрузки.

Имена ограничений

В стандарте SQL говорится, что имена ограничений таблицы и ограничений домена должны быть уникальными в схеме, содержащей эту таблицу или домен. Однако Postgres Pro менее строг: он требует только, чтобы имена были уникальны среди ограничений, присоединённых к данной конкретной таблице или домену. Но такого послабления нет для ограничений, построенных на индексах (ограничений `UNIQUE`, `PRIMARY KEY` и `EXCLUDE`), так как ограничение и связанный с ним индекс имеют одно имя, а имена индексов должны быть уникальны среди всех отношений в их схеме.

В настоящее время в Postgres Pro ограничения `NOT NULL` вообще не имеют имён, так что на них требования уникальности не распространяются. Однако это может поменяться в будущих выпусках.

Наследование

Множественное наследование посредством `INHERITS` является языковым расширением Postgres Pro. SQL:1999 и более поздние стандарты определяют единичное наследование с другим синтаксисом и смыслом. Наследование в стиле SQL:1999 пока ещё не поддерживается в Postgres Pro.

Таблицы с нулём столбцов

Postgres Pro позволяет создать таблицу без столбцов (например, `CREATE TABLE foo();`). Это расширение стандарта SQL, который не допускает таблицы с нулём столбцов. Таблицы с нулём столбцов сами по себе не очень полезны, но если их запретить, возникают странные особые ситуации с командой `ALTER TABLE DROP COLUMN`, так что лучшим вариантом кажется игнорировать это требование стандарта.

Множество столбцов идентификации

Postgres Pro позволяет иметь в таблице более одного столбца идентификации. В стандарте же говорится, что в таблице может быть максимум один столбец идентификации. Это ограничение ослаблено в основном для большей гибкости при выполнении изменений в схеме или миграции. Заметьте, что команда `INSERT` поддерживает только одно предложение переопределения значения, применяемое ко всему оператору, так что с несколькими столбцами идентификации различное поведение не поддерживается должным образом.

Генерируемые столбцы

Указание `STORED` отсутствует в стандарте, но используется и в других реализациях баз данных SQL. Стандарт SQL не предусматривает хранение генерируемых столбцов.

Предложение `LIKE`

Хотя предложение `LIKE` описано в стандарте SQL, многие варианты его использования, допустимые в Postgres Pro, в стандарте не описаны, а некоторые предусмотренные в стандарте возможности не реализованы в Postgres Pro.

Предложение `WITH`

Предложение `WITH` является расширением Postgres Pro; в стандарте параметры хранения не оговариваются.

Табличные пространства

Концепция табличных пространств в Postgres Pro отсутствует в стандарте. Как следствие, предложения `TABLESPACE` и `USING INDEX TABLESPACE` являются расширениями.

Типизированные таблицы

Типизированные таблицы реализуют подмножество стандарта SQL. Согласно стандарту, типизированная таблица содержит столбцы, соответствующие нижележащему составному типу, и ещё один столбец, ссылающийся на себя. Postgres Pro не поддерживает ссылающиеся на себя столбцы явно.

Предложение `PARTITION BY`

Предложение `PARTITION BY` является расширением Postgres Pro.

Предложение `PARTITION OF`

Предложение `PARTITION OF` является расширением Postgres Pro.

См. также

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE, CREATE TYPE

CREATE TABLE AS

CREATE TABLE AS — создать таблицу из результатов запроса

Синтаксис

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] имя_таблицы
    [ (имя_столбца [, ...] ) ]
    [ USING метод ]
    [ WITH ( параметр_хранения [= значение] [, ... ] ) | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE табл_пространство ]
    AS query
    [ WITH [ NO ] DATA ]
```

Описание

`CREATE TABLE AS` создаёт таблицу и наполняет её данными, полученными в результате выполнения `SELECT`. Столбцы этой таблицы получают имена и типы данных в соответствии со столбцами результата `SELECT` (хотя имена столбцов можно переопределить, добавив явно список новых имён столбцов).

`CREATE TABLE AS` напоминает создание представления, но на самом деле есть значительная разница: эта команда создаёт новую таблицу и выполняет запрос только раз, чтобы наполнить таблицу начальными данными. Последующие изменения в исходных таблицах запроса в новой таблице отражаться не будут. С представлением, напротив, определяющая его команда `SELECT` выполняется при каждой выборке из него.

Команда `CREATE TABLE AS` требует наличия права `CREATE` для схемы, выбранной для создания материализованного представления.

Параметры

`GLOBAL` или `LOCAL`

Для совместимости игнорируются. Использование этих ключевых слов считается устаревшим; за подробностями обратитесь к CREATE TABLE.

`TEMPORARY` или `TEMP`

Если указано, создаваемая таблица будет временной. За подробностями обратитесь к CREATE TABLE.

`UNLOGGED`

Если указано, создаваемая таблица будет нежурналируемой. За подробностями обратитесь к CREATE TABLE.

`IF NOT EXISTS`

Не считать ошибкой, если отношение с таким именем уже существует; просто выдать замечание и оставить таблицу без изменений.

`имя_таблицы`

Имя создаваемой таблицы (возможно, дополненное схемой).

`имя_столбца`

Имя столбца в создаваемой таблице. Если имена столбцов не заданы явно, они определяются по именам столбцов результата запроса.

`USING` *`метод`*

Это дополнительное предложение задаёт табличный метод доступа, который будет использоваться для сохранения содержимого новой таблицы; типом этого метода доступа должен быть `TABLE`. Подробнее об этом рассказывается в Главе 59. В случае отсутствия этого указания для новой таблицы выбирается метод доступа по умолчанию. За подробностями обратитесь к default_table_access_method.

`WITH (` *`параметр_хранения`* `[=` *`значение`*`] [, ... ] )`

Это предложение определяет дополнительные параметры хранения для новой таблицы; см. раздел Параметры хранения в описании CREATE TABLE. В целях обратной совместимости предложение `WITH` для таблицы также может содержать указание `OIDS=FALSE`, отмечающее, что строки новой таблицы не должны содержать OID (идентификатор объекта); указание `OIDS=TRUE` более не поддерживается.

`WITHOUT OIDS`

Обеспечивающий обратную совместимость синтаксис создания таблицы с характеристикой `WITHOUT OIDS`; создание таблицы с указанием `WITH OIDS` более не поддерживается.

`ON COMMIT`

Поведением временных таблиц в конце блока транзакции позволяет управлять предложение `ON COMMIT`, которое принимает три параметра:

`PRESERVE ROWS`

Никакое специальное действие в конце транзакции не выполняется. Это поведение по умолчанию.

`DELETE ROWS`

Все строки в этой временной таблице будут удаляться в конце каждого блока транзакции. По сути, при каждой фиксации транзакции будет автоматически выполняться `TRUNCATE`.

`DROP`

Эта временная таблица будет удаляться в конце текущего блока транзакции.

`TABLESPACE` *`табл_пространство`*

Здесь *`табл_пространство`* — имя табличного пространства, в котором будет создаваться новая таблица. Если оно не указано, выбирается default_tablespace или temp_tablespaces, если таблица временная.

`query`

Команда `SELECT`, `TABLE` или `VALUES`, либо команда `EXECUTE`, выполняющая подготовленный запрос `SELECT`, `TABLE` или `VALUES`.

`WITH [ NO ] DATA`

Это предложение определяет, будут ли данные, выданные запросом, копироваться в новую таблицу. Если нет, то копируется только структура. По умолчанию данные копируются.

Примечания

Функциональность этой команды подобна SELECT INTO, но предпочтительнее использовать её, во избежание путаницы с другими применениями синтаксиса `SELECT INTO`. Кроме того, набор возможностей `CREATE TABLE AS` шире, чем у `SELECT INTO`.

Примеры

Создание таблицы `films_recent`, содержащей только последние записи из таблицы `films`:

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

Чтобы скопировать таблицу полностью, можно также использовать короткую форму команды `TABLE`:

```
CREATE TABLE films2 AS
  TABLE films;
```

Создание временной таблицы `films_recent`, содержащей только последние записи таблицы `films`, с применением подготовленного оператора. Новая таблица прекратит существование при фиксации транзакции:

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

Совместимость

`CREATE TABLE AS` соответствует стандарту SQL. Нестандартные расширения перечислены ниже:

- Стандарт требует заключать предложение подзапроса в скобки, но в Postgres Pro эти скобки необязательны.
- Стандарт требует наличия указания `WITH [ NO ] DATA`, в Postgres Pro оно необязательно.
- Postgres Pro работает с временными таблицами не так, как описано в стандарте; за подробностями обратитесь к CREATE TABLE.
- Предложение `WITH` является расширением Postgres Pro; в стандарте параметры хранения не оговариваются.
- Концепция табличных пространств в Postgres Pro отсутствует в стандарте. Как следствие, предложение `TABLESPACE` является расширением.

См. также

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

CREATE TABLESPACE

CREATE TABLESPACE — создать табличное пространство

Синтаксис

```
CREATE TABLESPACE табл_пространство
    [ OWNER { новый_владелец | CURRENT_ROLE | CURRENT_USER | SESSION_USER } ]
    LOCATION 'directory'
    [ WITH ( параметр_табличного_пространства = значение [, ... ] ) ]
```

Описание

`CREATE TABLESPACE` регистрирует новое табличное пространство на уровне кластера баз данных. Имя табличного пространства должно отличаться от имён уже существующих табличных пространств в кластере.

Табличные пространства позволяют администраторам баз данных определять альтернативные расположения в файловой системе, где могут находиться файлы, содержащие объекты базы данных (например, таблицы или индексы). Создавать табличные пространства могут только суперпользователи и пользователи с правами роли `pg_create_tablespace`, но их владельцами могут быть назначены и обычные пользователи.

Пользователь, имеющий соответствующие права, может передать параметр *`табл_пространство`* команде `CREATE DATABASE`, `CREATE TABLE`, `CREATE INDEX` или `ADD CONSTRAINT`, чтобы файлы данных для этих объектов хранились в указанном табличном пространстве.

Предупреждение

Табличное пространство нельзя использовать отдельно от кластера, в котором оно было определено; см. Раздел 21.6.

Параметры

`табл_пространство`

Имя создаваемого табличного пространства. Это имя не может начинаться с `pg_`, так как такие имена зарезервированы для системных табличных пространств.

`имя_пользователя`

Имя пользователя, который будет владельцем табличного пространства. Если опущено, владельцем по умолчанию станет пользователь, выполняющий команду.

`directory`

Каталог, который будет использован для этого табличного пространства. Этот каталог должен уже существовать (`CREATE TABLESPACE` не создаст его), быть пустым и принадлежать системному пользователю Postgres Pro. Задаваться его расположение должно абсолютным путём.

`параметр_табличного_пространства`

Устанавливаемый или сбрасываемый параметр табличного пространства. В настоящее время поддерживаются только параметры `seq_page_cost`, `random_page_cost`, `effective_io_concurrency` и `maintenance_io_concurrency`. При установке этих значений для заданного табличного пространства переопределяются обычная оценка стоимости чтения страниц из таблиц в этом пространстве и характеристики предвыборки во время выполнения, зависящие от одноимённых параметров конфигурации (см. seq_page_cost, random_page_cost, effective_io_concurrency, maintenance_io_concurrency). Это может быть полезно, если одно из табличных пространств размещено на диске, который быстрее или медленнее остальной дисковой подсистемы.

Примечания

`CREATE TABLESPACE` не может быть выполнена внутри блока транзакции.

Примеры

Чтобы создать табличное пространство `dbspace`, расположенное в файловой системе в каталоге `/data/dbs`, сначала создайте этот каталог средствами операционной системы и установите для него подходящего владельца:

```
mkdir /data/dbs
chown postgres:postgres /data/dbs
```

Затем выполните в Postgres Pro команду, собственно создающую табличное пространство:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

Чтобы создать табличное пространство, которое будет принадлежать другому пользователю БД, выполните такую команду:

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

Совместимость

`CREATE TABLESPACE` является расширением Postgres Pro.

См. также

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

CREATE TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION — создать конфигурацию текстового поиска

Синтаксис

```
CREATE TEXT SEARCH CONFIGURATION имя (
    PARSER = имя_анализатора |
    COPY = исходная_конфигурация
)
```

Описание

`CREATE TEXT SEARCH CONFIGURATION` создаёт конфигурацию текстового поиска. Конфигурация текстового поиска определяет анализатор текстового поиска, разделяющий строку на фрагменты, и словари, позволяющие установить, какие именно фрагменты представляют интерес при поиске.

Если задаётся только анализатор, новая конфигурация текстового поиска не будет содержать сопоставления типов фрагментов со словарями и, как следствие, будет игнорировать все слова. Для создания сопоставлений, которые бы сделали конфигурацию полезной, затем нужно будет воспользоваться командой `ALTER TEXT SEARCH CONFIGURATION`. Другой вариант команды позволяет скопировать конфигурацию текстового поиска.

Если указывается имя схемы, конфигурация текстового поиска создаётся в указанной схеме. В противном случае она создаётся в текущей схеме.

Владельцем конфигурации текстового поиска становится пользователь её создавший.

За дополнительными сведениями обратитесь к Главе 12.

Параметры

имя

Имя создаваемой конфигурации текстового поиска, возможно, дополненное схемой.

имя_анализатора

Имя анализатора текстового поиска, который будет использоваться этой конфигурацией.

исходная_конфигурация

Имя существующей конфигурации текстового поиска, которая будет скопирована.

Примечания

Параметры `PARSER` и `COPY` являются взаимоисключающими, так как при копировании существующей конфигурации выбранный в ней анализатор копируется тоже.

Совместимость

Оператор `CREATE TEXT SEARCH CONFIGURATION` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH DICTIONARY — создать словарь текстового поиска

Синтаксис

```
CREATE TEXT SEARCH DICTIONARY имя (
    TEMPLATE = шаблон
    [, параметр = значение [, ... ]]
)
```

Описание

`CREATE TEXT SEARCH DICTIONARY` создаёт словарь текстового поиска. Словарь текстового поиска определяет способ выделения слов, представляющих или не представляющих интерес для поиска. Работа словаря зависит от шаблона текстового поиска, в котором определяются функции, собственно выполняющие действия. Обычно словарь задаёт некоторые параметры, управляющие частным поведением функций шаблона.

Если указывается имя схемы, словарь текстового поиска создаётся в указанной схеме. В противном случае он создаётся в текущей схеме.

Владельцем словаря текстового поиска становится пользователь его создавший.

За дополнительными сведениями обратитесь к Главе 12.

Параметры

имя

Имя создаваемого словаря текстового поиска, возможно, дополненное схемой.

шаблон

Имя шаблона текстового поиска, который будет определять общее поведение этого словаря.

параметр

Имя параметра шаблона, устанавливаемого для данного словаря.

значение

Значение для параметра, связанного с шаблоном. Если это не простой идентификатор или число, его следует заключить в кавычки (при желании его можно заключать в кавычки всегда).

Параметры могут перечисляться в любом порядке.

Примеры

Команда в следующем примере создаёт словарь на базе Snowball с нестандартным списком стоп-слов.

```
CREATE TEXT SEARCH DICTIONARY my_russian (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

Совместимость

Оператор `CREATE TEXT SEARCH DICTIONARY` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSER — создать анализатор текстового поиска

Синтаксис

```
CREATE TEXT SEARCH PARSER имя (
    START = функция_начала ,
    GETTOKEN = функция_выдачи_фрагмента ,
    END = функция_окончания ,
    LEXTYPES = функция_лекс_типов
    [, HEADLINE = функция_выдержек ]
)
```

Описание

`CREATE TEXT SEARCH PARSER` создаёт анализатор текстового поиска. Анализатор текстового поиска определяет способ разделения текстовой строки на фрагменты и назначения типов (категорий) этим фрагментам. Анализатор не очень полезен сам по себе, для осуществления поиска он должен быть подключён к конфигурации текстового поиска вместе с определёнными словарями.

Если указывается имя схемы, словарь текстового поиска создаётся в указанной схеме. В противном случае он создаётся в текущей схеме.

Выполнить `CREATE TEXT SEARCH PARSER` может только суперпользователь. (Это ограничение введено потому, что ошибочное определение анализатора текстового поиска может вызвать нарушения или даже сбой в работе сервера.)

За дополнительными сведениями обратитесь к Главе 12.

Параметры

имя

Имя создаваемого анализатора текстового поиска, возможно, дополненное схемой.

функция_начала

Имя функции, вызываемой в начале обработки.

функция_выдачи_фрагмента

Имя функции, выдающей следующий фрагмент.

функция_окончания

Имя функции, вызываемой по окончании обработки.

функция_лекс_типов

Имя функции перечисления лексических типов (эта функция выдаёт информацию о множестве типов фрагментов, выделяемых анализатором).

функция_выдержек

Имя функции извлечения выдержек (эта функция выделяет краткое содержание для набора фрагментов).

Имена функций могут быть дополнены именем схемы, если требуется. Типы аргументов не указываются, так как список аргументов для всех типов функций предопределён. Обязательными являются все функции, кроме функции выдержек.

Аргументы могут перечисляться в любом порядке, не только в том, что показан выше.

Совместимость

Оператор `CREATE TEXT SEARCH PARSER` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE — создать шаблон текстового поиска

Синтаксис

```
CREATE TEXT SEARCH TEMPLATE имя (
    [ INIT = функция_инициализации , ]
    LEXIZE = функция_выделения_лексем
)
```

Описание

`CREATE TEXT SEARCH TEMPLATE` создаёт новый шаблон текстового поиска. Шаблоны текстового поиска определяют функции для реализации словарей текстового поиска. Шаблон сам по себе бесполезен, но его экземпляр нужно создать для словаря. Словарь обычно определяет параметры, передаваемые функциям шаблона.

Если указывается имя схемы, шаблон текстового поиска создаётся в указанной схеме. В противном случае он создаётся в текущей схеме.

Выполнить `CREATE TEXT SEARCH TEMPLATE` может только суперпользователь. Это ограничение введено потому, что ошибочное определение шаблона текстового поиска может вызвать нарушения или даже сбой в работе сервера. Смысл отделения шаблонов от словарей в том, что шаблон покрывает «небезопасные» аспекты определения словаря. Параметры, которые можно задать при определении словаря, не могут причинить вред, поэтому создавать словари разрешено и непривилегированным пользователям.

За дополнительными сведениями обратитесь к Главе 12.

Параметры

`имя`

Имя создаваемого шаблона текстового поиска, возможно, дополненное схемой.

`функция_инициализации`

Имя функции инициализации для шаблона.

`функция_выделения_лексем`

Имя функции выделения лексем.

Имена функций могут быть дополнены именем схемы, если требуется. Типы аргументов не указываются, так как список аргументов для всех типов функций предопределён. Функция выделения лексем является обязательной, а функция инициализации может отсутствовать.

Аргументы могут перечисляться в любом порядке, не только в том, что показан выше.

Совместимость

Оператор `CREATE TEXT SEARCH TEMPLATE` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

CREATE TRANSFORM

CREATE TRANSFORM — создать трансформацию

Синтаксис

```
CREATE [ OR REPLACE ] TRANSFORM FOR имя_типа LANGUAGE имя_языка (
    FROM SQL WITH FUNCTION имя_функции_из_sql [ (тип_аргумента [, ...]) ],
    TO SQL WITH FUNCTION имя_функции_в_sql [ (тип_аргумента [, ...]) ]
);
```

Описание

`CREATE TRANSFORM` определяет новую трансформацию. `CREATE OR REPLACE TRANSFORM` либо создаёт трансформацию, либо заменяет существующую.

Трансформация определяет, как преобразовать тип данных для процедурного языка. Например, если написать на языке PL/Python функцию, использующую тип `hstore`, PL/Python заведомо не знает, как должны представляться значения `hstore` в среде Python. Обычно реализации языка нисходят к текстовому представлению, но это может быть неудобно, когда более уместен был бы, например, ассоциативный массив или список.

Трансформация определяет две функции:

- Функция «из SQL» преобразует тип из среды SQL в среду языка. Эта функция будет вызываться для аргументов функции, написанной на этом языке.
- Функция «в SQL» преобразует тип из среды языка в среду SQL. Эта функция будет вызываться для значения, возвращаемого из функции на этом языке.

Предоставлять обе эти функции не требуется, можно ограничиться одной. Если одна из них не указана, при необходимости выбирается поведение, принятое для языка по умолчанию. (Чтобы полностью перекрыть путь трансформации в одну сторону, можно написать функцию, которая будет всегда выдавать ошибку.)

Чтобы создать трансформацию, необходимо быть владельцем и иметь право `USAGE` для типа, иметь право `USAGE` для языка, а также быть владельцем и иметь право `EXECUTE` для функций из-SQL и в-SQL, если они задаются.

Параметры

`имя_типа`

Имя типа данных, для которого предназначена трансформация.

`имя_языка`

Имя языка, для которого предназначена трансформация.

`имя_функции_из_sql``[(`*`тип_аргумента`*` [, ...])]`

Имя функции для преобразования типа из среды SQL в среду языка. Она должна принимать один аргумент типа `internal` и возвращать тип `internal`. Фактический аргумент будет иметь тип, заданный для трансформации, и сама функция должна рассчитывать на это. (Но на уровне SQL нельзя объявить функцию, возвращающую тип `internal`, если она не принимает минимум один аргумент типа `internal`.) Фактически возвращаемое значение будет определяться реализацией языка. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

`имя_функции_в_sql``[(`*`тип_аргумента`*` [, ...])]`

Имя функции для преобразования типа из среды языка в среду SQL. Она должна принимать один аргумент типа `internal` и возвращать тип, для которого создаётся трансформация. Фактическое значение аргумента будет определяться реализацией языка. Если список аргументов отсутствует, имя функции должно быть уникальным в её схеме.

Примечания

Для удаления трансформаций применяется `DROP TRANSFORM`.

Примеры

Чтобы создать трансформацию для типа `hstore` и языка `plpython3u`, сначала нужно создать тип и язык:

```
CREATE TYPE hstore ...;

CREATE EXTENSION plpython3u;
```

Затем создайте необходимые функции:

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

И наконец, создайте трансформацию, соединяющую всё это вместе:

```
CREATE TRANSFORM FOR hstore LANGUAGE plpython3u (
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),
    TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

На практике эти команды помещаются в расширение.

В разделе `contrib` представлено несколько расширений, в которых определены трансформации, что может послужить практическим примером реализации.

Совместимость

Первая форма `CREATE TRANSFORM` является расширением Postgres Pro. В стандарте SQL есть команда `CREATE TRANSFORM`, но её предназначение — преобразовывать типы для языков на стороне клиента. Этот вариант использования не поддерживается Postgres Pro.

См. также

CREATE FUNCTION, CREATE LANGUAGE, CREATE TYPE, DROP TRANSFORM

CREATE TRIGGER

CREATE TRIGGER — создать триггер

Синтаксис

```
CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER | INSTEAD OF }
 { событие [ OR ... ] }
    ON имя_таблицы
    [ FROM ссылающаяся_таблица ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] имя_переходного_отношения } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( условие ) ]
    EXECUTE { FUNCTION | PROCEDURE } имя_функции ( аргументы )

Здесь допускается событие:

    INSERT
    UPDATE [ OF имя_столбца [, ... ] ]
    DELETE
    TRUNCATE
```

Описание

`CREATE TRIGGER` создаёт новый триггер, а `CREATE OR REPLACE TRIGGER` создаёт новый триггер или заменяет существующий. Триггер будет связан с указанной таблицей, представлением или сторонней таблицей и будет выполнять заданную функцию *`имя_функции`* при определённых операциях с этой таблицей.

Чтобы заменить текущее определение существующего триггера, воспользуйтесь командой `CREATE OR REPLACE TRIGGER`, указав в ней имя существующего триггера и родительскую таблицу. Все остальные свойства этого триггера будут изменены.

Триггер можно настроить так, чтобы он срабатывал до операции со строкой (до проверки ограничений и попытки выполнить `INSERT`, `UPDATE` или `DELETE`) или после её завершения (после проверки ограничений и выполнения `INSERT`, `UPDATE` или `DELETE`), либо вместо операции (при добавлении, изменении и удалении строк в представлении). Если триггер срабатывает до или вместо события, он может пропустить операцию с текущей строкой, либо изменить добавляемую строку (только для операций `INSERT` и `UPDATE`). Если триггер срабатывает после события, он «видит» все изменения, включая результат действия других триггеров.

Триггер с пометкой `FOR EACH ROW` вызывается один раз для каждой строки, изменяемой в процессе операции. Например, операция `DELETE`, удаляющая 10 строк, приведёт к срабатыванию всех триггеров `ON DELETE` в целевом отношении 10 раз подряд, по одному разу для каждой удаляемой строки. Триггер с пометкой `FOR EACH STATEMENT`, напротив, вызывается только один раз для конкретной операции, вне зависимости от того, как много строк она изменила (в частности, при выполнении операции, изменяющей ноль строк, всё равно будут вызваны все триггеры `FOR EACH STATEMENT`).

Триггеры, срабатывающие в режиме `INSTEAD OF`, должны быть помечены `FOR EACH ROW` и могут быть определены только для представлений. Триггеры `BEFORE` и `AFTER` для представлений должны быть помечены `FOR EACH STATEMENT`.

Кроме того, триггеры можно определить и для команды `TRUNCATE`, но только типа `FOR EACH STATEMENT`.

В следующей таблице перечисляются типы триггеров, которые могут использоваться для таблиц, представлений и сторонних таблиц:

Когда	Событие	На уровне строк	На уровне оператора
`BEFORE`	`INSERT`/`UPDATE`/`DELETE`	Таблицы и сторонние таблицы	Таблицы, представления и сторонние таблицы
	`TRUNCATE`	—	Таблицы и сторонние таблицы
`AFTER`	`INSERT`/`UPDATE`/`DELETE`	Таблицы и сторонние таблицы	Таблицы, представления и сторонние таблицы
	`TRUNCATE`	—	Таблицы и сторонние таблицы
`INSTEAD OF`	`INSERT`/`UPDATE`/`DELETE`	Представления	—
	`TRUNCATE`	—	—

Кроме того, в определении триггера можно указать логическое условие `WHEN`, которое определит, вызывать триггер или нет. В триггерах на уровне строк условия `WHEN` могут проверять старые и/или новые значения столбцов в строке. Триггеры на уровне оператора так же могут содержать условие `WHEN`, хотя для них это не столь полезно, так как в этом условии нельзя ссылаться на какие-либо значения в таблице.

Если для одного события определено несколько триггеров одного типа, они будут срабатывать в алфавитном порядке их имён.

Когда указывается параметр `CONSTRAINT`, эта команда создаёт *триггер ограничения*. Он подобен обычным триггерам, но отличается тем, что время его срабатывания можно изменить командой `SET CONSTRAINTS`. Триггеры ограничений должны быть триггерами типа `AFTER ROW` для обычных (не сторонних) таблиц. Они могут срабатывать либо в конце оператора, вызвавшего целевое событие, либо в конце содержащей его транзакции; в последнем случае они называются *отложенными*. Срабатывание ожидающего отложенного триггера можно вызвать немедленно, воспользовавшись командой `SET CONSTRAINTS`. Предполагается, что триггеры ограничений будут генерировать исключения при нарушении ограничений.

Когда указывается `REFERENCING`, для триггера собираются *переходные отношения*, представляющие собой множества строк, включающие все строки, которые были добавлены, удалены или изменены текущим оператором SQL. Это позволяет триггеру наблюдать общую картину того, что сделал оператор, а не только одну строку за другой. Это указание допускается только для триггера `AFTER` для обычных (не сторонних) таблиц. Этот триггер не должен быть триггером ограничения. Кроме того, если это триггер для `UPDATE`, у него должен отсутствовать список *имён_столбцов* при использовании этого указания. Указание `OLD TABLE` может быть задано только один раз и только для триггера, который может срабатывать при `UPDATE` или `DELETE`; оно создаёт переходное отношение, содержащее *образы-до-изменения* всех строк, модифицированных или удалённых оператором. Указание `NEW TABLE`, подобным образом, может быть задано только единожды и только для триггера, который может срабатывать для `UPDATE` или `INSERT`; оно создаёт переходное отношение, содержащее *образы-после-изменения* всех строк, модифицированных или добавленных оператором.

`SELECT` не изменяет никакие строки, поэтому создавать триггеры для `SELECT` нельзя. Для решения задач, в которых требуются подобные триггеры, могут подойти правила или представления.

За дополнительными сведениями о триггерах обратитесь к Главе 37.

Параметры

имя

Имя, назначаемое новому триггеру. Это имя должно отличаться от имени любого другого триггера в этой же таблице. Имя не может быть дополнено схемой — триггер наследует схему от своей таблицы. Для триггеров ограничений это имя также используется, когда требуется скорректировать поведение триггера с помощью команды `SET CONSTRAINTS`.

`BEFORE`
`AFTER`
`INSTEAD OF`

Определяет, будет ли заданная функция вызываться до, после или вместо события. Для триггера ограничения можно указать только `AFTER`.

`событие`

Принимает одно из значений: `INSERT`, `UPDATE`, `DELETE` или `TRUNCATE`; этот параметр определяет событие, при котором будет срабатывать триггер. Несколько событий можно указать, добавив между ними слово `OR`, если только не запрашиваются переходные отношения.

Для событий `UPDATE` можно указать список столбцов, используя такую запись:

```
UPDATE OF имя_столбца1 [, имя_столбца2 ... ]
```

Такой триггер сработает, только если в списке столбцов, указанном в целевой команде `UPDATE`, окажется минимум один из перечисленных или если какой-нибудь из них будет генерируемым и при этом зависящим от столбца, который фигурирует в `UPDATE`.

Для событий `INSTEAD OF UPDATE` указание списка столбцов не допускается. Список столбцов также нельзя задать, когда запрашиваются переходные отношения.

`имя_таблицы`

Имя (возможно, дополненное схемой) таблицы, представления или сторонней таблицы, для которых предназначен триггер.

`ссылающаяся_таблица`

Имя (возможно, дополненное схемой) другой таблицы, на которую ссылается ограничение. Оно используется для ограничений внешнего ключа и не рекомендуется для обычного применения. Это указание допускается только для триггеров ограничений.

`DEFERRABLE`
`NOT DEFERRABLE`
`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

Время срабатывания триггера по умолчанию. Подробнее возможные варианты описаны в документации CREATE TABLE. Это указание допускается только для триггеров ограничений.

`REFERENCING`

Это ключевое слово непосредственно предшествует объявлению одного или двух имён, по которым можно будет обращаться к переходным отношениями, образуемым при выполнении целевого оператора.

`OLD TABLE`
`NEW TABLE`

Это предложение указывает, будет ли следующее имя относиться к переходному отношению с образом-до-изменения или к переходному отношению с образом-после-изменения.

`имя_переходного_отношения`

Имя (неполное, без схемы), которое будет использоваться в триггере для обращения к этому переходному отношению.

`FOR EACH ROW`
`FOR EACH STATEMENT`

Определяет, будет ли функция триггера срабатывать один раз для каждой строки, либо для SQL-оператора. Если не указано ничего, подразумевается `FOR EACH STATEMENT` (для оператора). Для триггеров ограничений можно указать только `FOR EACH ROW`.

`условие`

Логическое выражение, определяющее, будет ли выполняться функция триггера. Если для триггера задано указание `WHEN`, функция будет вызываться, только когда *`условие`* возвращает `true`. В триггерах `FOR EACH ROW` условие `WHEN` может ссылаться на значения столбца в старой и/или новой строке, в виде `OLD.`*`имя_столбца`* и `NEW.`*`имя_столбца`*, соответственно. Разумеется, триггеры `INSERT` не могут ссылаться на `OLD`, а триггеры `DELETE` не могут ссылаться на `NEW`.

Триггеры `INSTEAD OF` не поддерживают условия `WHEN`.

В настоящее время выражения `WHEN` не могут содержать подзапросы.

Учтите, что для триггеров ограничений вычисление условия `WHEN` не откладывается, а выполняется немедленно после операции, изменяющей строки. Если результат условия — ложь, сам триггер не откладывается для последующего выполнения.

`имя_функции`

Заданная пользователем функция, объявленная как функция без аргументов и возвращающая тип `trigger`, которая будет вызываться при срабатывании триггера.

В синтаксисе `CREATE TRIGGER` ключевые слова `FUNCTION` и `PROCEDURE` равнозначны, но указываемая триггерная функция должна в любом случае быть функцией, а не процедурой. Ключевое слово `PROCEDURE` здесь поддерживается по историческим причинам и считается устаревшим.

`аргументы`

Необязательный список аргументов через запятую, которые будут переданы функции при срабатывании триггера. В качестве аргументов функции передаются строковые константы. И хотя в этом списке можно записать и простые имена или числовые константы, они тоже будут преобразованы в строки. Порядок обращения к таким аргументам в функции триггера может отличаться от обычных аргументов, поэтому его следует уточнить в описании языка реализации этой функции.

Примечания

Чтобы создать или изменить триггер, пользователь должен иметь право `TRIGGER` для этой таблицы. Также пользователь должен иметь право `EXECUTE` для триггерной функции.

Для удаления триггера применяется команда `DROP TRIGGER`.

При создании триггера уровня строк для секционированной таблицы такой же «клонированный» триггер будет создан в каждой из её существующих секций; идентичный триггер будет установлен и в каждой секции, создаваемой или присоединяемой позднее. Если в дочерней таблице встретится триггер с конфликтующим именем, возникнет ошибка. Исключение составляет команда `CREATE OR REPLACE TRIGGER`, которая заменит существующий одноимённый триггер клонированным. При отсоединения секции от родительской таблицы клонированные триггеры в ней удаляется.

Триггер для избранных столбцов (определённый с помощью `UPDATE OF` *`имя_столбца`*) будет срабатывать, когда его столбцы перечислены в качестве целевых в списке `SET` команды `UPDATE`. Изменения, вносимые в строки триггерами `BEFORE UPDATE`, при этом не учитываются, поэтому значения столбцов можно изменить так, что триггер не сработает. И наоборот, при выполнении команды `UPDATE ... SET x = x ...` триггер для столбца `x` сработает, хотя значение столбца не меняется.

В триггере `BEFORE` условие `WHEN` вычисляется непосредственно перед возможным вызовом функции, поэтому проверка `WHEN` существенно не отличается от проверки того же условия в начале функции триггера. В частности, учтите, что строка `NEW`, которую видит ограничение, содержит текущие значения, возможно изменённые предыдущими триггерами. Кроме того, в триггере `BEFORE` условие `WHEN` не может проверять системные столбцы в строке `NEW` (например, `ctid`), так как они ещё не установлены.

В триггере `AFTER` условие `WHEN` проверяется сразу после изменения строки, и если оно выполняется, событие запоминается, чтобы вызвать триггер в конце оператора. Если же для триггера `AFTER` условие `WHEN` не выполняется, нет необходимости запоминать событие для последующей обработки или заново перечитывать строку в конце оператора. Это приводит к значительному ускорению операторов, изменяющих множество строк, когда триггер должен срабатывать только для некоторых из них.

В некоторых случаях одна команда SQL может вызывать сразу нескольких видов триггеров. Например, `INSERT` с предложением `ON CONFLICT DO UPDATE` может выполнять операции как добавления, так и изменения, так что она при необходимости будет вызывать триггеры обоих видов. При этом переходные отношения, предоставляемые триггерам, будут разными в зависимости от типа события; то есть триггер `INSERT` будет видеть только добавленные строки, а триггер `UPDATE` — только изменённые.

Изменения или удаления строк, вызванные действиями по обеспечению целостности внешнего ключа, например, `ON UPDATE CASCADE` или `ON DELETE SET NULL`, считаются частью SQL-команды, вызвавшей эти действия (заметьте, что такие действия не могут быть отложенными). В затрагиваемой таблице будут вызваны соответствующие триггеры, и таким образом появляется возможность вызова триггеров для SQL-команды, не соответствующей непосредственно их типу. В простых ситуациях триггеры, запрашивающие переходные отношения, будут видеть все изменения, произведённые в их таблице одной исходной командой SQL, в виде одного переходного отношения. Однако возможны случаи, в которых присутствие триггера `AFTER ROW`, запрашивающего переходные отношения, приведёт к тому, что операции для обеспечения целостности внешнего ключа, вызванные одной SQL-командой, будут разделены на несколько этапов, и на каждом будут свои переходные отношения. В таких случаях все существующие триггеры уровня оператора будут срабатывать единожды при создании набора переходных отношений, что гарантирует, что эти триггеры будут видеть каждую обрабатываемую строку в переходном отношении один и только один раз.

Триггеры уровня операторов для представления срабатывают, только если операция с представлением обрабатывается триггером уровня строк `INSTEAD OF`. Если операция обрабатывается правилом `INSTEAD`, то вместо исходного оператора, обращающегося к представлению, выполняются те операторы, что генерирует правило, поэтому вызываться будут триггеры, связанные с таблицами, к которым обращаются эти заменяющие операторы. Аналогично, для автоматически изменяемого представления выполнение операции сводится к переписыванию оператора в виде операции с базовой таблицей представления, так что срабатывать будут триггеры уровня операторов для базовой таблицы.

При изменении данных в секционированной таблице или таблице с потомками срабатывают триггеры уровня оператора, связанные с явно задействованной таблицей, но не триггеры уровня оператора для её секций или дочерних таблиц. Триггеры уровня строк, напротив, срабатывают для строк в затрагиваемых секциях или дочерних таблицах, даже если они явно не присутствуют в запросе. Если триггер уровня оператора был определён с переходными отношениями, названными в указании `REFERENCING`, то в них будут видны образы строк из всех затронутых секций или дочерних таблиц. В случае с потомками в иерархии наследования образы строк будут содержать только столбцы, присутствующие в таблице, с которой связан триггер.

В настоящее время триггеры уровня строки с переходными отношениями нельзя определить для секций или дочерних таблиц в иерархии наследования. Кроме того, для секционированных таблиц нельзя определить триггеры `INSTEAD OF`.

В настоящее время указание `OR REPLACE` не поддерживается для триггеров ограничений.

Заменять существующий триггер в рамках транзакции, которая уже выполнила действие, изменяющее данные в таблице триггера, не рекомендуется. Решения по срабатыванию триггера, как и их составляющие, принятые ранее, не будут пересмотрены, поэтому эффект такой операции может быть неожиданным.

Некоторые общие задачи можно решить с применением встроенных триггерных функций, обойдясь без написания собственного кода; см. Раздел 9.29.

Примеры

Выполнение функции `check_account_update` перед любым изменением строк в таблице `accounts`:

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

Изменение определения триггера, чтобы данная функция выполнялась только при указании столбца `balance` в качестве целевого столбца команды `UPDATE`:

```
CREATE OR REPLACE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

В этом примере функция будет выполняться, если значение столбца `balance` в действительности изменилось:

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE FUNCTION check_account_update();
```

Вызов функции, ведущей журнал изменений в `accounts`, но только если что-то изменилось:

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION log_account_update();
```

Выполнение для каждой строки функции `view_insert_row`, которая будет вставлять строки в нижележащие таблицы представления:

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE FUNCTION view_insert_row();
```

Выполнение функции `check_transfer_balances_to_zero` для каждого оператора, проверяющей, что строки `transfer` в совокупности дают нулевой баланс:

```
CREATE TRIGGER transfer_insert
    AFTER INSERT ON transfer
    REFERENCING NEW TABLE AS inserted
    FOR EACH STATEMENT
    EXECUTE FUNCTION check_transfer_balances_to_zero();
```

Выполнение функции `check_matching_pairs` для каждой строки, проверяющей, что соответствующие пары пунктов изменены синхронно (одним оператором):

```
CREATE TRIGGER paired_items_update
    AFTER UPDATE ON paired_items
    REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
    FOR EACH ROW
    EXECUTE FUNCTION check_matching_pairs();
```

В Разделе 37.4 приведён полный пример функции триггера, написанной на C.

Совместимость

Оператор `CREATE TRIGGER` в Postgres Pro реализует подмножество возможностей, описанных в стандарте SQL. В настоящее время в нём отсутствует следующая функциональность:

- Тогда как имена переходных таблиц для триггеров `AFTER` задаются предложением `REFERENCING` стандартным образом, переменные строк, применяемые в триггерах `FOR EACH ROW` нельзя объявлять в предложении `REFERENCING`. Порядок обращения к таким строкам зависит от языка, на котором написана триггерная функция, но для каждого языка он вполне определённый. Некоторые языки по сути действуют так, как будто в команде присутствует предложение `REFERENCING` с указанием `OLD ROW AS OLD NEW ROW AS NEW`.
- Стандарт позволяет использовать переходные таблицы с триггерами `UPDATE`, ограничивающими набор отслеживаемых столбцов, но тогда и набор строк, видимых в переходных таблицах, должен зависеть от списка целевых столбцов триггера. В настоящее время такое поведение в Postgres Pro не реализовано.
- Postgres Pro позволяет задать в качестве действия триггера только функцию, определённую пользователем. Стандарт допускает также выполнение ряда других команд SQL, например, `CREATE TABLE`. Однако это ограничение несложно преодолеть, создав пользовательскую функцию, выполняющую требуемые команды.

В стандарте SQL определено, что несколько триггеров должны срабатывать по порядку создания. Postgres Pro упорядочивает их по именам, так как это было признано более удобным.

В стандарте SQL определено, что триггеры `BEFORE DELETE` при каскадном удалении срабатывают *после* завершения каскадного `DELETE`. В Postgres Pro триггеры `BEFORE DELETE` всегда срабатывают перед операцией удаления, даже если она каскадная. Это поведение выбрано как более логичное. Ещё одно отклонение от стандарта проявляется, когда триггеры `BEFORE`, срабатывающие в результате ссылочной операции, изменяют строки или не дают выполнить изменение. Это может привести к нарушению ограничений или сохранению данных, не соблюдающих ссылочную целостность.

Возможность задать несколько действий для одного триггера с помощью ключевого слова `OR` — реализованное в Postgres Pro расширение стандарта SQL.

Возможность вызывать триггеры для `TRUNCATE` — реализованное в Postgres Pro расширение стандарта SQL, как и возможность определять триггеры на уровне оператора для представлений.

`CREATE CONSTRAINT TRIGGER` — реализованное в Postgres Pro расширение стандарта SQL, так же как и указание `OR REPLACE`.

См. также

ALTER TRIGGER, DROP TRIGGER, CREATE FUNCTION, SET CONSTRAINTS

CREATE TYPE

CREATE TYPE — создать новый тип данных

Синтаксис

```
CREATE TYPE имя AS
    ( [ имя_атрибута тип_данных [ COLLATE правило_сортировки ] [, ... ] ] )

CREATE TYPE имя AS ENUM
    ( [ 'метка' [, ... ] ] )

CREATE TYPE имя AS RANGE (
    SUBTYPE = подтип
    [ , SUBTYPE_OPCLASS = класс_оператора_подтипа ]
    [ , COLLATION = правило_сортировки ]
    [ , CANONICAL = функция_нормализации ]
    [ , SUBTYPE_DIFF = функция_разницы_подтипа ]
    [ , MULTIRANGE_TYPE_NAME = имя_мультидиапазонного_типа ]
)

CREATE TYPE имя (
    INPUT = функция_ввода,
    OUTPUT = функция_вывода
    [ , RECEIVE = функция_получения ]
    [ , SEND = функция_отправки ]
    [ , TYPMOD_IN = функция_ввода_модификатора_типа ]
    [ , TYPMOD_OUT = функция_вывода_модификатора_типа ]
    [ , ANALYZE = функция_анализа ]
    [ , SUBSCRIPT = функция_обращения_по_индексу ]
    [ , INTERNALLENGTH = { внутр_длина | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = выравнивание ]
    [ , STORAGE = хранение ]
    [ , LIKE = тип_образец ]
    [ , CATEGORY = категория ]
    [ , PREFERRED = предпочитаемый ]
    [ , DEFAULT = default ]
    [ , ELEMENT = элемент ]
    [ , DELIMITER = разделитель ]
    [ , COLLATABLE = сортируемый ]
)

CREATE TYPE имя
```

Описание

`CREATE TYPE` регистрирует новый тип данных для использования в текущей базе данных. Владельцем типа становится создавший его пользователь.

Если указано имя схемы, тип создаётся в указанной схеме. В противном случае он создаётся в текущей схеме. Имя типа должно отличаться от имён любых других существующих типов или доменов в той же схеме. (А так как с таблицами связываются типы данных, имя типа должно также отличаться и от имён существующих таблиц в этой схеме.)

Команда `CREATE TYPE` имеет пять форм, показанных выше в сводке синтаксиса. Они создают соответственно *составной тип*, *перечисление*, *диапазон*, *базовый тип* или *тип-пустышку*. Первые четыре эти типа рассматриваются по порядку ниже. Тип-пустышка представляет собой просто заготовку для типа, который будет определён позже; он создаётся командой `CREATE TYPE` с одним именем, без параметров. Типы-пустышки необходимы для определения прямых ссылок при создании базовых типов и типов-диапазонов, как описывается в соответствующих разделах.

Составные типы

Первая форма `CREATE TYPE` создаёт составной тип. Составной тип задаётся списком имён и типами данных атрибутов. Если тип данных является сортируемым, то для атрибута можно также задать правило сортировки. Составной тип по сути не отличается от типа строки таблицы, но `CREATE TYPE` избавляет от необходимости создавать таблицу, когда всё, что нужно, это создать тип. Отдельный составной тип может быть полезен, например, для передачи аргументов или результатов функции.

Чтобы создать составной тип, необходимо иметь право `USAGE` для типов всех его атрибутов.

Типы перечислений

Вторая форма `CREATE TYPE` создаёт тип-перечисление (такие типы описываются в Разделе 8.7). Перечисления принимают список меток в кавычках. Максимальная длина каждой метки — `NAMEDATALEN` байт (64 байта в стандартной сборке Postgres Pro). (Также возможно создать перечисляемый тип без меток, но этот тип нельзя будет использовать для хранения значений, пока командой `ALTER TYPE` не будет добавлена хотя бы одна метка.)

Диапазонные типы

Третья форма `CREATE TYPE` создаёт тип-диапазон (такие типы описываются в Разделе 8.17).

Задаваемый для диапазона *`подтип`* может быть любым типом со связанным классом операторов B-дерева (что позволяет определить порядок значений в диапазоне). Обычно порядок элементов определяет класс операторов B-дерева по умолчанию, но его можно изменить, задав имя другого класса в параметре *`класс_операторов_подтипа`*. Если подтип поддерживает сортировку и требуется, чтобы значения упорядочивались с нестандартным правилом сортировки, его имя можно задать в параметре *`правило_сортировки`*.

Необязательная *`функция_нормализации`* должна принимать один аргумент определяемого типа диапазона и возвращать значение того же типа. Она используется для преобразования значений диапазона в нормализованную форму, когда это уместно. За дополнительными сведениями обратитесь к Подразделу 8.17.8. Создаётся *`функция_нормализации`* несколько нетривиально, так как она должна быть уже определена, прежде чем можно будет объявить тип-диапазон. Для этого нужно сначала создать тип-пустышку, который будет заготовкой типа, не имеющей никаких свойств, кроме имени и владельца. Это можно сделать, выполнив команду `CREATE TYPE` *`имя`* без дополнительных параметров. Затем можно объявить функцию, для которой тип-пустышка будет типом аргумента и результата, и, наконец, объявить тип-диапазон с тем же именем. При этом тип-пустышка автоматически заменится полноценным типом-диапазоном.

Необязательная *`функция_разницы_подтипа`* должна принимать в аргументах два значения типа *`подтип`* и возвращать значение `double precision`, представляющее разницу между двумя данными значениями. Хотя эту функцию можно не использовать, она позволяет кардинально увеличить эффективность индексов GiST для столбцов с типом-диапазоном. За дополнительными сведениями обратитесь к Подразделу 8.17.8.

В необязательном параметре *`имя_мультидиапазонного_типа`* задаётся имя соответствующего типа. В отсутствие данного параметра имя этого типа выбирается автоматически следующим образом. Если имя диапазонного типа содержит подстроку `range`, в имени мультидиапазонного типа она заменяется подстрокой `multirange`. В противном случае к имени диапазонного типа добавляется окончание `_multirange`.

Базовые типы

Четвёртая форма `CREATE TYPE` создаёт новый базовый тип (скалярный тип). Чтобы создать новый базовый тип, нужно быть суперпользователем. (Это ограничение введено потому, что ошибочное определение типа может вызвать нарушения или даже сбой в работе сервера.)

Эти параметры могут перечисляться в любом порядке, не только в показанном выше, и большинство из них необязательные. Прежде чем создавать тип, необходимо зарегистрировать две или более функций (с помощью `CREATE FUNCTION`). Обязательными являются функции *`функция_ввода`* и *`функция_вывода`*, тогда как *`функция_получения`*, *`функция_отправки`*, *`функция_модификатора_типа`*, *`функция_вывода_модификатора_типа`*, *`функция_анализа`* и *`функция_обращения_по_индексу`* могут отсутствовать. Обычно эти функции разрабатываются на C или другом низкоуровневом языке.

`Функция_ввода` преобразует внешнее текстовое представление типа во внутреннее, с которым работают операторы и функции, определённые для этого типа. *`Функция_вывода`* выполняет обратное преобразование. Функцию ввода можно объявить как принимающую один аргумент типа `cstring`, либо как принимающую три аргумента типов `cstring`, `oid` и `integer`. В первом аргументе передаётся вводимый текст в виде строки в стиле C, во втором аргументе — собственный OID типа (кроме типов массивов, для которых передаётся OID типа элемента), а в третьем — `модификатор_типа` для целевого столбца, если он определён (или -1 в противном случае). Функция ввода должна возвращать значение нового типа данных. Обычно функция ввода должна быть строгой (STRICT); если это не так, при получении на вход значения NULL она будет вызываться с первым параметром NULL. Функция может в этом случае сама вернуть NULL или вызвать ошибку. (Это полезно в основном для поддержки функций ввода доменных типов, которые не должны принимать данные NULL.) Функция вывода должна принимать один аргумент нового типа данных, а возвращать она должна `cstring`. Для значений NULL функции вывода не вызываются.

Необязательная *`функция_получения`* преобразует двоичное внешнее представление типа во внутреннее представление. Если эта функция отсутствует, новый тип не сможет участвовать в двоичном вводе. Двоичное представление следует выбирать таким, чтобы оно легко переводилось во внутреннюю форму и при этом было переносимым до разумной степени. (Например, для стандартных целочисленных типов данных во внешнем двоичном представлении выбран сетевой порядок байтов, тогда как внутреннее представление определяется порядком байтов в процессоре.) Функция получения должна выполнить проверку вводимого значения на допустимость. Функция получения может быть объявлена как принимающая один аргумент типа `internal`, либо как принимающая три аргумента типов `internal`, `oid` и `integer`. В первом аргументе передаётся указатель на буфер `StringInfo`, содержащий полученную байтовую строку, а дополнительные аргументы такие же, как и для функции ввода текста. Функция получения должна возвращать значение нового типа данных. Обычно функция получения должна быть строгой (STRICT); если это не так, при получении на вход значения NULL, она будет вызываться с первым параметром NULL. Функция может в этом случае сама вернуть NULL или вызывать ошибку. (Это полезно в основном для поддержки функций получения доменных типов, которые не должны принимать значения NULL.) Подобным образом, необязательная *`функция_отправки`* преобразует данные из внутреннего во внешнее двоичное представление. Если эта функция не определена, новый тип не может участвовать в двоичном выводе. Функция отправки должна принимать один аргумент нового типа данных, а возвращать она должна `bytea`. Для значений NULL функции отправки не вызываются.

Здесь у вас может возникнуть вопрос, как функции ввода и вывода могут быть объявлены принимающими или возвращающими значения нового типа, если они должны быть созданы до объявления нового типа. Ответ довольно прост: сначала нужно создать *тип-пустышку*, который будет заготовкой типа, не имеющей никаких свойств, кроме имени и владельца. Это можно сделать, выполнив команду `CREATE TYPE` *`имя`* без дополнительных параметров. Затем можно будет определить функции ввода-вывода на C, ссылающиеся на этот тип. И наконец, команда `CREATE TYPE` с полным определением заменит тип-пустышку окончательным и полноценным определением, после чего новый тип можно будет использовать как обычно.

Необязательные *`функция_ввода_модификатора_типа`* и *`функция_вывода_модификатора_типа`* требуются, только если типы поддерживают модификаторы, или, другими словами, дополнительные ограничения, связываемые с объявлением типа, например `char(5)` или `numeric(30,2)`. В Postgres Pro типы могут принимать в качестве модификаторов одну или несколько простых констант или идентификаторов. Однако эти данные должны упаковываться в единственное неотрицательное целочисленное значение, которое и будет храниться в системных каталогах. *`Функция_ввода_модификатора_типа`* получает объявленные модификаторы в виде строки `cstring`. Она должна проверить значения на допустимость (и вызвать ошибку, если они неверны), а затем выдать неотрицательное значение `integer`, которое будет сохранено в столбце «typmod». Если для типа не определена *функция_ввода_модификатора_типа*, модификаторы типа приниматься не будут. *Функция_вывода_модификатора_типа* преобразует внутреннее целочисленное значение typmod обратно, в форму, понятную пользователю. Она должна вернуть значение `cstring`, которое именно в этом виде будет добавлено к имени типа; например, функция для `numeric` должна вернуть `(30,2)`. *Функция_вывода_модификатора_типа* может быть опущена, в этом случае сохранённое целочисленное значение typmod по умолчанию будет выводиться просто в виде числа, заключённого в скобки.

Необязательная *функция_анализа* выполняет сбор специфической для этого типа статистики в столбцах с таким типом данных. По умолчанию `ANALYZE` пытается собрать статистику, используя операторы «равно» и «меньше», если для этого типа определён класс операторов B-дерева по умолчанию. Для нескалярных типов это поведение скорее всего не подойдёт, поэтому его можно переопределить, задав собственную функцию анализа. Эта функция должна принимать единственный аргумент типа `internal` и возвращать результат `boolean`.

Необязательное указание *функции_обращения_по_индексу* позволяет добавить для типа возможность обращения по индексу в SQL-командах. Указание этой функции не делает тип «настоящим» массивом; например, такой тип не будет рассматриваться как возможный тип результата конструкций `ARRAY[]`. Но если извлечение значений при обращении по индексу для некоторого типа выглядит естественным, заданная *функция_обращения_по_индексу* позволяет определить, что именно подразумевается под таким обращением. Данная функция должна объявляться как принимающая один аргумент типа `internal` и возвращающая результат типа `internal`, в котором передаётся указатель на структуру с методами (функциями), реализующими обращение по индексу. В деталях API этих методов описывается в *src/include/nodes/subscripting.h*. Дополнительная информация приведена в Типы массивов ниже.

Если особенности внутреннего представления нового типа известны функциям ввода-вывода и другим функциям, созданным специально для работы с этим типом, необходимо определить ряд характеристик внутреннего представления, о которых должен знать Postgres Pro. В первую очередь это *internallength* (внутренняя длина). Если базовый тип данных имеет фиксированную длину, в *internallength* указывается эта длина в виде положительного числа, а если длина переменная, в *internallength* задаётся значение `VARIABLE`. (Внутри при этом `typlen` принимает значение -1.) Внутреннее представление всех типов переменной длины должно начинаться с 4-байтового целого, задающего общую длину значения этого типа. (Заметьте, что поле длины часто кодируется, как описано в Разделе 63.2; обращаться к нему напрямую неразумно.)

Необязательный флаг `PASSEDBYVALUE` указывает, что значения этого типа данных передаются по значению, а не по ссылке. Типы, передаваемые по значению, должны быть фиксированной длины и их внутреннее представление не может быть больше размера типа `Datum` (4 байта на одних машинах, 8 — на других).

Параметр *выравнивание* определяет, как требуется выравнивать данные этого типа. Допускается выравнивание по границам 1, 2, 4 или 8 байт. Заметьте, что типы переменной длины должны быть выровнены как минимум по границе 4 байт, так как их первым компонентом обязательно должен быть `int4`.

Параметр *хранение* позволяет выбрать стратегию хранения для типов данных переменной длины. (Для типов с фиксированной длиной поддерживается только вариант `plain`.) Если выбрана стратегия `plain`, данные этого типа всегда хранятся внутри, без сжатия. Со стратегией `extended` система сначала попытается сжать большое значение, а затем выносит его из строки основной таблицы, если оно всё же окажется слишком большим. С `external` значение может быть вынесено из основной таблицы, но система не будет пытаться сжать его. Стратегия `main` позволяет сжать данные, но не стремится вынести их из основной таблицы. (Элементы данных с этой стратегией хранения тем не менее могут быть вынесены из основной таблицы, если другого способа уместить их в строке нет, но всё же она отдаёт большее предпочтение основной таблице, по сравнению со стратегиями `extended` и `external`.)

Значения параметра *хранение*, отличные от `plain`, подразумевают, что функции типа данных могут принимать значения в формате *toast*, описанном в Разделе 63.2 и Подразделе 36.13.1. Эти значения просто определяют стратегию хранения TOAST по умолчанию для столбцов отделяемого в TOAST типа данных; пользователи могут выбирать другие стратегии для отдельных столбцов, применяя команду `ALTER TABLE SET STORAGE`.

Параметр *тип_образец* позволяет задать основные свойства представления типа другим способом: скопировать их из существующего типа. В частности, из указанного типа будут скопированы свойства *internallength*, *passedbyvalue*, *alignment* и *storage*. (Также возможно, хотя обычно это не требуется, переопределить некоторые из этих значений, указав их вместе с предложением `LIKE`.) Определять представление типа таким образом особенно удобно, когда низкоуровневая реализация нового типа некоторым образом опирается на существующий тип.

Параметры *категория* и *предпочитаемый* позволяют определять, какое неявное приведение будет применяться в неоднозначных ситуациях. Каждый тип данных принадлежит к некоторой категории, обозначаемой одним символом ASCII, при этом он может быть, либо не быть «предпочитаемым» в этой категории. Анализатор запроса по возможности выберет приведение к предпочитаемому типу (но только среди других типов той же категории), когда это может помочь разрешить имя перегруженной функции или оператора. За дополнительными подробностями обратитесь к Главе 10. Если для типа не определено неявное приведение к какому-либо другому типу или обратное, для этих параметров достаточно оставить значения по умолчанию. Однако если есть группа связанных типов, для которых определены неявные приведения, часто бывает полезно пометить их все как принадлежащие некоторой категории и назначить один или два «наиболее общих» предпочитаемыми в этой категории. Параметр *категория* особенно полезен при добавлении типа, определённого пользователем, в существующую встроенную категорию, например, в категорию числовых или строковых типов. Однако так же возможно создать категории типов, полностью определённые пользователем. В качестве имени такой категории можно выбрать любой ASCII-символ, кроме латинской заглавной буквы.

Если пользователь хочет назначить столбцам с этим типом данных значение по умолчанию, отличное от NULL, он может задать его в этой команде, указав его после ключевого слова `DEFAULT`. (Такое значение по умолчанию можно переопределить явным предложением `DEFAULT`, добавленным при создании столбца.)

Чтобы обозначить, что тип является массивом фиксированной длины, укажите тип элементов массива, воспользовавшись ключевым словом `ELEMENT`. Например, чтобы определить массив из четырёхбайтовых целых (`int4`), укажите `ELEMENT = int4`. За подробностями обратитесь к Типы массивов.

Параметр *delimiter* позволяет задать разделитель, который будет вставляться между значениями во внешнем представлении массива с элементами этого типа. По умолчанию разделителем является запятая (`,`). Заметьте, что разделитель связывается с типом элементов массива, а не с типом самого массива.

Если необязательный логический параметр *сортируемый* равен true, определения столбцов и выражения с этим типом могут включать указания о порядке сортировки, в предложении `COLLATE`. Как именно будут использоваться эти указания, зависит от реализации функций, работающих с этим типом; эти указания не действуют автоматически просто от того, что тип помечен как сортируемый.

Типы массивов

При создании любого нового типа Postgres Pro автоматически создаёт соответствующий тип массива, имя которого он получает, добавляя подчёркивание перед именем типа элементов. Если полученное имя оказывается не короче `NAMEDATALEN` байт, оно усекается. (Если полученное таким образом имя конфликтует с именем уже существующего типа, процесс повторяется, пока не будет получено уникальное имя.) Этот неявно создаваемый тип массива имеет переменную длину и использует встроенные функции ввода и вывода `array_in` и `array_out`. Более того, этот тип используется системой при обработке конструкций ARRAY[] с типом, созданным пользователем. Тип массива отражает любые изменения владельца или схемы связанного типа элемента и удаляется сам при удалении типа элемента.

Вы можете вполне резонно спросить, зачем нужен параметр `ELEMENT`, если система создаёт правильный тип массива автоматически. В основном параметр `ELEMENT` полезен в случае, когда вы создаёте тип фиксированной длины, который внутри оказывается массивом одинаковых элементов, и вы хотите, чтобы к этим элементам можно было обращаться по индексу, помимо того, что вы можете реализовать какие угодно операции с типом в целом. Например, тип `point` представлен просто как два числа с плавающей точкой, к которым можно обратиться так: `point[0]` и `point[1]`. Заметьте, что это работает только с типами фиксированной длины, которые представляют собой в точности последовательность одинаковых полей фиксированной длины. По историческим причинам (т. е. это определённо некорректно, но менять уже слишком поздно), индексы в массивах фиксированной длины начинаются с нуля, а не с 1, как в массивах переменной длины.

Указание `SUBSCRIPT` позволяет добавить для типа данных поддержку обращения по индексу, даже для типов, которые система не считает массивами. Поведение, описанное выше для массивов фиксированной длины, на самом деле реализуется функцией `raw_array_subscript_handler`, заданной в качестве обработчика `SUBSCRIPT` — она используется автоматически, если для типа фиксированной длины задан параметр `ELEMENT`, но не задан `SUBSCRIPT`.

Когда задаётся пользовательская функция `SUBSCRIPT`, указывать также `ELEMENT` обязательно, только если функция-обработчик `SUBSCRIPT` должна узнать `typelem`, чтобы определить возвращаемый тип. Учтите, что с указанием `ELEMENT` система будет предполагать, что новый тип содержит тип элемента либо каким-либо образом физически зависит от него; например, изменить свойства типа элемента нельзя, если созданы какие-либо столбцы зависимого типа.

Параметры

имя

Имя создаваемого типа (возможно, дополненное схемой).

имя_атрибута

Имя атрибута (столбца) составного типа.

тип_данных

Имя существующего типа данных, который станет типом столбца составного типа.

правило_сортировки

Имя существующего правила сортировки, связываемого со столбцом составного типа или с типом-диапазоном.

метка

Строковая константа, представляющая текстовую метку, связанную с отдельным значением типа-перечисления.

подтип

Имя типа элемента, множество значений которого будет представлять тип-диапазон.

класс_оператора_подтипа

Имя класса операторов B-дерева для подтипа.

функция_нормализации

Имя функции нормализации для типа-диапазона.

функция_разницы_подтипа

Имя функции разницы для значений подтипа.

имя_мультидиапазонного_типа

Имя соответствующего мультидиапазонного типа.

функция_ввода

Имя функции, преобразующей данные из внешнего текстового представления типа во внутреннюю форму.

функция_вывода

Имя функции, преобразующей данные из внутренней формы во внешнее текстовое представление типа.

функция_получения

Имя функции, преобразующей данные из внешнего двоичного представления типа во внутреннюю форму.

функция_отправки

Имя функции, преобразующей данные из внутренней формы во внешнее двоичное представление типа.

функция_ввода_модификатора_типа

Имя функции, преобразующей массив модификаторов типа во внутреннюю форму.

функция_вывода_модификатора_типа

Имя функции, преобразующей внутреннюю форму модификаторов типа во внешнее текстовое представление.

функция_анализа

Имя функции, производящей статистический анализ типа данных.

функция_обращения_по_индексу

Имя функции, определяющей, как для типа данных работает извлечение значения по индексу.

внутр_длина

Числовая константа, задающая размер внутреннего представления нового типа в байтах. По умолчанию предполагается, что тип имеет переменную длину.

выравнивание

Требуемое выравнивание для типа данных. Допустимые значения этого параметра, если он указывается: `char`, `int2`, `int4` или `double`; по умолчанию подразумевается `int4`.

хранение

Стратегия хранения для типа данных. Допустимые значения этого параметра, если он указывается: `plain`, `external`, `extended` или `main`; по умолчанию подразумевается `plain`.

тип_образец

Имя существующего типа данных, от которого новый тип получит свойства представления. Из этого типа будут скопированы значения параметров *internallength*, *passedbyvalue*, *alignment* и *storage*, если их не переопределят явные указания, заданные дополнительно в этой команде `CREATE TYPE`.

категория

Код категории (один символ ASCII) для этого типа. По умолчанию подразумевается `'U'` (что означает пользовательский тип, «User-defined»). Коды других стандартных категорий можно найти в Таблице 51.67. Для нестандартных категорий можно выбрать другие ASCII-символы.

`предпочитаемый`

Если значение этого параметра равно true, создаваемый тип будет предпочитаемым в своей категории. По умолчанию подразумевается false. Будьте очень осторожны, создавая новый предпочитаемый тип в существующей категории, так как это может поменять поведение выражений неожиданным образом.

`default`

Значение по умолчанию для создаваемого типа данных. Если не указано, значением по умолчанию будет NULL.

`элемент`

Создаваемый тип будет массивом; этот параметр определяет тип элементов массива.

`разделитель`

Символ, разделяющий значения в массивах, образованных из значений создаваемого типа.

`сортируемый`

Если значение этого параметра равно true, в операциях с создаваемым типом может учитываться информация о правилах сортировки. По умолчанию подразумевается false.

Примечания

Так как на использование типа данных после создания не накладываются ограничения, объявление базового типа или типа-диапазона по сути даёт всем право на выполнение функций, упомянутых в определении типа. Обычно это не проблема для таких функций, какие бывают полезны в определении типов. Но прежде чем создать тип, преобразование которого во внешнюю форму и обратно будет использовать «секретную» информацию, стоит подумать дважды.

В PostgreSQL до версии 8.3 имя генерируемого типа-массива всегда образовалось из имени типа элемента и добавленного спереди символа подчёркивания (_). (Таким образом, допустимая максимальная длина имени типа была на символ меньше, чем длины других имён.) Хотя и сейчас имя типа массива чаще всего образуется таким образом, оно может быть и другим в случае достижения максимальной длины или конфликтов с именами пользовательских типов, начинающихся с подчёркивания. Поэтому полагаться на это соглашение в коде не рекомендуется. Вместо этого, имя типа массива, связанного с данным типом, следует определять по значению `pg_type.typarray`.

Вообще же можно посоветовать не использовать имена типов и таблиц, начинающиеся с подчёркивания. Хотя сервер сможет сгенерировать другое имя, не конфликтующее с пользовательским, некоторая путаница всё же возможна, особенно со старыми клиентскими приложениями, которые могут полагать, что имя типа, начинающееся с подчёркивания, всегда относится к типу массива.

В PostgreSQL до версии 8.2 у `CREATE TYPE` *`имя`* отсутствовала форма для создания типа-пустышки. Поэтому для создания нового базового типа требовалось сначала создать функцию ввода. При таком подходе Postgres Pro воспринимал тип возврата функции ввода как имя нового типа данных и неявно создавал тип-пустышку, на который затем можно было ссылаться в определениях остальных функций ввода-вывода. Этот подход по-прежнему работает, но считается устаревшим и может быть запрещён в будущих версиях. Кроме того, во избежание непреднамеренного заполнения каталогов типами-пустышками, появляющимися в результате простых опечаток в определении функций, тип-пустышка будет создаваться таким образом, только если функция ввода написана на C.

В Postgres Pro версии 16 и выше желательно, чтобы функции ввода базовых типов возвращали некритические ошибки с использованием новой функции `errsave()`/`ereturn()` вместо вызова исключений `ereport()`, как в предыдущих версиях.

Примеры

В этом примере создаётся составной тип, а затем он используется в определении функции:

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

В этом примере создаётся тип-перечисление, а затем он используется в определении таблицы:

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

В этом примере создаётся тип-диапазон:

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

В следующем примере создаётся базовый тип данных `box`, а затем он используется в определении таблицы:

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

Если бы внутренней структурой `box` был массив из четырёх элементов `float4`, вместо этого можно было бы использовать определение:

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

В таком случае к числам, составляющим значение этого типа, можно было бы обращаться по индексу. В остальном поведение этого типа будет таким же.

В этом примере создаётся тип большого объекта, а затем он используется в определении таблицы:

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

Другие примеры, в том числе демонстрирующие подходящие функции ввода-вывода, можно найти в Разделе 36.13.

Совместимость

Первая форма команды `CREATE TYPE`, создающая составной тип, соответствует стандарту SQL. Другие формы являются расширениями Postgres Pro. Для оператора `CREATE TYPE` в стандарте SQL также определены другие формы, не реализованные в Postgres Pro.

Возможность создавать составной тип без атрибутов — специфическое отклонение Postgres Pro от стандарта (как и аналогичная особенность команды `CREATE TABLE`).

См. также

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

CREATE USER

CREATE USER — создать роль в базе данных

Синтаксис

```
CREATE USER имя [ [ WITH ] параметр [ ... ] ]

Здесь параметр:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль' | PASSWORD NULL
    | VALID UNTIL 'дата_время'
    | IN ROLE имя_роли [, ...]
    | IN GROUP имя_роли [, ...]
    | ROLE имя_роли [, ...]
    | ADMIN имя_роли [, ...]
    | USER имя_роли [, ...]
    | SYSID uid
    | PROFILE имя_профиля
```

Описание

Команда `CREATE USER` теперь является просто синонимом `CREATE ROLE`. Единственное отличие в том, что для команды, записанной в виде `CREATE USER`, по умолчанию подразумевается `LOGIN`, а в виде `CREATE ROLE` подразумевается `NOLOGIN`.

Совместимость

Оператор `CREATE USER` является расширением Postgres Pro. В стандарте SQL определение пользователей считается зависимым от реализации.

См. также

CREATE ROLE

CREATE USER MAPPING

CREATE USER MAPPING — создать сопоставление пользователя для стороннего сервера

Синтаксис

```
CREATE USER MAPPING [ IF NOT EXISTS ] FOR { имя_пользователя | USER | CURRENT_ROLE |
 CURRENT_USER | PUBLIC }
    SERVER имя_сервера
    [ OPTIONS ( параметр 'значение' [ , ... ] ) ]
```

Описание

`CREATE USER MAPPING` создаёт сопоставление пользователя на внешнем сервере. Сопоставление пользователя обычно содержит информацию о подключении, которую будет использовать обёртка сторонних данных вместе с информацией о стороннем сервере для получения доступа к внешнему ресурсу.

Владелец стороннего сервера может создать сопоставление для любых пользователей на этом сервере. Кроме того, пользователь может создать сопоставление для своего собственного имени пользователя, если он наделён правом `USAGE` на данном сервере.

Параметры

`IF NOT EXISTS`

Не считать ошибкой, если сопоставление данного пользователя для данного стороннего сервера уже существует. В этом случае будет выдано замечание. Заметьте, что нет никакой гарантии, что существующее сопоставление как-то соотносится с тем, которое могло бы быть создано.

`имя_пользователя`

Имя существующего пользователя, для которого создаётся сопоставление на стороннем сервере. Ключевые слова `CURRENT_ROLE`, `CURRENT_USER` и `USER` обозначают имя текущего пользователя. Если указывается `PUBLIC`, создаётся так называемое общее сопоставление, которое будет использоваться при отсутствии сопоставления для конкретного пользователя.

`имя_сервера`

Имя существующего сервера, для которого создаётся сопоставление пользователя.

`OPTIONS ( параметр 'значение' [, ... ] )`

В этом предложении задаются параметры сопоставления. Эти параметры обычно определяют фактическое имя и пароль пользователя на целевом сервере. Имена параметров должны быть уникальными. Набор допустимых имён и значений параметров определяется обёрткой сторонних данных внешнего сервера.

Примеры

Создание сопоставления для пользователя `bob` на сервере `foo`:

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

Совместимость

`CREATE USER MAPPING` соответствует стандарту ISO/IEC 9075-9 (SQL/MED).

См. также

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

CREATE VIEW

CREATE VIEW — создать представление

Синтаксис

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW имя [ ( имя_столбца
 [, ...] ) ]
    [ WITH ( имя_параметра_представления [= значение_параметра_представления]
 [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Описание

`CREATE VIEW` создаёт представление запроса. Создаваемое представление лишено физической материализации, поэтому указанный запрос будет выполняться при каждом обращении к представлению.

Команда `CREATE OR REPLACE VIEW` действует подобным образом, но если представление с этим именем уже существует, оно заменяется. Новый запрос должен выдавать те же столбцы, что выдавал запрос, ранее определённый для этого представления (то есть, столбцы с такими же именами должны иметь те же типы данных и следовать в том же порядке), но может добавить несколько новых столбцов в конце списка. Вычисления, в результате которых формируются столбцы представления, могут быть совершенно другими.

Если задано имя схемы (например, `CREATE VIEW myschema.myview ...`), представление создаётся в указанной схеме, в противном случае — в текущей. Временные представления существуют в специальной схеме, так что при создании таких представлений имя схемы задать нельзя. Имя представления должно отличаться от имён других отношений (таблиц, последовательностей, индексов, представлений, материализованных представлений или сторонних таблиц) в этой схеме.

Параметры

`TEMPORARY` или `TEMP`

С таким указанием представление создаётся как временное. Временные представления автоматически удаляются в конце сеанса. Существующее постоянное представление с тем же именем не будет видно в текущем сеансе, пока существует временное, однако к нему можно обратиться, дополнив имя указанием схемы.

Если в определении представления задействованы временные таблицы, представление так же создаётся как временное (вне зависимости от присутствия явного указания `TEMPORARY`).

`RECURSIVE`

Создаёт рекурсивное представление. Синтаксис

```
CREATE RECURSIVE VIEW [ схема . ] имя (имена_столбцов) AS SELECT ...;
```

равнозначен

```
CREATE VIEW [ схема . ] имя AS WITH RECURSIVE имя (имена_столбцов) AS (SELECT ...)
 SELECT имена_столбцов FROM имя;
```

Для рекурсивного представления обязательно должен задаваться список с именами столбцов.

имя

Имя создаваемого представления (возможно, дополненное схемой).

`имя_столбца`

Необязательный список имён, назначаемых столбцам представления. Если отсутствует, имена столбцов формируются из результатов запроса.

`WITH ( имя_параметра_представления [= значение_параметра_представления] [, ... ] )`

В этом предложении могут задаваться следующие необязательные параметры представления:

`check_option` (enum)

Этот параметр может принимать значение `local` (локально) или `cascaded` (каскадно) и равнозначен указанию `WITH [ CASCADED | LOCAL ] CHECK OPTION` (см. ниже).

`security_barrier` (boolean)

Этот параметр следует использовать, если представление должно обеспечивать защиту на уровне строк. За дополнительными подробностями обратитесь к Разделу 39.5.

`security_invoker` (boolean)

При обращении к представлению, имеющему эту характеристику, для нижележащих базовых отношений будут проверяться права обращающегося пользователя, а не владельца представления. Подробнее об этом рассказывается в замечаниях ниже.

Все вышеуказанные параметры можно изменить для существующих представлений командой `ALTER VIEW`.

query

Команда `SELECT` или `VALUES`, которая выдаёт столбцы и строки представления.

`WITH [ CASCADED | LOCAL ] CHECK OPTION`

Это указание управляет поведением автоматически изменяемых представлений. Если оно присутствует, при выполнении операций `INSERT`, `UPDATE` и `MERGE` с этим представлением будет проверяться, удовлетворяют ли новые строки условию, определяющему представление (то есть, проверяется, будут ли новые строки видны через это представление). Если они не удовлетворяют условию, операция не будет выполнена. Если указание `CHECK OPTION` отсутствует, команды `INSERT` и `UPDATE` смогут создавать в этом представлении строки, которые не будут видны в нём. Поддерживаются следующие варианты проверки:

`LOCAL`

Новые строки проверяются только по условиям, определённым непосредственно в самом представлении. Любые условия, определённые в нижележащих базовых представлениях, не проверяются (если только в них нет указания `CHECK OPTION`).

`CASCADED`

Новые строки проверяются по условиям данного представления и всех нижележащих базовых. Если указано `CHECK OPTION`, а `LOCAL` и `CASCADED` опущено, подразумевается указание `CASCADED`.

Указание `CHECK OPTION` нельзя использовать с рекурсивными представлениями.

Заметьте, что `CHECK OPTION` поддерживается только для автоматически изменяемых представлений, не имеющих триггеров `INSTEAD OF` и правил `INSTEAD`. Если автоматически изменяемое представление определено поверх базового представления с триггерами `INSTEAD OF`, то для проверки ограничений автоматически изменяемого представления можно применить указание `LOCAL CHECK OPTION`, хотя условия базового представления с триггерами `INSTEAD OF` при этом проверяться не будут (каскадная проверка не будет спускаться к представлению, модифицируемому триггером, и любые параметры проверки, определённые для такого представления, будут просто игнорироваться). Если для представления или любого из его базовых отношений определено правило `INSTEAD`, приводящее к перезаписи команды `INSERT` или `UPDATE`, в перезаписанном запросе все параметры проверки будут игнорироваться, в том числе проверки автоматически изменяемых представлений, определённых поверх отношений с правилом `INSTEAD`. Команда `MERGE` не поддерживается, если для представления или любого из его базовых отношений определены правила.

Примечания

Для удаления представлений применяется оператор `DROP VIEW`.

Позаботьтесь о том, чтобы столбцы представления получили желаемые имена и типы. Например, такая команда:

```
CREATE VIEW vista AS SELECT 'Hello World';
```

плоха тем, что именем столбца по умолчанию будет `?column?`, а типом данных — `text`; и это может быть не совсем то, чего вы хотите. Лучше записывать строковую константу в результате представления примерно так:

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

По умолчанию доступ к нижележащим базовым отношениям, на которые ссылается представление, определяется правами владельца представления. В некоторых случаях это позволяет организовать безопасный, но ограниченный доступ к нижележащим таблицам. Однако учтите, что не все представления могут быть защищёнными; за подробностями обратитесь к Разделу 39.5.

Если для представления свойство `security_invoker` имеет значение `true`, доступ к нижележащим базовым отношениям определяется правами пользователя, выполняющего запрос, а не владельца представления. Таким образом, пользователь такого представления должен иметь соответствующие разрешения для доступа к представлению и его базовым отношениям.

Если какое-либо из нижележащих базовых отношений является представлением с контекстом безопасности вызывающего, оно будет обрабатываться так, как если бы в исходном запросе обращались непосредственно к нему. Таким образом, для представления с характеристикой `security_invoker` права доступа к нижележащим базовым отношениям должен иметь текущий пользователь, даже если на верхнем уровне он обращался к представлению без такой характеристики.

Если для какого-либо нижележащего базового отношения включена защита на уровне строк, по умолчанию действуют политики защиты для владельца представления, и доступ к любым дополнительным отношениям, на которые ссылаются эти политики, определяются правами этого пользователя. Однако если свойство представления `security_invoker` имеет значение `true`, вместо этого действуют политики для вызывающего пользователя и проверяются его права, как если бы запрос, обращающийся к такому представлению, обращался непосредственно к его базовым отношениям.

Функции, вызываемые в представлении, обрабатываются так же, как если бы они вызывались непосредственно из запроса, обращающегося к представлению. Следовательно, пользователь представления должен иметь разрешения на вызов всех функций, используемых в представлении. Функции в представлении выполняются с правами пользователя, выполняющего запрос, или владельца функции, в зависимости от того, определены ли функции как `SECURITY INVOKER` или `SECURITY DEFINER`. Так, например, функция `CURRENT_USER`, вызванная непосредственно из представления, всегда будет выдавать имя вызывающего пользователя, а не владельца представления. На это не влияет характеристика представления `security_invoker`, поэтому обращение к представлению со свойством `security_invoker`, равным `false`, *не* равнозначно вызову функции с характеристикой `SECURITY DEFINER`, и эти концепции следует различать.

Пользователь, создающий или заменяющий представление, должен иметь права `USAGE` для всех схем, фигурирующих в запросе представления, чтобы найти используемые в этом запросе объекты в этих схемах. Однако заметьте, что поиск объектов производится только при создании или замене представления. Поэтому пользователям представления требуется только право `USAGE` для схемы, содержащей представление, а не для всех схем, используемых в запросе представления, даже если это представление с контекстом безопасности вызывающего.

При выполнении `CREATE OR REPLACE VIEW` для существующего представления меняется только правило SELECT, определяющее представление, и параметры `WITH ( ... )`, а также `CHECK OPTION`. Другие свойства представления, включая владельца, права и правила, кроме SELECT, остаются неизменными. Чтобы изменить определение представления, необходимо быть его владельцем (или членом роли-владельца).

Изменяемые представления

Простые представления становятся изменяемыми автоматически: система позволит выполнять команды `INSERT`, `UPDATE`, `DELETE` и `MERGE` с таким представлением так же, как и с обычной таблицей. Представление будет автоматически изменяемым, если оно удовлетворяют одновременно всем следующим условиям:

- Список `FROM` в запросе, определяющем представлении, должен содержать ровно один элемент, и это должна быть таблица или другое изменяемое представление.
- Определение представления не должно содержать предложения `WITH`, `DISTINCT`, `GROUP BY`, `HAVING`, `LIMIT` и `OFFSET` на верхнем уровне запроса.
- Определение представления не должно содержать операции с множествами (`UNION`, `INTERSECT` и `EXCEPT`) на верхнем уровне запроса.
- Список выборки в запросе не должен содержать агрегатные и оконные функции, а также функции, возвращающие множества.

Автоматически обновляемое представление может содержать как изменяемые, так и не изменяемые столбцы. Столбец будет изменяемым, если это простая ссылка на изменяемый столбец нижележащего базового отношения; в противном случае этот столбец будет доступен только для чтения, и если команда `INSERT`, `UPDATE` или `MERGE` попытается записать значение в него, возникнет ошибка.

Если представление автоматически изменяемое, система будет преобразовывать обращающиеся к нему операторы `INSERT`, `UPDATE`, `DELETE` и `MERGE` в соответствующие операторы, обращающиеся к нижележащему базовому отношению. При этом в полной мере поддерживаются операторы `INSERT` с предложением `ON CONFLICT UPDATE`.

Если автоматически изменяемое представление содержит условие `WHERE`, это условие ограничивает набор строк, которые могут быть изменены командой `UPDATE`, `DELETE` или `MERGE` в этом представлении. Однако `UPDATE` или `MERGE` могут изменить строку так, что она больше не будет соответствовать условию `WHERE` и, как следствие, больше не будет видна через представление. Команды `INSERT` и `MERGE` подобным образом могут вставить в базовое отношение строки, которые не удовлетворят условию `WHERE` и поэтому не будут видны через представление (`ON CONFLICT UPDATE` может подобным образом воздействовать на существующую строку, не видимую через представление). Чтобы запретить командам `INSERT`, `UPDATE` и `MERGE` создавать такие строки, которые не видны через представление, можно воспользоваться указанием `CHECK OPTION`.

Если автоматически изменяемое представление имеет свойство `security_barrier` (барьер безопасности), то все условия `WHERE` этого представления (и все условия с герметичными операторами (`LEAKPROOF`)) будут всегда вычисляться перед условиями, добавленными пользователем представления. За подробностями обратитесь к Разделу 39.5. Заметьте, что по этой причине строки, которые в конце концов не были выданы (потому что не прошли проверку в пользовательском условии `WHERE`), могут всё же остаться заблокированными. Чтобы определить, какие условия применяются на уровне отношения (и, как следствие, избавляют часть строк от блокировки), можно воспользоваться командой `EXPLAIN`.

Более сложные представления, не удовлетворяющие этим условиям, по умолчанию доступны только для чтения: система не позволит выполнить операции `INSERT`, `UPDATE`, `DELETE` или `MERGE` в таком представлении. Создать эффект изменяемого представления для них можно, определив триггеры `INSTEAD OF`, которые будут преобразовывать запросы на изменение данных в соответствующие действия с другими таблицами. За дополнительными сведениями обратитесь к CREATE TRIGGER. Так же есть возможность создавать правила (см. CREATE RULE), но на практике триггеры проще для понимания и применения. Кроме того, команда `MERGE` не поддерживается для отношений с правилами.

Учтите, что пользователь, выполняющий операции добавления, изменения или удаления данных в представлении, должен иметь соответствующие права для этого представления. Кроме того, владелец представления должен иметь сопутствующие права в нижележащих базовых отношениях, хотя пользователь, собственно выполняющий эти операции, может этих прав не иметь (см. Раздел 39.5). Однако если для представления свойство `security_invoker` равно `true`, пользователь, выполняющий изменение, а не владелец представления должен иметь соответствующие права для нижележащих базовых отношений.

Примеры

Создание представления, содержащего все комедийные фильмы:

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

Эта команда создаст представление со столбцами, которые содержались в таблице `film` в момент выполнения команды. Хотя при создании представления было указано `*`, столбцы, добавляемые в таблицу позже, частью представления не будут.

Создание представления с указанием `LOCAL CHECK OPTION`:

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

Эта команда создаст представление на базе представления `comedies`, выдающее только комедии (`kind = 'Comedy'`) универсальной возрастной категории `classification = 'U'`. Любая попытка выполнить в представлении `INSERT` или `UPDATE` со строкой, не удовлетворяющей условию `classification = 'U'`, будет отвергнута, но ограничение по полю `kind` (тип фильма) проверяться не будет.

Создание представления с указанием `CASCADED CHECK OPTION`:

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

Это представление будет проверять, удовлетворяют ли новые строки обоим условиям: по столбцу `kind` и по столбцу `classification`.

Создание представления с изменяемыми и неизменяемыми столбцами:

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

Это представление будет поддерживать операции `INSERT`, `UPDATE` и `DELETE`. Изменяемыми будут все столбцы из таблицы `films`, тогда как вычисляемые столбцы `country` и `avg_rating` будут доступны только для чтения.

Создание рекурсивного представления, содержащего числа от 1 до 100:

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

Заметьте, что несмотря на то, что имя рекурсивного представления дополнено схемой в этой команде `CREATE`, внутренняя ссылка представления на себя же схемой не дополняется. Это связано с тем, что имя неявно создаваемого CTE не может дополняться схемой.

Совместимость

Команда `CREATE OR REPLACE VIEW` — языковое расширение Postgres Pro. Так же расширениями являются концепция временного представления, предложение `WITH ( ... )`, представления с барьером безопасности и представления с контекстом безопасности вызывающего.

См. также

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

DEALLOCATE

DEALLOCATE — освободить подготовленный оператор

Синтаксис

```
DEALLOCATE [ PREPARE ] { имя | ALL }
```

Описание

`DEALLOCATE` применяется для освобождения ранее подготовленного оператора SQL. Если не освободить подготовленный оператор явно, он будет освобождён при завершении сеанса.

За дополнительными сведениями о подготовленных операторах обратитесь к PREPARE.

Параметры

`PREPARE`

Это ключевое слово игнорируется.

`имя`

Имя подготовленного оператора, подлежащего освобождению.

`ALL`

Освобождает все подготовленные операторы.

Совместимость

В стандарте SQL есть оператор `DEALLOCATE`, но он предназначен только для применения во встраиваемом SQL.

См. также

EXECUTE, PREPARE

DECLARE

DECLARE — определить курсор

Синтаксис

```
DECLARE имя [ BINARY ] [ ASENSITIVE | INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

Описание

Оператор `DECLARE` позволяет пользователю создавать курсоры, с помощью которых можно выбирать по очереди некоторое количество строк из результата большого запроса. Когда курсор создан, через него можно получать строки, применяя команду `FETCH`.

Примечание

На этой странице описывается применение курсоров на уровне команд SQL. Если вы попытаетесь использовать курсоры внутри функции PL/pgSQL, правила будут другими — см. Раздел 41.7.

Параметры

`имя`

Имя создаваемого курсора. Оно должно отличаться от имён других активных курсоров в сеансе.

`BINARY`

Курсор с таким свойством возвращает данные в двоичном, а не текстовом формате.

`ASENSITIVE`
`INSENSITIVE`

Свойство «чувствительности» курсора определяет, будут ли изменения нижележащих данных курсора, выполненные в той же транзакции после объявления курсора, видны в нём. `INSENSITIVE` означает, что они не будут видны, а `ASENSITIVE` выбирает поведение, определяемое реализацией. Третий вариант, `SENSITIVE`, при выборе которого такие изменения должны быть видны в курсоре, в Postgres Pro не поддерживается. В Postgres Pro все курсоры являются «нечувствительными», поэтому данные ключевые слова не действуют и принимаются только для совместимости со стандартом SQL.

Указание `INSENSITIVE` вместе с `FOR UPDATE` или `FOR SHARE` считается ошибочным.

`SCROLL`
`NO SCROLL`

Указание `SCROLL` определяет, что курсор может прокручивать набор данных и получать строки непоследовательно (например, в обратном порядке). В зависимости от сложности плана запроса указание `SCROLL` может отрицательно отразиться на скорости выполнения запроса. Указание `NO SCROLL`, напротив, определяет, что через курсор нельзя будет получать строки в произвольном порядке. По умолчанию прокрутка в некоторых случаях разрешается; но это не равнозначно эффекту указания `SCROLL`. За подробностями обратитесь к разделу Замечания.

`WITH HOLD`
`WITHOUT HOLD`

Указание `WITH HOLD` определяет, что курсор можно продолжать использовать после успешной фиксации создавшей его транзакции. `WITHOUT HOLD` определяет, что курсор нельзя будет использовать за рамками транзакции, создавшей его. Если не указано ни `WITHOUT HOLD`, ни `WITH HOLD`, по умолчанию подразумевается `WITHOUT HOLD`.

query

Команда `SELECT` или `VALUES`, выдающая строки, которые будут получены через курсор.

Ключевые слова `ASENSITIVE`, `BINARY`, `INSENSITIVE` и `SCROLL` могут указываться в любом порядке.

Замечания

Обычный курсор выдаёт данные в текстовом виде, в каком их выдаёт `SELECT`. Однако с указанием `BINARY` курсор может выдавать их и в двоичном формате. Это упрощает операции преобразования данных для сервера и клиента, за счёт дополнительных усилий, требующихся от программиста для работы с платформозависимыми двоичными форматами. Например, если запрос получает значение 1 из целочисленного столбца, обычный курсор выдаст строку, содержащую `1`, тогда как через двоичный курсор будет получено четырёхбайтовое поле, содержащее внутреннее представление значения (с сетевым порядком байтов).

Двоичные курсоры должны применяться с осмотрительностью. Многие приложения, в том числе psql, не приспособлены к работе с двоичными курсорами и ожидают, что данные будут поступать в текстовом формате.

Примечание

Когда клиентское приложение выполняет команду `FETCH`, используя «расширенный» протокол запросов, в сообщении Bind этого протокола указывается, в каком формате, текстовом или двоичном, должны быть получены данные. Это указание переопределяет свойство курсора, заданное в его объявлении. Таким образом, концепция курсора, объявляемого двоичным, становится устаревшей при использовании расширенного протокола запросов — любой курсор может быть прочитан как текстовый или двоичный.

Если в команде объявления курсора не указано `WITH HOLD`, созданный ей курсор может использоваться только в текущей транзакции. Таким образом, оператор `DECLARE` без `WITH HOLD` бесполезен вне блока транзакции: курсор будет существовать только до завершения этого оператора. Поэтому Postgres Pro сообщает об ошибке, если такая команда выполняется вне блока транзакции. Чтобы определить блок транзакции, примените команды `BEGIN` и `COMMIT` (или `ROLLBACK`).

Если в объявлении курсора указано `WITH HOLD` и транзакция, создавшая курсор, успешно фиксируется, к этому курсору могут продолжать обращаться последующие транзакции в этом сеансе. (Но если создавшая курсор транзакция прерывается, курсор уничтожается.) Курсор со свойством `WITH HOLD` (удерживаемый) может быть закрыт явно, командой `CLOSE`, либо неявно, по завершении сеанса. В текущей реализации строки, представляемые удерживаемым курсором, копируются во временный файл или в область памяти, так что они остаются доступными для следующих транзакций.

Объявить курсор со свойством `WITH HOLD` можно, только если запрос не содержит указаний `FOR UPDATE` и `FOR SHARE`.

Указание `SCROLL` добавляется при определении курсора, который будет выбирать данные в обратном порядке. Это поведение требуется стандартом SQL. Однако для совместимости с предыдущими версиями, Postgres Pro допускает выборку в обратном направлении и без указания `SCROLL`, если план запроса курсора достаточно прост, чтобы реализовать прокрутку назад без дополнительных операций. Тем не менее разработчикам приложений не следует рассчитывать на то, что курсор, созданный без указания `SCROLL`, можно будет прокручивать назад. С указанием `NO SCROLL` прокрутка назад запрещается в любом случае.

Выборка в обратном направлении также запрещается, если запрос содержит указания `FOR UPDATE` и `FOR SHARE`; в этом случае указание `SCROLL` не принимается.

Внимание

Прокручиваемые курсоры могут выдавать неожиданные результаты, если они вызывают изменчивые функции (см. Раздел 36.7). Когда повторно выбирается ранее прочитанная строка, функции могут вызываться снова и выдавать результаты, отличные от полученных в первый раз. Для запроса, вызывающего изменчивые функции, лучше всего указать `NO SCROLL`. Если такой способ не подходит, другая возможность обойти эту проблему — объявить курсор с указанием `WITH HOLD` и зафиксировать транзакцию, прежде чем читать из него какие-либо строки. В этом случае весь набор данных курсора будет материализован во временном хранилище, так что изменчивые функции будут выполнены для каждой строки лишь единожды.

Если запрос в определении курсора включает указания `FOR UPDATE` или `FOR SHARE`, возвращаемые курсором строки блокируются в момент первой выборки, так же, как это происходит при выполнении `SELECT` с этими указаниями. Кроме того, при чтении строк будут возвращаться их наиболее актуальные версии.

Внимание

Обычно рекомендуется использовать `FOR UPDATE`, если курсор предназначается для применения в командах `UPDATE ... WHERE CURRENT OF` и `DELETE ... WHERE CURRENT OF`. Указание `FOR UPDATE` предотвращает изменение строк другими сеансами после того, как они были считаны, и до того, как выполнится команда. Без `FOR UPDATE` последующая команда с `WHERE CURRENT OF` не сработает, если строка будет изменена после создания курсора.

Ещё одна причина использовать указание `FOR UPDATE` в том, что без него последующие команды с `WHERE CURRENT OF` могут выдать ошибку, если запрос курсора не удовлетворяет оговоренному в стандарте SQL критерию «простой изменяемости» (в частности, курсор должен ссылаться только на одну таблицу и не должен использовать группировку и сортировку (`ORDER BY`)). Курсоры, не удовлетворяющие этому критерию, могут работать либо не работать, в зависимости от конкретного выбранного плана; так что в худшем случае приложение может работать в тестовой, но сломается в производственной среде. С указанием `FOR UPDATE` курсор гарантированно будет изменяемым.

Не использовать же `FOR UPDATE` для команд с `WHERE CURRENT OF` в основном имеет смысл, только если требуется получить прокручиваемый курсор или курсор, изолированный от одновременно производимых изменений (то есть, продолжающий показывать прежние данные). Если это действительно необходимо, обязательно учтите при реализации приведённые выше замечания.

В стандарте SQL механизм курсоров предусмотрен только для встраиваемого SQL. Сервер Postgres Pro не реализует для курсоров оператор `OPEN`; курсор считается открытым при объявлении. Однако ECPG, встраиваемый препроцессор SQL для Postgres Pro, следует соглашениям стандарта, в том числе поддерживая для курсоров операторы `DECLARE` и `OPEN`.

Структура данных сервера, определяющая открытый курсор, называется *порталом*. Имена порталов раскрываются в клиентском протоколе, то есть если имя портала известно, клиент может выбирать строки напрямую через открытый портал. Если курсор создаётся командой `DECLARE`, имя портала совпадает с именем курсора.

Получить список всех доступных курсоров можно, обратившись к системному представлению `pg_cursors`.

Примеры

Объявление курсора:

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

Другие примеры использования курсора можно найти в FETCH.

Совместимость

Стандарт SQL допускает курсоры только во встраиваемом SQL и в модулях. Postgres Pro позволяет использовать курсоры интерактивно.

Согласно стандарту SQL, изменения, которые вносит нечувствительный курсор, выполняющий оператор `UPDATE ... WHERE CURRENT OF` и `DELETE ... WHERE CURRENT OF`, должны быть видны в этом же курсоре. Postgres Pro обрабатывает эти операторы как и любые другие операторы, изменяющие данные, так что нечувствительные курсоры не видят результатов их действия.

Двоичные курсоры являются расширением Postgres Pro.

См. также

CLOSE, FETCH, MOVE

DELETE

DELETE — удалить записи таблицы

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
DELETE FROM [ ONLY ] имя_таблицы [ * ] [ [ AS ] псевдоним ]
    [ USING элемент_FROM [, ...] ]
    [ WHERE условие | WHERE CURRENT OF имя_курсора ]
    [ RETURNING { * | выражение_результата [ [ AS ] имя_результата ] } [, ...] ]
```

Описание

Команда `DELETE` удаляет из указанной таблицы строки, удовлетворяющие условию `WHERE`. Если предложение `WHERE` отсутствует, она удаляет из таблицы все строки, в результате будет получена рабочая, но пустая таблица.

> ### Подсказка
>
> `TRUNCATE` реализует более быстрый механизм удаления всех строк из таблицы.

Удалить строки в таблице, используя информацию из других таблиц в базе данных, можно двумя способами: применяя вложенные запросы или указав дополнительные таблицы в предложении `USING`. Выбор предпочитаемого варианта зависит от конкретных обстоятельств.

Предложение `RETURNING` указывает, что команда `DELETE` должна вычислить и возвратить значения для каждой фактически удалённой строки. Вычислить в нём можно любое выражение со столбцами целевой таблицы и/или столбцами других таблиц, упомянутых в `USING`. Список `RETURNING` имеет тот же синтаксис, что и список результатов `SELECT`.

Чтобы удалять данные из таблицы, необходимо иметь право `DELETE` для неё, а также право `SELECT` для всех таблиц, перечисленных в предложении `USING`, и таблиц, данные которых считываются в *`условии`*.

Параметры

`запрос_WITH`

Предложение `WITH` позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе `DELETE`. Подробнее об этом см. Раздел 7.8 и SELECT.

`имя_таблицы`

Имя (возможно, дополненное схемой) таблицы, из которой будут удалены строки. Если перед именем таблицы добавлено `ONLY`, соответствующие строки удаляются только из указанной таблицы. Без `ONLY` строки будут также удалены из всех таблиц, унаследованных от указанной. При желании, после имени таблицы можно указать `*`, чтобы явно обозначить, что операция затрагивает все дочерние таблицы.

`псевдоним`

Альтернативное имя целевой таблицы. Когда указывается это имя, оно полностью скрывает фактическое имя таблицы. Например, в запросе `DELETE FROM foo AS f` дополнительные компоненты оператора `DELETE` должны обращаться к целевой таблице по имени `f`, а не `foo`.

`элемент_FROM`

Табличное выражение, позволяющее добавить в условие `WHERE` столбцы из других таблиц. В этом выражении используется тот же синтаксис, что и в предложении `FROM` оператора `SELECT`;

например, в нём можно определить псевдоним для таблицы. Повторять в нём имя целевой таблицы нужно, только если требуется определить замкнутое соединение (в этом случае для данного имени должен определяться псевдоним).

условие

Выражение, возвращающее значение типа `boolean`. Удалены будут только те строки, для которых это выражение возвращает `true`.

имя_курсора

Имя курсора, который будет использоваться в условии `WHERE CURRENT OF`. С таким условием будет удалена строка, выбранная из этого курсора последней. Курсор должен образовываться запросом, не применяющим группировку, к целевой таблице команды `DELETE`. Заметьте, что `WHERE CURRENT OF` нельзя задать вместе с логическим условием. За дополнительными сведениями об использовании курсоров с `WHERE CURRENT OF` обратитесь к DECLARE.

выражение_результата

Выражение, которое будет вычисляться и возвращаться командой `DELETE` после удаления каждой строки. В этом выражении можно использовать имена любых столбцов таблицы *имя_таблицы* или таблиц, перечисленных в списке `USING`. Чтобы получить все столбцы, достаточно написать `*`.

имя_результата

Имя, назначаемое возвращаемому столбцу.

Выводимая информация

В случае успешного завершения, `DELETE` возвращает метку команды в виде

`DELETE` *`число`*

Здесь *число* — количество удалённых строк. Заметьте, что это число может быть меньше числа строк, соответствующих *условию*, если удаления были подавлены триггером `BEFORE DELETE`. Если *число* равно 0, это означает, что запрос не удалил ни одной строки (это не считается ошибкой).

Если команда `DELETE` содержит предложение `RETURNING`, её результат будет похож на результат оператора `SELECT` (с теми же столбцами и значениями, что содержатся в списке `RETURNING`), полученный для строк, удалённых этой командой.

Примечания

Postgres Pro позволяет ссылаться на столбцы других таблиц в условии `WHERE`, когда эти таблицы перечисляются в предложении `USING`. Например, удалить все фильмы определённого продюсера можно так:

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

По сути в этом запросе выполняется соединение таблиц `films` и `producers`, и все успешно включённые в соединение строки в `films` помечаются для удаления. Этот синтаксис не соответствует стандарту. Следуя стандарту, эту задачу можно решить так:

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

В ряде случаев запрос в стиле соединения легче написать и он может работать быстрее, чем в стиле вложенного запроса.

Примеры

Удаление всех фильмов, кроме мюзиклов:

```
DELETE FROM films WHERE kind <> 'Musical';
```

Очистка таблицы `films`:

```
DELETE FROM films;
```

Удаление завершённых задач с получением всех данных удалённых строк:

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

Удаление из `tasks` строки, на которой в текущий момент располагается курсор `c_tasks`:

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

Несмотря на отсутствие предложения `LIMIT` для команды `DELETE`, можно ограничить число удаляемых строк, использовав метод в описании команды `UPDATE`:

```
WITH delete_batch AS (
  SELECT l.ctid FROM user_logs AS l
    WHERE l.status = 'archived'
    ORDER BY l.creation_date
    FOR UPDATE
    LIMIT 10000
)
DELETE FROM user_logs AS dl
  USING delete_batch AS del
  WHERE dl.ctid = del.ctid;
```

Совместимость

Эта команда соответствует стандарту SQL, но предложения `USING` и `RETURNING` являются расширениями Postgres Pro, как и возможность использовать `WITH` с `DELETE`.

См. также

TRUNCATE

DISCARD

DISCARD — очистить состояние сеанса

Синтаксис

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

Описание

`DISCARD` высвобождает внутренние ресурсы, связанные с сеансом использования базы данных. Эта команда полезна для частичного или полного сброса состояния сеанса. Для освобождения различных типов ресурсов она поддерживает несколько разных подкоманд; вариант `DISCARD ALL` включает в себя все остальные и также сбрасывает дополнительное состояние.

Параметры

`PLANS`

Высвобождает все кешированные планы запросов, вынуждая сервер провести планирование заново при следующем использовании связанного подготовленного оператора.

`SEQUENCES`

Сбрасывает кешированное состояние, связанное с последовательностями, включая внутреннюю информацию `currval()`/`lastval()` и любые предварительно выделенные значения последовательностей, которые ещё не выдала функция `nextval()`. (Кеширование значений последовательности описано в CREATE SEQUENCE.)

`TEMPORARY` или `TEMP`

Удаляет все временные таблицы, созданные в текущем сеансе.

`ALL`

Высвобождает все временные ресурсы, связанные с текущим сеансом, и сбрасывает сеанс к начальному состоянию. В настоящее время действует так же, как и следующая последовательность операторов:

```
CLOSE ALL;
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
DISCARD SEQUENCES;
```

Примечания

`DISCARD ALL` нельзя выполнять внутри блока транзакции.

Совместимость

`DISCARD` является расширением Postgres Pro.

DO

DO — выполнить анонимный блок кода

Синтаксис

```
DO [ LANGUAGE имя_языка ] код
```

Описание

`DO` выполняет анонимный блок кода или, другими словами, разовую анонимную функцию на процедурном языке.

Блок кода воспринимается, как если бы это было тело функции, которая не имеет параметров и возвращает `void`. Этот код разбирается и выполняется один раз.

Необязательное предложение `LANGUAGE` можно записать до или после блока кода.

Параметры

код

Выполняемый код на процедурном языке. Он должен задаваться в виде текстовой строки (её рекомендуется заключать в доллары), как и код в `CREATE FUNCTION`.

имя_языка

Имя процедурного языка, на котором написан код. По умолчанию подразумевается `plpgsql`.

Примечания

Применяемый процедурный язык должен быть уже зарегистрирован в текущей базе с помощью команды `CREATE EXTENSION`. По умолчанию зарегистрирован только `plpgsql`, но не другие языки.

Пользователь должен иметь право `USAGE` для данного процедурного языка, либо быть суперпользователем, если этот язык недоверенный. Такие же требования действуют и при создании функции на этом языке.

Если команда `DO` исполняется в блоке транзакции, код процедуры не может вызывать операторы управления транзакциями. Такие операторы допускаются, только если `DO` выполняется в отдельной транзакции.

Примеры

Следующий код даст все права для всех представлений в схеме `public` роли `webuser`:

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' ||
 quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;
```

Совместимость

Оператор `DO` отсутствует в стандарте SQL.

См. также

CREATE LANGUAGE

DROP ACCESS METHOD

DROP ACCESS METHOD — удалить метод доступа

Синтаксис

```
DROP ACCESS METHOD [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP ACCESS METHOD` удаляет существующий метод доступа. Удалять методы доступа разрешено только суперпользователям.

Параметры

`IF EXISTS`

Не считать ошибкой, если метод доступа не существует. В этом случае будет выдано замечание.

имя

Имя существующего метода доступа.

`CASCADE`

Автоматически удалять объекты, зависящие от данного метода доступа (например, классы и семейства операторов, а также индексы), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении метода доступа, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление метода доступа `heptree`:

```
DROP ACCESS METHOD heptree;
```

Совместимость

`DROP ACCESS METHOD` является расширением Postgres Pro.

См. также

CREATE ACCESS METHOD

DROP AGGREGATE

DROP AGGREGATE — удалить агрегатную функцию

Синтаксис

```
DROP AGGREGATE [ IF EXISTS ] имя ( сигнатура_агр_функции ) [, ...] [ CASCADE |
 RESTRICT ]

Здесь сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`DROP AGGREGATE` удаляет существующую агрегатную функцию. Пользователь, выполняющий эту команду, должен быть владельцем агрегатной функции.

Параметры

`IF EXISTS`

Не считать ошибкой, если агрегатная функция не существует. В этом случае будет выдано замечание.

имя

Имя существующей агрегатной функции (возможно, дополненное схемой).

режим_аргумента

Режим аргумента: `IN` или `VARIADIC`. По умолчанию подразумевается `IN`.

имя_аргумента

Имя аргумента. Заметьте, что на самом деле `DROP AGGREGATE` не обращает внимание на имена аргументов, так как для однозначной идентификации агрегатной функции достаточно только типов аргументов.

тип_аргумента

Тип входных данных, с которыми работает агрегатная функция. Чтобы сослаться на агрегатную функцию без аргументов, укажите вместо списка аргументов `*`, а чтобы сослаться на сортирующую агрегатную функцию, добавьте `ORDER BY` между указаниями непосредственных и агрегируемых аргументов.

`CASCADE`

Автоматически удалять объекты, зависящие от данной агрегатной функции (например, использующие её представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении агрегатной функции, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примечания

Альтернативные варианты указания сортирующих агрегатов описаны в ALTER AGGREGATE.

Примеры

Удаление агрегатной функции `myavg` для типа `integer`:

```
DROP AGGREGATE myavg(integer);
```

Удаление гипотезирующей агрегатной функции `myrank`, которая принимает произвольный список сортируемых столбцов и соответствующий список непосредственных аргументов:

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

Удаление нескольких агрегатных функций одной командой:

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

Совместимость

Оператор `DROP AGGREGATE` отсутствует в стандарте SQL.

См. также

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

DROP CAST — удалить приведение типа

Синтаксис

```
DROP CAST [ IF EXISTS ] (исходный_тип AS целевой_тип) [ CASCADE | RESTRICT ]
```

Описание

`DROP CAST` удаляет ранее определённое приведение типа.

Чтобы удалить приведение, необходимо быть владельцем исходного или целевого типа данных. Такие же требования действуют и при создании приведения.

Параметры

`IF EXISTS`

Не считать ошибкой, если приведение не существует. В этом случае будет выдано замечание.

`исходный_тип`

Имя исходного типа данных для приведения.

`целевой_тип`

Имя целевого типа данных для приведения.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от приведений не зависят никакие объекты.

Примеры

Удаление приведения типа `text` к типу `int`:

```
DROP CAST (text AS int);
```

Совместимость

Команда `DROP CAST` соответствует стандарту SQL.

См. также

CREATE CAST

DROP COLLATION

DROP COLLATION — удалить правило сортировки

Синтаксис

```
DROP COLLATION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP COLLATION` удаляет ранее определённое правило сортировки. Чтобы удалить правило сортировки, необходимо быть его владельцем.

Параметры

`IF EXISTS`

Не считать ошибкой, если правило сортировки не существует. В этом случае будет выдано замечание.

`имя`

Имя правила сортировки, возможно, дополненное схемой.

`CASCADE`

Автоматически удалять объекты, зависящие от данного правила сортировки, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении правила сортировки, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление правила сортировки с именем `german`:

```
DROP COLLATION german;
```

Совместимость

Команда `DROP COLLATION` соответствует стандарту SQL, за исключением параметра `IF EXISTS`, являющегося расширением Postgres Pro.

См. также

ALTER COLLATION, CREATE COLLATION

DROP CONVERSION

DROP CONVERSION — удалить преобразование

Синтаксис

```
DROP CONVERSION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP CONVERSION` удаляет ранее определённое преобразование. Чтобы удалить преобразование, необходимые быть его владельцем.

Параметры

`IF EXISTS`

Не считать ошибкой, если преобразование не существует. В этом случае будет выдано замечание.

`имя`

Имя преобразования, возможно, дополненное схемой.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от преобразований не зависят никакие объекты.

Примеры

Удаление преобразования с именем `myname`:

```
DROP CONVERSION myname;
```

Совместимость

Оператор `DROP CONVERSION` отсутствует в стандарте SQL, хотя в нём есть оператор `DROP TRANSLATION`, сопутствующий оператору `CREATE TRANSLATION`, который, в свою очередь, подобен `CREATE CONVERSION` в Postgres Pro.

См. также

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

DROP DATABASE — удалить базу данных

Синтаксис

```
DROP DATABASE [ IF EXISTS ] имя [ [ WITH ] ( параметр [, ...] ) ]

Здесь допускается параметр:

    FORCE
```

Описание

Команда `DROP DATABASE` удаляет базу данных. Она удаляет из системного каталога записи, относящиеся к базе, а также удаляет с диска каталог, содержащий данные. Выполнить её может только владелец базы данных. Кроме того, нельзя удалить базу, к которой вы подключены в данный момент. (Чтобы выполнить эту команду, подключитесь к `postgres` или любой другой базе данных.) Также команда не будет выполнена, когда к целевой базе подключены какие-то ещё пользователи, если вы не добавите указание `FORCE`, описанное ниже.

Действие команды `DROP DATABASE` нельзя отменить. Используйте её с осторожностью!

Параметры

`IF EXISTS`

Не считать ошибкой, если база данных не существует. В этом случае будет выдано замечание.

`имя`

Имя базы данных, подлежащей удалению.

`FORCE`

Попытаться принудительно завершить все существующие подключения к целевой базе данных. Подключения не завершаются, если в целевой базе имеются подготовленные транзакции, активные слоты логической репликации или подписки.

Завершает подключения фонового рабочего процесса и подключения, которые разрешено завершить текущему пользователю с помощью `pg_terminate_backend`, описанной в Подразделе 9.28.2. Если завершатся не все подключения, эта команда выдаст ошибку.

Примечания

`DROP DATABASE` нельзя выполнять внутри блока транзакции.

Эту команду нельзя выполнить, если установлено подключение к удаляемой базе данных. Поэтому может быть удобнее вместо неё использовать программу dropdb, которая сама вызывает эту команду внутри.

Совместимость

Оператор `DROP DATABASE` отсутствует в стандарте SQL.

См. также

CREATE DATABASE

DROP DOMAIN

DROP DOMAIN — удалить домен

Синтаксис

```
DROP DOMAIN [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP DOMAIN` удаляет домен. Удалить домен может только его владелец.

Параметры

`IF EXISTS`

Не считать ошибкой, если домен не существует. В этом случае будет выдано замечание.

имя

Имя существующего домена (возможно, дополненное схемой).

`CASCADE`

Автоматически удалять объекты, зависящие от данного домена (например, столбцы таблиц), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении домена, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление домена `box`:

```
DROP DOMAIN box;
```

Совместимость

Эта команда соответствует стандарту SQL, за исключением параметра `IF EXISTS`, являющегося расширением Postgres Pro.

См. также

CREATE DOMAIN, ALTER DOMAIN

DROP EVENT TRIGGER

DROP EVENT TRIGGER — удалить событийный триггер

Синтаксис

```
DROP EVENT TRIGGER [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP EVENT TRIGGER` удаляет существующий событийный триггер. Пользователь, выполняющий эту команду, должен быть владельцем событийного триггера.

Параметры

`IF EXISTS`

Не считать ошибкой, если событийный триггер не существует. В этом случае будет выдано замечание.

`имя`

Имя событийного триггера, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного триггера, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении триггера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление триггера `snitch`:

```
DROP EVENT TRIGGER snitch;
```

Совместимость

Оператор `DROP EVENT TRIGGER` отсутствует в стандарте SQL.

См. также

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

DROP EXTENSION

DROP EXTENSION — удалить расширение

Синтаксис

```
DROP EXTENSION [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP EXTENSION` удаляет расширения из базы данных. При удалении расширения также удаляются все составляющие его объекты и другие явно зависимые подпрограммы (см. ALTER ROUTINE, указание `DEPENDS ON EXTENSION` *`имя_расширения`*).

Чтобы выполнить `DROP EXTENSION`, необходимо быть владельцем расширения.

Параметры

`IF EXISTS`

Не считать ошибкой, если расширение не существует. В этом случае будет выдано замечание.

`имя`

Имя установленного расширения.

`CASCADE`

Автоматически удалять объекты, зависящие от данного расширения, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Этот параметр предотвращает удаление указанных расширений, если существуют зависимые от них объекты, за исключением самих расширений, их внутренних объектов и явно зависящих от них подпрограмм. Это поведение по умолчанию.

Примеры

Удаление расширения `hstore` из текущей базы данных.

```
DROP EXTENSION hstore;
```

Эта команда не будет выполнена, если какие-либо объекты из `hstore` будут задействованы, например, если в какой-либо таблице окажется столбец типа `hstore`. Чтобы принудительно удалить и эти зависимые объекты, необходимо добавить параметр `CASCADE`.

Совместимость

`DROP EXTENSION` является расширением Postgres Pro.

См. также

CREATE EXTENSION, ALTER EXTENSION

DROP FOREIGN DATA WRAPPER

DROP FOREIGN DATA WRAPPER — удалить обёртку сторонних данных

Синтаксис

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP FOREIGN DATA WRAPPER` удаляет существующую обёртку сторонних данных. Пользователь, выполняющий эту команду, должен быть владельцем обёртки.

Параметры

`IF EXISTS`

Не считать ошибкой, если обёртка сторонних данных не существует. В этом случае будет выдано замечание.

`имя`

Имя существующей обёртки сторонних данных.

`CASCADE`

Автоматически удалять объекты, зависящие от данной обёртки сторонних данных (например, сторонние таблицы и серверы), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении обёртки сторонних данных, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление обёртки сторонних данных `dbi`:

```
DROP FOREIGN DATA WRAPPER dbi;
```

Совместимость

`DROP FOREIGN DATA WRAPPER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Предложение `IF EXISTS` является расширением Postgres Pro.

См. также

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER

DROP FOREIGN TABLE

DROP FOREIGN TABLE — удалить стороннюю таблицу

Синтаксис

```
DROP FOREIGN TABLE [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP FOREIGN TABLE` удаляет стороннюю таблицу. Удалить стороннюю таблицу может только её владелец.

Параметры

`IF EXISTS`

Не считать ошибкой, если сторонняя таблица не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) сторонней таблицы, подлежащей удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данной сторонней таблицы (например, представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении сторонней таблицы, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление двух сторонних таблиц, `films` и `distributors`:

```
DROP FOREIGN TABLE films, distributors;
```

Совместимость

Эта команда соответствует ISO/IEC 9075-9 (SQL/MED), но возможность удалять в одной команде несколько таблиц и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

DROP FUNCTION

DROP FUNCTION — удалить функцию

Синтаксис

```
DROP FUNCTION [ IF EXISTS ] имя [ ( [ [ режим_аргумента ] [ имя_аргумента
 ] тип_аргумента [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

Описание

`DROP FUNCTION` удаляет определение существующей функции. Пользователь, выполняющий эту команду, должен быть владельцем функции. Помимо имени функции требуется указать типы её аргументов, так как в базе данных могут существовать несколько функций с одним именем, но с разными списками аргументов.

Параметры

`IF EXISTS`

Не считать ошибкой, если функция не существует. В этом случае будет выдано замечание.

`имя`

Имя существующей функции (возможно, дополненное схемой). Если список аргументов не указан, имя функции должно быть уникальным в её схеме.

`режим_аргумента`

Режим аргумента: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `DROP FUNCTION` не учитывает аргументы `OUT`, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы `IN`, `INOUT` и `VARIADIC`.

`имя_аргумента`

Имя аргумента. Заметьте, что на самом деле `DROP FUNCTION` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

`тип_аргумента`

Тип данных аргументов функции (возможно, дополненный именем схемы), если таковые имеются.

`CASCADE`

Автоматически удалять объекты, зависящие от данной функции (например, операторы или триггеры), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении функции, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удаляет функцию, вычисляющую квадратный корень:

```
DROP FUNCTION sqrt(integer);
```

Удаление нескольких функций одной командой:

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

Если имя функции уникально в её схеме, на неё можно сослаться без списка аргументов:

```
DROP FUNCTION update_employee_salaries;
```

Заметьте, что это отличается от

```
DROP FUNCTION update_employee_salaries();
```

Данная форма ссылается на функцию с нулём аргументов, тогда как первый вариант подходит для функции с любым числом аргументов, в том числе и с нулём, если имя функции уникально.

Совместимость

Эта команда соответствует стандарту SQL, но дополнена следующими расширениями Postgres Pro:

- Стандарт допускает удаление в одной команде только одной функции.
- Указание `IF EXISTS`
- Возможность указывать режимы и имена аргументов

См. также

CREATE FUNCTION, ALTER FUNCTION, DROP PROCEDURE, DROP ROUTINE

DROP GROUP

DROP GROUP — удалить роль в базе данных

Синтаксис

```
DROP GROUP [ IF EXISTS ] имя [, ...]
```

Описание

`DROP GROUP` теперь является синонимом оператора `DROP ROLE`.

Совместимость

Оператор `DROP GROUP` отсутствует в стандарте SQL.

См. также

DROP ROLE

DROP INDEX

DROP INDEX — удалить индекс

Синтаксис

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP INDEX` удаляет существующий индекс из базы данных. Выполнить эту команду может только владелец индекса.

Параметры

`CONCURRENTLY`

С этим указанием индекс удаляется, не блокируя одновременные операции выборки, добавления, изменения и удаления данных в таблице индекса. Обычный оператор `DROP INDEX` запрашивает блокировку `ACCESS EXCLUSIVE` для таблицы, не допуская другие обращения к ней до завершения удаления. Если же добавлено это указание, команда, напротив, будет ждать завершения конфликтующих транзакций.

Применяя это указание, надо учитывать несколько особенностей. В частности, при этом можно задать имя только одного индекса, а параметр `CASCADE` не поддерживается. (Таким образом, индекс, поддерживающий ограничение `UNIQUE` или `PRIMARY KEY`, так удалить нельзя.) Кроме того, обычную команду `DROP INDEX` можно выполнить в блоке транзакции, а `DROP INDEX CONCURRENTLY` — нет. Наконец, с этим указанием нельзя удалить индексы секционированных таблиц.

Для временных таблиц `DROP INDEX` всегда выполняется более простым, неблокирующим способом, так как они не могут использоваться никакими другими сеансами.

`IF EXISTS`

Не считать ошибкой, если индекс не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) индекса, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного индекса, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении индекса, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удалит индекс `title_idx`:

```
DROP INDEX title_idx;
```

Совместимость

`DROP INDEX` является языковым расширением Postgres Pro. Средства обеспечения индексов в стандарте SQL не описаны.

См. также

CREATE INDEX

DROP LANGUAGE

DROP LANGUAGE — удалить процедурный язык

Синтаксис

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP LANGUAGE` удаляет определение ранее зарегистрированного процедурного языка. Выполнить `DROP LANGUAGE` может только владелец языка или суперпользователь.

> ### Примечание
>
> Начиная с PostgreSQL 9.1, большинство процедурных языков были преобразованы в «расширения», так что теперь их следует удалять командами `DROP EXTENSION`, а не `DROP LANGUAGE`.

Параметры

`IF EXISTS`

Не считать ошибкой, если язык не существует. В этом случае будет выдано замечание.

имя

Имя существующего процедурного языка.

`CASCADE`

Автоматически удалять объекты, зависящие от данного языка (например, функции на этом языке), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении языка, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удаляет процедурный язык `plsample`:

```
DROP LANGUAGE plsample;
```

Совместимость

Оператор `DROP LANGUAGE` отсутствует в стандарте SQL.

См. также

ALTER LANGUAGE, CREATE LANGUAGE

DROP MATERIALIZED VIEW

DROP MATERIALIZED VIEW — удалить материализованное представление

Синтаксис

```
DROP MATERIALIZED VIEW [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP MATERIALIZED VIEW` удаляет существующее материализованное представление. Выполнить эту команду может только владелец материализованного представления.

Параметры

`IF EXISTS`

Не считать ошибкой, если материализованное представление не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) материализованного представления, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного материализованного представления (например, другие материализованные или обычные представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении материализованного представления, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удаляет материализованное представление с именем `order_summary`:

```
DROP MATERIALIZED VIEW order_summary;
```

Совместимость

`DROP MATERIALIZED VIEW` — расширение Postgres Pro.

См. также

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

DROP OPERATOR

DROP OPERATOR — удалить оператор

Синтаксис

```
DROP OPERATOR [ IF EXISTS ] имя ( { тип_слева | NONE } , тип_справа ) [, ...] [ CASCADE
 | RESTRICT ]
```

Описание

`DROP OPERATOR` удаляет существующий оператор из базы данных. Выполнить эту команду может только владелец оператора.

Параметры

`IF EXISTS`

Не считать ошибкой, если оператор не существует. В этом случае будет выдано замечание.

`имя`

Имя существующего оператора (возможно, дополненное схемой).

`тип_слева`

Тип данных левого операнда оператора; если у оператора нет левого операнда, укажите `NONE`.

`тип_справа`

Тип данных правого операнда оператора.

`CASCADE`

Автоматически удалять объекты, зависящие от данного оператора (например, использующие его представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении оператора, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление оператора возведения в степень `a^b` для типа `integer`:

```
DROP OPERATOR ^ (integer, integer);
```

Удаление префиксного оператора двоичного дополнения `~b` для типа `bit`:

```
DROP OPERATOR ~ (none, bit);
```

Удаление нескольких операторов одной командой:

```
DROP OPERATOR ~ (none, bit), ^ (integer, integer);
```

Совместимость

Команда `DROP OPERATOR` отсутствует в стандарте SQL.

См. также

CREATE OPERATOR, ALTER OPERATOR

DROP OPERATOR CLASS

DROP OPERATOR CLASS — удалить класс операторов

Синтаксис

```
DROP OPERATOR CLASS [ IF EXISTS ] имя USING индексный_метод [ CASCADE | RESTRICT ]
```

Описание

`DROP OPERATOR CLASS` удаляет существующий класс операторов. Выполнить эту команду может только владелец класса операторов.

`DROP OPERATOR CLASS` не удаляет операторы или функции, связанные с этим классом. Если же существуют индексы, зависящие от этого класса, класс будет удалён успешно (вместе с индексами), только если добавить указание `CASCADE`.

Параметры

`IF EXISTS`

Не считать ошибкой, если класс операторов не существует. В этом случае будет выдано замечание.

`имя`

Имя существующего класса операторов (возможно, дополненное схемой).

`индексный_метод`

Имя индексного метода, для которого предназначен этот класс операторов.

`CASCADE`

Автоматически удалять объекты, зависящие от данного класса операторов (например, использующие его индексы), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении класса операторов, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примечания

`DROP OPERATOR CLASS` не удалит семейство операторов, содержавшее этот класс, даже если в этом семействе больше ничего не останется (в том числе, если семейство было неявно создано командой `CREATE OPERATOR CLASS`). Пустое семейство операторов безвредно, но порядка ради затем следует удалить и его, командой `DROP OPERATOR FAMILY`; или, возможно, выполнить `DROP OPERATOR FAMILY` в первую очередь.

Примеры

Удаление класса операторов B-дерева с именем `widget_ops`:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

Эта команда не будет выполнена, если в базе существуют индексы, использующие этот класс. Чтобы удалить такие индексы вместе с классом операторов, нужно добавить указание `CASCADE`.

Совместимость

Команда `DROP OPERATOR CLASS` отсутствует в стандарте SQL.

См. также

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

DROP OPERATOR FAMILY — удалить семейство операторов

Синтаксис

```
DROP OPERATOR FAMILY [ IF EXISTS ] имя USING индексный_метод [ CASCADE | RESTRICT ]
```

Описание

`DROP OPERATOR FAMILY` удаляет существующее семейство операторов. Выполнить эту команду может только владелец семейства операторов.

`DROP OPERATOR FAMILY` удаляет также все классы операторов, содержащиеся в семействе, но не удаляет связанные с ним операторы или функции. Если от классов операторов, содержащихся в семействе, зависят какие-либо индексы, семейство будет удалено успешно (вместе с классами и индексами), только если добавить указание `CASCADE`.

Параметры

`IF EXISTS`

Не считать ошибкой, если семейство операторов не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) существующего семейства операторов.

индексный_метод

Имя индексного метода, для которого предназначено это семейство операторов.

`CASCADE`

Автоматически удалять объекты, зависящие от данного семейства операторов, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении семейства операторов, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление семейства операторов B-дерева с именем `float_ops`:

```
DROP OPERATOR FAMILY float_ops USING btree;
```

Эта команда не будет выполнена, если в базе существуют индексы, использующие классы операторов из этого семейства. Чтобы удалить такие индексы вместе с семейством операторов, нужно добавить указание `CASCADE`.

Совместимость

Команда `DROP OPERATOR FAMILY` отсутствует в стандарте SQL.

См. также

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

DROP OWNED

DROP OWNED — удалить объекты базы данных, принадлежащие роли

Синтаксис

```
DROP OWNED BY { имя | CURRENT_ROLE | CURRENT_USER | SESSION_USER } [, ...] [ CASCADE |
 RESTRICT ]
```

Описание

`DROP OWNED` удаляет в текущей базе данных все объекты, принадлежащие любой из указанных ролей. Кроме того, эти роли лишаются всех прав, которые они имели для объектов текущей базы данных или общих объектов (баз данных, табличных пространств, параметров конфигурации).

Параметры

имя

Имя роли, все объекты которой будут уничтожены, а права отозваны.

`CASCADE`

Автоматически удалять объекты, зависящие от каждого затрагиваемого объекта, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении объектов, принадлежащих роли, если от каких-либо из них зависят другие объекты в базе данных. Это поведение по умолчанию.

Примечания

`DROP OWNED` часто применяется при подготовке к удалению одной или нескольких ролей. Так как команда `DROP OWNED` затрагивает объекты только в текущей базе данных, обычно её нужно выполнять в каждой базе данных, которая содержит объекты, принадлежащие удаляемой роли.

С указанием `CASCADE` эта команда может рекурсивно удалить объекты, принадлежащие и другим пользователям.

Команда `REASSIGN OWNED` даёт альтернативную возможность — сменить владельца для всех объектов базы, принадлежащих одной или нескольким ролям. Однако `REASSIGN OWNED` не затрагивает права, назначенные для объектов, не принадлежащих указанным ролям.

Базы данных и табличные пространства, принадлежащие указанным ролям, эта команда не удаляет.

За подробностями обратитесь к Разделу 20.4.

Совместимость

Команда `DROP OWNED` — расширение Postgres Pro.

См. также

REASSIGN OWNED, DROP ROLE

DROP POLICY

DROP POLICY — удалить политику защиты на уровне строк для таблицы

Синтаксис

```
DROP POLICY [ IF EXISTS ] имя ON имя_таблицы [ CASCADE | RESTRICT ]
```

Описание

`DROP POLICY` удаляет указанную политику из таблицы. Заметьте, что если из таблицы удаляется последняя политика, а в таблице продолжает действовать защита на уровне строк (включённая командой `ALTER TABLE`), будет применена политика запрета по умолчанию. Отключить защиту на уровне строк для таблицы можно с помощью команды `ALTER TABLE ... DISABLE ROW LEVEL SECURITY`, независимо от того, определены ли политики для этой таблицы или нет.

Параметры

`IF EXISTS`

Не считать ошибкой, если политика не существует. В этом случае будет выдано замечание.

`имя`

Имя политики, подлежащей удалению.

`имя_таблицы`

Имя (возможно, дополненное схемой) таблицы, к которой применяется эта политика.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от политик не зависят никакие объекты.

Примеры

Удаление политики `p1` из таблицы `my_table`:

```
DROP POLICY p1 ON my_table;
```

Совместимость

`DROP POLICY` является расширением Postgres Pro.

См. также

CREATE POLICY, ALTER POLICY

DROP PROCEDURE

DROP PROCEDURE — удалить процедуру

Синтаксис

```
DROP PROCEDURE [ IF EXISTS ] имя [ ( [ [ режим_аргумента ] [ имя_аргумента
 ] тип_аргумента [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

Описание

`DROP PROCEDURE` удаляет определение одной или нескольких существующих процедур. Пользователь, выполняющий эту команду, должен быть владельцем процедур(ы). Помимо имени процедуры обычно требуется указать типы её аргументов, так как в базе данных могут существовать несколько процедур с одним именем, но с разными списками аргументов.

Параметры

`IF EXISTS`

Не считать ошибкой, если процедура не существует. В этом случае будет выдано замечание.

`имя`

Имя существующей процедуры (возможно, дополненное схемой).

`режим_аргумента`

Режим аргумента: `IN` или `VARIADIC`. По умолчанию подразумевается `IN` (но учтите следующие замечания).

`имя_аргумента`

Имя аргумента. Заметьте, что на самом деле `DROP PROCEDURE` не обращает внимание на имена аргументов, так как для идентификации процедуры используются только их типы.

`тип_аргумента`

Тип данных аргумента или аргументов (возможно, дополненный именем схемы), если таковые имеются. Подробности следуют ниже.

`CASCADE`

Автоматически удалять объекты, зависящие от данной процедуры, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении процедуры, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примечания

Если существует только одна процедура с заданным именем, список аргументов можно опустить. В этом случае также следует опустить скобки.

В Postgres Pro достаточно перечислить только входные аргументы (включая аргументы `INOUT`), так как одноимённые подпрограммы не могут иметь одинаковые списки входных аргументов. Команда `DROP` на самом деле не проверяет корректность типов для выходных аргументов (`OUT`), поэтому аргументы, помеченные `OUT`, ни на что не влияют. Тем не менее, рекомендуется указывать их для согласованности с соответствующей командой `CREATE`.

Для совместимости со стандартом SQL также возможно записать все аргументы любых типов (включая аргументы `OUT`) без маркеров *`режим_аргумента`*. В этом случае типы выходных аргументов (`OUT`) будут сверяться с указанными при создании. При этом возникает неоднозначность, так как в случае отсутствия маркеров *`режим_аргумента`* в списке аргументов неясно, какое поведение ожидается. Команда `DROP` попытается выполнить поиск обоими способами и выдаст ошибку, если будут найдены две разные процедуры. Для исключения такой неоднозначности рекомендуется явно указывать маркеры `IN`, а не полагаться на то, что они подразумеваются по умолчанию; таким образом будет выбираться традиционное поведение Postgres Pro.

Описанные правила поиска действуют также с другими командами, работающими с существующими процедурами, например `ALTER PROCEDURE` и `COMMENT ON PROCEDURE`.

Примеры

Если существует только одна процедура `do_db_maintenance`, этой команды будет достаточно, чтобы удалить её:

```
DROP PROCEDURE do_db_maintenance;
```

При таком определении процедуры:

```
CREATE PROCEDURE do_db_maintenance(IN target_schema text, OUT results text) ...
```

любая из следующих команд удалит её:

```
DROP PROCEDURE do_db_maintenance(IN target_schema text, OUT results text);
DROP PROCEDURE do_db_maintenance(IN text, OUT text);
DROP PROCEDURE do_db_maintenance(IN text);
DROP PROCEDURE do_db_maintenance(text);
DROP PROCEDURE do_db_maintenance(text, text);  -- возможна неоднозначность
```

Однако последняя команда будет неоднозначной, если существует, например, такая процедура:

```
CREATE PROCEDURE do_db_maintenance(IN target_schema text, IN options text) ...
```

Совместимость

Эта команда соответствует стандарту SQL, но дополнена расширениями Postgres Pro:

- Стандарт позволяет удалять с помощью этой команды только одну процедуру.
- Указание `IF EXISTS` является расширением.
- Возможность указывать режимы и имена аргументов является расширением, а правила поиска при указании режимов могут меняться.

См. также

CREATE PROCEDURE, ALTER PROCEDURE, DROP FUNCTION, DROP ROUTINE

DROP PROFILE

DROP PROFILE — удалить профиль аутентификации

Синтаксис

```
DROP PROFILE [ IF EXISTS ] имя
```

Описание

Команда `DROP PROFILE` удаляет указанный профиль. Выполнять эту команду разрешено только суперпользователям базы данных или пользователям с правами роли `pg_manage_profiles`.

Профиль нельзя удалить, если он используется в какой-либо базе данных в кластере; в этом случае выдаётся ошибка. Чтобы профиль можно было удалить, вы должны удалить все роли, которым он назначен, или переназначить им другой профиль. Для переназначения профиля воспользуйтесь командой ALTER ROLE.

Параметры

`IF EXISTS`

Не считать ошибкой, если профиль не существует. В этом случае будет выдано замечание.

имя

Имя профиля, подлежащего удалению.

Примеры

Удаление указанного профиля:

```
DROP PROFILE admin_profile;
```

См. также

CREATE PROFILE, ALTER PROFILE, ALTER ROLE

DROP PUBLICATION

DROP PUBLICATION — удалить публикацию

Синтаксис

```
DROP PUBLICATION [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP PUBLICATION` удаляет существующую публикацию из базы данных.

Удалить публикацию может только её владелец или суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если публикация не существует. В этом случае будет выдано замечание.

`имя`

Имя существующей публикации.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от публикаций не зависят никакие объекты.

Примеры

Удаление публикации:

```
DROP PUBLICATION mypublication;
```

Совместимость

`DROP PUBLICATION` является расширением Postgres Pro.

См. также

CREATE PUBLICATION, ALTER PUBLICATION

DROP ROLE

DROP ROLE — удалить роль в базе данных

Синтаксис

```
DROP ROLE [ IF EXISTS ] имя [, ...]
```

Описание

`DROP ROLE` удаляет указанные роли. Удалить роль суперпользователя может только суперпользователь, а чтобы удалить роль обычного пользователя, необходимо иметь право `CREATEROLE` и `ADMIN OPTION` для роли.

Если на эту роль есть ссылки в какой-либо базе данных в кластере, возникнет ошибка и роль не будет удалена. Прежде чем удалять роль, необходимо удалить все принадлежащие ей объекты (или сменить их владельца), а также лишить её данных ей прав для других объектов. Для этой цели можно применить команды `REASSIGN OWNED` и `DROP OWNED`; за подробностями обратитесь к Разделу 20.4.

Однако ликвидировать членство в ролях, связанное с этой ролью, не требуется; `DROP ROLE` автоматически исключит данную роль из других ролей, и третьи роли из данной. Сами роли при этом не удаляются и другим образом никак не затрагиваются.

Параметры

`IF EXISTS`

Не считать ошибкой, если роль не существует. В этом случае будет выдано замечание.

имя

Имя роли, подлежащей удалению.

Примечания

Postgres Pro включает программу dropuser, которая предоставляет ту же функциональность (на самом деле она вызывает эту команду), но может запускаться в командной оболочке.

Примеры

Удаление роли:

```
DROP ROLE jonathan;
```

Совместимость

В стандарте SQL определена команда `DROP ROLE`, но она может удалять только по одной роли, а для её выполнения требуются другие права, не такие как в Postgres Pro.

См. также

CREATE ROLE, ALTER ROLE, SET ROLE

DROP ROUTINE

DROP ROUTINE — удалить подпрограмму

Синтаксис

```
DROP ROUTINE [ IF EXISTS ] имя [ ( [ [ режим_аргумента ] [ имя_аргумента
 ] тип_аргумента [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

Описание

`DROP ROUTINE` удаляет определение одной или нескольких подпрограмм. «Подпрограммами» считаются агрегатные и обычные функции, а также процедуры. Описание параметров, дополнительные примеры и подробности приведены в описаниях DROP AGGREGATE, DROP FUNCTION и DROP PROCEDURE.

Примечания

Правила поиска, реализуемые командой `DROP ROUTINE`, по сути не отличаются от правил `DROP PROCEDURE`; в частности, `DROP ROUTINE` таким же образом по возможности рассматривает список аргументов без маркеров *`режим_аргумента`* как соответствующий определению стандарта SQL, в котором аргументы `OUT` включаются в этот список. (`DROP AGGREGATE` и `DROP FUNCTION` действуют по-другому.)

В некоторых случаях, когда одно имя дано нескольким подпрограммам разных видов, команда `DROP ROUTINE` может не справиться с неоднозначностью, тогда как более конкретная команда (`DROP FUNCTION` и т. п.) могла бы выполнить удаление. Решить эту проблему можно, уточнив список аргументов.

Эти правила поиска также применяются в других командах, работающих с существующими подпрограммами, например `ALTER ROUTINE` и `COMMENT ON ROUTINE`.

Примеры

Удаление подпрограммы `foo` для типа `integer`:

```
DROP ROUTINE foo(integer);
```

Эта команда будет работать независимо от того, является ли `foo` агрегатом, функцией или процедурой.

Совместимость

Эта команда соответствует стандарту SQL, но дополнена следующими расширениями Postgres Pro:

- Стандарт позволяет удалять с помощью этой команды только одну подпрограмму.
- Указание `IF EXISTS` является расширением.
- Возможность указывать режимы и имена аргументов является расширением, а правила поиска при указании режимов могут меняться.
- Поддержка пользовательских агрегатных функций относится к расширениям.

См. также

DROP AGGREGATE, DROP FUNCTION, DROP PROCEDURE, ALTER ROUTINE

Заметьте, что также отсутствует команда `CREATE ROUTINE`.

DROP RULE

DROP RULE — удалить правило перезаписи

Синтаксис

```
DROP RULE [ IF EXISTS ] имя ON имя_таблицы [ CASCADE | RESTRICT ]
```

Описание

`DROP RULE` удаляет правило перезаписи.

Параметры

`IF EXISTS`

Не считать ошибкой, если правило не существует. В этом случае будет выдано замечание.

`имя`

Имя правила, подлежащего удалению.

`имя_таблицы`

Имя (возможно, дополненное схемой) существующей таблицы (или представления), к которой применяется это правило.

`CASCADE`

Автоматически удалять объекты, зависящие от данного правила, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении правила, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление правила перезаписи `newrule`:

```
DROP RULE newrule ON mytable;
```

Совместимость

`DROP RULE` является языковым расширением Postgres Pro, как и вся система перезаписи запросов.

См. также

CREATE RULE, ALTER RULE

DROP SCHEMA

DROP SCHEMA — удалить схему

Синтаксис

```
DROP SCHEMA [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP SCHEMA` удаляет схемы из базы данных.

Схему может удалить только её владелец или суперпользователь. Заметьте, что владелец может удалить схему (вместе со всеми содержащимися в ней объектами), даже если он не владеет некоторыми объектами в своей схеме.

Параметры

`IF EXISTS`

Не считать ошибкой, если схема не существует. В этом случае будет выдано замечание.

`имя`

Имя схемы.

`CASCADE`

Автоматически удалять объекты, содержащиеся в этой схеме (таблицы, функции и т. д.), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении схемы, если она содержит какие-либо объекты. Это поведение по умолчанию.

Примечания

С указанием `CASCADE` эта команда может удалить объекты не только в данной схеме, но и в других.

Примеры

Удаление схемы `mystuff` из базы данных вместе со всем, что в ней содержится:

```
DROP SCHEMA mystuff CASCADE;
```

Совместимость

Команда `DROP SCHEMA` полностью соответствует стандарту SQL, но возможность удалять в одной команде несколько схем и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

DROP SEQUENCE — удалить последовательность

Синтаксис

```
DROP SEQUENCE [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP SEQUENCE` удаляет генераторы числовых последовательностей. Удалить последовательность может только её владелец или суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если последовательность не существует. В этом случае будет выдано замечание.

`имя`

Имя последовательности (возможно, дополненное схемой).

`CASCADE`

Автоматически удалять объекты, зависящие от данной последовательности, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении последовательности, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление последовательности `serial`:

```
DROP SEQUENCE serial;
```

Совместимость

`DROP SEQUENCE` соответствует стандарту SQL, но возможность удалять в одной команде несколько последовательностей и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

CREATE SEQUENCE, ALTER SEQUENCE

DROP SERVER

DROP SERVER — удалить описание стороннего сервера

Синтаксис

```
DROP SERVER [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP SERVER` удаляет существующее описание стороннего сервера. Пользователь, выполняющий эту команду, должен быть владельцем сервера.

Параметры

`IF EXISTS`

Не считать ошибкой, если сервер не существует. В этом случае будет выдано замечание.

`имя`

Имя существующего сервера.

`CASCADE`

Автоматически удалять объекты, зависящие от данного триггера, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении сервера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление определения сервера `foo`, если оно существует:

```
DROP SERVER IF EXISTS foo;
```

Совместимость

`DROP SERVER` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Предложение `IF EXISTS` является расширением Postgres Pro.

См. также

CREATE SERVER, ALTER SERVER

DROP STATISTICS

DROP STATISTICS — удалить расширенную статистику

Синтаксис

```
DROP STATISTICS [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP STATISTICS` удаляет объект(ы) статистики из базы данных. Удалить объект статистики может только владелец объекта, владелец схемы или суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если объект статистики не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) объекта статистики, подлежащего удалению.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от статистики не зависят никакие объекты.

Примеры

Удаление двух объектов статистики в разных схемах (если эти объекты отсутствуют, ошибки не будет):

```
DROP STATISTICS IF EXISTS
    accounting.users_uid_creation,
    public.grants_user_role;
```

Совместимость

Оператор `DROP STATISTICS` отсутствует в стандарте SQL.

См. также

ALTER STATISTICS, CREATE STATISTICS

DROP SUBSCRIPTION

DROP SUBSCRIPTION — удалить подписку

Синтаксис

```
DROP SUBSCRIPTION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP SUBSCRIPTION` удаляет подписку из кластера баз данных.

Выполнить эту команду может только владелец подписки.

Команду `DROP SUBSCRIPTION` нельзя выполнять в блоке транзакции, если подписка связана со слотом репликации. (Для освобождения слота можно использовать команду `ALTER SUBSCRIPTION`.)

Параметры

имя

Имя подписки, подлежащей удалению.

`CASCADE`
`RESTRICT`

Эти ключевые слова игнорируются, так как от подписок не зависят никакие объекты.

Примечания

При удалении подписки, связанной со слотом репликации на удалённом узле (это типичная ситуация), команда `DROP SUBSCRIPTION` подключится к удалённому узлу и попытается удалить слот репликации (и все остальные слоты синхронизации таблиц) в ходе этой операции. Это необходимо для освобождения ресурсов, выделенных для подписки на удалённом узле. Если при этом происходит сбой, либо из-за недоступности удалённого узла, либо из-за ошибки при удалении слота репликации, либо вообще из-за его отсутствия, команда `DROP SUBSCRIPTION` прерывается. Для разрешения этой ситуации сначала отключите подписку, выполнив `ALTER SUBSCRIPTION ... DISABLE`, а затем разорвите связь подписки с этим слотом репликации, выполнив команду `ALTER SUBSCRIPTION ... SET (slot_name = NONE)`. После этого команда `DROP SUBSCRIPTION` не будет пытаться выполнять какие-либо действия на удалённом узле. Заметьте, что если удалённый слот репликации фактически продолжает существовать, его (и все связанные слоты синхронизации таблиц) нужно будет удалить вручную; в противном случае для него и связанных слотов будет по-прежнему сохраняться WAL, что в конце концов может привести к переполнению диска. См. также Подраздел 28.2.1.

Если подписка связана со слотом репликации, команду `DROP SUBSCRIPTION` нельзя выполнять внутри блока транзакции.

Примеры

Удаление подписки:

```
DROP SUBSCRIPTION mysub;
```

Совместимость

`DROP SUBSCRIPTION` является расширением Postgres Pro.

См. также

CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

DROP TABLE

DROP TABLE — удалить таблицу

Синтаксис

```
DROP TABLE [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP TABLE` удаляет таблицы из базы данных. Удалить таблицу может только её владелец, владелец схемы или суперпользователь. Чтобы опустошить таблицу, не удаляя её саму, вместо этой команды следует использовать `DELETE` или `TRUNCATE`.

`DROP TABLE` всегда удаляет все индексы, правила, триггеры и ограничения, существующие в целевой таблице. Однако чтобы удалить таблицу, на которую ссылается представление или ограничение внешнего ключа в другой таблице, необходимо дополнительно указать `CASCADE`. (С указанием `CASCADE` зависимое представление удаляется полностью, тогда как в случае с ограничением внешнего ключа удаляется именно это ограничение, а не вся таблица, к которой оно относится.)

Параметры

`IF EXISTS`

Не считать ошибкой, если таблица не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) таблицы, подлежащей удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данной таблицы (например, представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении таблицы, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление таблиц `films` и `distributors`:

```
DROP TABLE films, distributors;
```

Совместимость

Эта команда соответствует стандарту SQL, но возможность удалять в одной команде несколько таблиц и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

DROP TABLESPACE — удалить табличное пространство

Синтаксис

```
DROP TABLESPACE [ IF EXISTS ] имя
```

Описание

`DROP TABLESPACE` удаляет табличное пространство из системы.

Удалить табличное пространство может только его владелец или суперпользователь. Перед удалением его необходимо очистить от всех объектов базы данных. Даже если в текущей базе данных не будет ни одного объекта, находящегося в этом пространстве, в нём вполне могут оставаться объекты других баз данных. Кроме того, если табличное пространство указано в списке temp_tablespaces любого активного сеанса, команда `DROP` может завершиться ошибкой, если в этом пространстве окажутся временные файлы.

Параметры

`IF EXISTS`

Не считать ошибкой, если табличное пространство не существует. В этом случае будет выдано замечание.

`имя`

Имя табличного пространства.

Примечания

`DROP TABLESPACE` не может быть выполнена в блоке транзакции.

Примеры

Удаление табличного пространства `mystuff` из системы:

```
DROP TABLESPACE mystuff;
```

Совместимость

`DROP TABLESPACE` является расширением Postgres Pro.

См. также

CREATE TABLESPACE, ALTER TABLESPACE

DROP TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH CONFIGURATION — удалить конфигурацию текстового поиска

Синтаксис

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP TEXT SEARCH CONFIGURATION` удаляет существующую конфигурацию текстового поиска. Выполнить эту команду может только владелец конфигурации.

Параметры

`IF EXISTS`

Не считать ошибкой, если конфигурация текстового поиска не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) существующей конфигурации текстового поиска.

`CASCADE`

Автоматически удалять объекты, зависящие от данной конфигурации текстового поиска, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении конфигурации текстового поиска, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление конфигурации текстового поиска `my_english`:

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

Эта команда не будет выполнена, если какие-либо индексы ссылаются на эту конфигурацию в вызовах `to_tsvector`. Чтобы удалить такие индексы вместе с конфигурацией, следует добавить указание `CASCADE`.

Совместимость

Оператор `DROP TEXT SEARCH CONFIGURATION` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH DICTIONARY

DROP TEXT SEARCH DICTIONARY — удалить словарь текстового поиска

Синтаксис

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP TEXT SEARCH DICTIONARY` удаляет существующий словарь текстового поиска. Выполнить эту команду может только владелец словаря.

Параметры

`IF EXISTS`

Не считать ошибкой, если словарь текстового поиска не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) существующего словаря текстового поиска.

`CASCADE`

Автоматически удалять объекты, зависящие от данного словаря текстового поиска, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении словаря текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удалить словарь текстового поиска `english`:

```
DROP TEXT SEARCH DICTIONARY english;
```

Эта команда не будет выполнена, если существуют конфигурации текстового поиска, использующие этот словарь. Чтобы удалить такие конфигурации вместе со словарём, следует добавить указание `CASCADE`.

Совместимость

Оператор `DROP TEXT SEARCH DICTIONARY` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY

DROP TEXT SEARCH PARSER

DROP TEXT SEARCH PARSER — удалить анализатор текстового поиска

Синтаксис

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP TEXT SEARCH PARSER` удаляет существующий анализатор текстового поиска. Выполнить эту команду может только суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если анализатор текстового поиска не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) существующего анализатора текстового поиска.

`CASCADE`

Автоматически удалять объекты, зависящие от данного анализатора текстового поиска, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении анализатора текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление анализатора текстового поиска `my_parser`:

```
DROP TEXT SEARCH PARSER my_parser;
```

Эта команда не будет выполнена, если существуют конфигурации текстового поиска, использующие это анализатор. Чтобы удалить такие конфигурации вместе с анализатором, следует добавить указание `CASCADE`.

Совместимость

Оператор `DROP TEXT SEARCH PARSER` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

DROP TEXT SEARCH TEMPLATE

DROP TEXT SEARCH TEMPLATE — удалить шаблон текстового поиска

Синтаксис

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] имя [ CASCADE | RESTRICT ]
```

Описание

`DROP TEXT SEARCH TEMPLATE` удаляет существующий шаблон текстового поиска. Выполнить эту команду может только суперпользователь.

Параметры

`IF EXISTS`

Не считать ошибкой, если шаблон текстового поиска не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) существующего шаблона текстового поиска.

`CASCADE`

Автоматически удалять объекты, зависящие от данного шаблона текстового поиска, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении шаблона текстового поиска, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление шаблона текстового поиска `thesaurus`:

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

Эта команда не будет выполнена, если существуют словари текстового поиска, использующие этот шаблон. Чтобы удалить такие словари вместе с шаблоном, следует добавить указание `CASCADE`.

Совместимость

Оператор `DROP TEXT SEARCH TEMPLATE` отсутствует в стандарте SQL.

См. также

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

DROP TRANSFORM

DROP TRANSFORM — удалить трансформацию

Синтаксис

```
DROP TRANSFORM [ IF EXISTS ] FOR имя_типа LANGUAGE имя_языка [ CASCADE | RESTRICT ]
```

Описание

`DROP TRANSFORM` удаляет ранее определённую трансформацию.

Чтобы удалить трансформацию, необходимо быть владельцем типа и языка. Такие же требования действуют и при создании трансформации.

Параметры

`IF EXISTS`

Не считать ошибкой, если трансформация не существует. В этом случае будет выдано замечание.

`имя_типа`

Имя типа данных, для которого предназначена трансформация.

`имя_языка`

Имя языка, для которого предназначена трансформация.

`CASCADE`

Автоматически удалять объекты, зависящие от данной трансформации, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении трансформации, если от неё зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление трансформации для типа `hstore` и языка `plpython3u`:

```
DROP TRANSFORM FOR hstore LANGUAGE plpython3u;
```

Совместимость

Эта форма `DROP TRANSFORM` является расширением Postgres Pro. За подробностями обратитесь к описанию CREATE TRANSFORM.

См. также

CREATE TRANSFORM

DROP TRIGGER

DROP TRIGGER — удалить триггер

Синтаксис

```
DROP TRIGGER [ IF EXISTS ] имя ON имя_таблицы [ CASCADE | RESTRICT ]
```

Описание

`DROP TRIGGER` удаляет существующее определение триггера. Пользователь, выполняющий эту команду, должен быть владельцем таблицы, для которой определён данный триггер.

Параметры

`IF EXISTS`

Не считать ошибкой, если триггер не существует. В этом случае будет выдано замечание.

`имя`

Имя триггера, подлежащего удалению.

`имя_таблицы`

Имя (возможно, дополненное схемой) таблицы, для которой определён триггер.

`CASCADE`

Автоматически удалять объекты, зависящие от данного триггера, и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении триггера, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление триггера `if_dist_exists` в таблице `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

Совместимость

Оператор `DROP TRIGGER` в Postgres Pro несовместим со стандартом SQL. В стандарте имена триггеров не считаются локальными по отношению к таблицам, так что синтаксис команды проще: `DROP TRIGGER имя`.

См. также

CREATE TRIGGER

DROP TYPE

DROP TYPE — удалить тип данных

Синтаксис

```
DROP TYPE [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP TYPE` удаляет определённый пользователем тип данных. Удалить тип может только его владелец.

Параметры

`IF EXISTS`

Не считать ошибкой, если тип не существует. В этом случае будет выдано замечание.

имя

Имя (возможно, дополненное схемой) типа данных, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного типа (например, столбцы таблиц, функции и операторы), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении типа, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Удаление типа данных `box`:

```
DROP TYPE box;
```

Совместимость

Эта команда подобна соответствующей команде в стандарте SQL, но указание `IF EXISTS` является расширением Postgres Pro. Однако учтите, что команда `CREATE TYPE` и механизм расширения типов в Postgres Pro значительно отличаются от стандарта SQL.

См. также

ALTER TYPE, CREATE TYPE

DROP USER

DROP USER — удалить роль в базе данных

Синтаксис

```
DROP USER [ IF EXISTS ] имя [, ...]
```

Описание

`DROP USER` — просто альтернативное написание команды `DROP ROLE`.

Совместимость

`DROP USER` является расширением Postgres Pro. В стандарте SQL определение пользователей считается зависимым от реализации.

См. также

DROP ROLE

DROP USER MAPPING

DROP USER MAPPING — удалить сопоставление пользователя для стороннего сервера

Синтаксис

```
DROP USER MAPPING [ IF EXISTS ] FOR { имя_пользователя | USER | CURRENT_ROLE |
 CURRENT_USER | PUBLIC } SERVER имя_сервера
```

Описание

`DROP USER MAPPING` удаляет существующее сопоставление пользователя для стороннего сервера.

Владелец стороннего сервера может удалить заданные для этого сервера сопоставления любых пользователей. Кроме того, обычный пользователь может удалить сопоставление для собственного имени, если он имеет право `USAGE` для сервера.

Параметры

`IF EXISTS`

Не считать ошибкой, если сопоставление не существует. В этом случае будет выдано замечание.

`имя_пользователя`

Имя пользователя для сопоставления. Значения `CURRENT_ROLE`, `CURRENT_USER` и `USER` соответствуют имени текущего пользователя. Значение `PUBLIC` соответствует именам всех текущих и будущих пользователей системы.

`имя_сервера`

Имя сервера, для которого меняется сопоставление пользователей.

Примеры

Удаление сопоставления пользователя `bob` для сервера `foo`, если оно существует:

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

Совместимость

`DROP USER MAPPING` соответствует стандарту ISO/IEC 9075-9 (SQL/MED). Предложение `IF EXISTS` является расширением Postgres Pro.

См. также

CREATE USER MAPPING, ALTER USER MAPPING

DROP VIEW

DROP VIEW — удалить представление

Синтаксис

```
DROP VIEW [ IF EXISTS ] имя [, ...] [ CASCADE | RESTRICT ]
```

Описание

`DROP VIEW` удаляет существующее представление. Выполнить эту команду может только владелец представления.

Параметры

`IF EXISTS`

Не считать ошибкой, если представление не существует. В этом случае будет выдано замечание.

`имя`

Имя (возможно, дополненное схемой) представления, подлежащего удалению.

`CASCADE`

Автоматически удалять объекты, зависящие от данного представления (например, другие представления), и, в свою очередь, все зависящие от них объекты (см. Раздел 5.15).

`RESTRICT`

Отказать в удалении представления, если от него зависят какие-либо объекты. Это поведение по умолчанию.

Примеры

Эта команда удаляет представление с именем `kinds`:

```
DROP VIEW kinds;
```

Совместимость

Эта команда соответствует стандарту SQL, но возможность удалять в одной команде несколько представлений и указание `IF EXISTS` являются расширениями Postgres Pro.

См. также

ALTER VIEW, CREATE VIEW

END

END — зафиксировать текущую транзакцию

Синтаксис

```
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Описание

`END` фиксирует текущую транзакцию. Все изменения, произведённые этой транзакцией, становятся видимыми для других и гарантированно сохранятся в случае сбоя. Эта команда является расширением Postgres Pro и равнозначна команде `COMMIT`.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

`AND CHAIN`

Если добавляется указание `AND CHAIN`, сразу после окончания текущей транзакции начинается новая с такими же характеристиками транзакции (см. SET TRANSACTION). В противном случае новая транзакция не начинается.

Примечания

Для прерывания транзакции используйте `ROLLBACK`.

При попытке выполнить `END` вне транзакции ничего не произойдёт, но будет выдано предупреждение.

Примеры

Следующая команда фиксирует текущую транзакцию и сохраняет все изменения:

```
END;
```

Совместимость

`END` является расширением Postgres Pro и выполняет ту же функцию, что и оператор `COMMIT`, описанный в стандарте SQL.

См. также

BEGIN, COMMIT, ROLLBACK

EXECUTE

EXECUTE — выполнить подготовленный оператор

Синтаксис

```
EXECUTE имя [ ( параметр [, ...] ) ]
```

Описание

`EXECUTE` выполняет подготовленный ранее оператор. Так как подготовленные операторы существуют только в рамках сеанса, они должны создаваться командой `PREPARE`, выполненной в текущем сеансе ранее.

Если команда `PREPARE`, создающая оператор, определяет некоторый набор параметров, команде `EXECUTE` должны быть переданы подходящие значения этих параметров; в противном случае возникнет ошибка. Заметьте, что подготовленные операторы (в отличие от функций) не перегружаются в зависимости от типа или числа параметров; имя подготовленного оператора должно быть уникальным в рамках текущего сеанса.

Чтобы узнать больше о создании и использовании подготовленных операторов, обратитесь к PREPARE.

Параметры

`имя`

Имя подготовленного оператора, который будет выполнен.

`параметр`

Фактическое значение параметра подготовленного оператора. Это может быть выражение, выдающее значение, совместимое с типом данных этого параметра, который был определён при создании подготовленного оператора.

Выводимая информация

Метка команды, возвращаемая `EXECUTE`, соответствует подготовленному оператору, а не оператору `EXECUTE`.

Примеры

Примеры приведены в разделе Примеры описания PREPARE.

Совместимость

В стандарте SQL есть оператор `EXECUTE`, но он предназначен только для применения во встраиваемом SQL. Эта версия оператора `EXECUTE` имеет также несколько другой синтаксис.

См. также

DEALLOCATE, PREPARE

EXPLAIN

EXPLAIN — показать план выполнения оператора

Синтаксис

```
EXPLAIN [ ( параметр [, ...] ) ] оператор

Здесь допускается параметр:

    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    SETTINGS [ boolean ]
    GENERIC_PLAN [ boolean ]
    BUFFERS [ boolean ]
    SERIALIZE [ { NONE | TEXT | BINARY } ]
    WAL [ boolean ]
    TIMING [ boolean ]
    SUMMARY [ boolean ]
    MEMORY [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

Описание

Эта команда выводит план выполнения, генерируемый планировщиком Postgres Pro для заданного оператора. План выполнения показывает, как будут сканироваться таблицы, затрагиваемые оператором — просто последовательно, по индексу и т. д. — а если запрос связывает несколько таблиц, какой алгоритм соединения будет выбран для объединения считанных из них строк.

Наибольший интерес в выводимой информации представляет ожидаемая стоимость выполнения оператора, которая показывает, сколько, по мнению планировщика, будет выполняться этот оператор (это значение измеряется в единицах стоимости, которые не имеют точного определения, но обычно это обращение к странице на диске). Фактически выводятся два числа: стоимость запуска до выдачи первой строки и общая стоимость выдачи всех строк. Для большинства запросов важна общая стоимость, но в таких контекстах, как подзапрос в `EXISTS`, планировщик будет минимизировать стоимость запуска, а не общую стоимость (так как исполнение запроса всё равно завершится сразу после получения одной строки). Кроме того, если количество возвращаемых строк ограничивается предложением `LIMIT`, планировщик интерполирует стоимость между двумя этими числами, выбирая наиболее выгодный план.

С параметром `ANALYZE` оператор будет выполнен на самом деле, а не только запланирован. При этом в вывод добавляются фактические сведения о времени выполнения, включая общее время, затраченное на каждый узел плана (в миллисекундах) и общее число строк, выданных в результате. Это помогает понять, насколько близки к реальности предварительные оценки планировщика.

Важно

Имейте в виду, что с указанием `ANALYZE` оператор действительно выполняется. Хотя `EXPLAIN` отбрасывает результат, который вернул бы `SELECT`, в остальном все действия выполняются как обычно. Если вы хотите выполнить `EXPLAIN ANALYZE` с командой `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `CREATE TABLE AS` или `EXECUTE`, не допуская изменения данных этой командой, воспользуйтесь таким приёмом:

```
BEGIN;
EXPLAIN ANALYZE ...;
```

```
ROLLBACK;
```

Параметры

`ANALYZE`

Выполнить команду и вывести фактическое время выполнения и другую статистику. По умолчанию этот параметр равен `FALSE`.

`VERBOSE`

Вывести дополнительную информацию о плане запроса. В частности, включить список столбцов результата для каждого узла в дереве плана, дополнить схемой имена таблиц и функций, всегда указывать для переменных в выражениях псевдоним их таблицы, а также выводить имена всех триггеров, для которых выдаётся статистика. Также для данного запроса будет выводиться идентификатор, если он был вычислен; за подробностями обратитесь к compute_query_id. По умолчанию этот параметр равен `FALSE`.

`COSTS`

Вывести рассчитанную стоимость запуска и общую стоимость каждого узла плана, а также рассчитанное число строк и ширину каждой строки. Этот параметр по умолчанию равен `TRUE`.

`SETTINGS`

Вывести информацию о параметрах конфигурации. А именно, будут выведены параметры, влияющие на планирование, значения которых отличаются от стандартных значений по умолчанию. В отсутствие данного указания подразумевается `FALSE` (эта информация не выводится).

`GENERIC_PLAN`

Использовать операторы с шаблонами параметров в виде `$1` и создать общий план, не зависящий от значений параметров. За подробной информацией об общих планах и типах операторов, поддерживающих параметры, обратитесь к `PREPARE`. Этот параметр нельзя использовать с `ANALYZE`. Значение по умолчанию — `FALSE`.

`BUFFERS`

Включить информацию об использовании буфера. В частности, вывести число попаданий, блоков прочитанных, загрязнённых и записанных в общем буфере, число попаданий, блоков прочитанных, загрязнённых и записанных в локальном буфере, число прочитанных и записанных временных блоков, а также время в миллисекундах, потраченное на чтение и запись блоков файлов данных, локальных блоков и блоков временных файлов, если включён параметр track_io_timing. *Попаданием* (hit) считается ситуация, когда требуемый блок уже находится в кеше и чтения с диска удаётся избежать. Блоки в общем буфере содержат данные обычных таблиц и индексов, в локальном — данные временных таблиц и индексов, а временные блоки предназначены для краткосрочного использования при выполнении сортировки, хеширования, материализации и подобных узлов плана. Число *загрязнённых* блоков (dirtied) показывает, сколько ранее не модифицированных блоков изменила данная операция; тогда как число *записанных* блоков (written) показывает, сколько ранее загрязнённых блоков данный серверный процесс вынес из кеша при обработке запроса. Значения, указываемые для узла верхнего уровня, включают значения всех его дочерних узлов. В текстовом формате выводятся только ненулевые значения. По умолчанию его значение равно `FALSE`.

`SERIALIZE`

Включить информацию о стоимости *сериализации* выходных данных запроса, преобразовывая её в текстовый или двоичный формат для отправки клиенту. Это может занять значительную часть времени обычной работы запроса, если функции вывода типа данных дорогие или если необходимо вывести значения в формате TOAST из отдельного хранилища. При значении по умолчанию, `SERIALIZE NONE`, эти преобразования не происходят. При заданных значениях

`SERIALIZE TEXT` или `SERIALIZE BINARY` необходимое преобразование происходит, а затраченное время оценивается (если не задано значение `TIMING OFF`). Если указан параметр `BUFFERS`, в стоимость также включаются обращения к буферам, участвующим в преобразовании. При этом команда `EXPLAIN` никогда не отправляет полученные данные клиенту, поэтому стоимость передачи по сети таким образом рассчитать нельзя. Сериализация может работать только в режиме `ANALYZE`. Если для `SERIALIZE` не указан аргумент, подразумевается `TEXT`.

`WAL`

Включить информацию о формировании записей WAL. В частности, вывести число записей, число полных образов страниц (fpi, full page images) и объём сгенерированных записей в байтах. В текстовом формате выводятся только ненулевые значения. Этот параметр может использоваться, только если также включён режим `ANALYZE`. По умолчанию он отключён (`FALSE`).

`TIMING`

Включить в вывод фактическое время запуска и время, затраченное на каждый узел. Постоянное чтение системных часов может значительно замедлить запрос, так что если достаточно знать фактическое число строк, имеет смысл сделать этот параметр равным `FALSE`. Время выполнения всего оператора замеряется всегда, даже когда этот параметр выключен и на уровне узлов время не подсчитывается. Этот параметр действует только в режиме `ANALYZE`. По умолчанию его значение равно `TRUE`.

`SUMMARY`

Включить сводку (например, суммарное время) после плана запроса. Сводка включается по умолчанию, когда используется `ANALYZE`, но этот параметр позволяет получить её и с другими вариантами команды. Время планирования в `EXPLAIN EXECUTE` включает время извлечения плана из кеша и время перепланирования, если оно потребовалось.

`MEMORY`

Вывести информацию о потреблении памяти на этапе планирования запросов. А именно, выводить точный объём памяти, используемый структурами данных в оперативной памяти, а также её общий объём с учётом издержек на выделение. Значение по умолчанию: `FALSE`.

`FORMAT`

Установить один из следующих форматов вывода: TEXT, XML, JSON или YAML. Последние три формата содержат ту же информацию, что и текстовый, но больше подходят для программного разбора. По умолчанию выбирается формат `TEXT`.

`boolean`

Включает или отключает заданный параметр. Для включения параметра можно написать `TRUE`, `ON` или `1`, а для отключения — `FALSE`, `OFF` или `0`. Значение *`boolean`* можно опустить, в этом случае подразумевается `TRUE`.

`оператор`

Любой оператор `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `VALUES`, `EXECUTE`, `DECLARE`, `CREATE TABLE AS` и `CREATE MATERIALIZED VIEW AS`, план выполнения которого вас интересует.

Выводимая информация

Результатом команды будет текстовое описание плана, выбранного для *`оператора`*, возможно, дополненное статистикой выполнения. Представленная информация описана в Разделе 14.1.

Примечания

Чтобы планировщик запросов Postgres Pro был достаточно информирован для эффективной оптимизации запросов, данные в `pg_statistic` должны быть актуальными для всех таблиц, задействованных в запросе. Обычно об этом автоматически заботится демон автоочистки. Но если в таблице недавно произошли значительные изменения, может потребоваться вручную выполнить `ANALYZE`, не дожидаясь, пока автоочистка обработает эти изменения.

Измеряя фактическую стоимость выполнения каждого узла в плане, текущая реализация `EXPLAIN ANALYZE` привносит накладные расходы профилирования в выполнение запроса. В результате этого, при запуске запроса командой `EXPLAIN ANALYZE` он может выполняться значительно дольше, чем при обычном выполнении. Объём накладных расходов зависит от природы запроса, а также от используемой платформы. Худшая ситуация наблюдается для узлов плана, которые сами по себе выполняются очень быстро, и в операционных системах, где получение текущего времени относительно длительная операция.

Примеры

Получение плана простого запроса для таблицы, содержащей единственный столбец типа `integer`, с 10000 строк:

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
---------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

План того же запроса, но выведенный в формате JSON:

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                             +
   {                           +
     "Plan": {                 +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                         +
   }                           +
 ]
(1 row)
```

Если в таблице есть индекс, а в запросе присутствует условие `WHERE`, для которого полезен этот индекс, `EXPLAIN` может показать другой план:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                         QUERY PLAN
--------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

План того же запроса, но в формате YAML:

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
-------------------------------
 - Plan:                      +
```

```
       Node Type: "Index Scan"  +
       Scan Direction: "Forward"+
       Index Name: "fi"         +
       Relation Name: "foo"     +
       Alias: "foo"             +
       Startup Cost: 0.00       +
       Total Cost: 5.98         +
       Plan Rows: 1             +
       Plan Width: 4            +
       Index Cond: "(i = 4)"
(1 row)
```

Рассмотрение формата XML оставлено в качестве упражнения для читателя.

План того же запроса без вывода оценок стоимости:

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
----------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

Пример плана для запроса с агрегатной функцией:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                             QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Пример использования `EXPLAIN EXECUTE` для отображения плана выполнения подготовленного запроса:

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                       QUERY PLAN
-------------------------------------------------------------------
------------------------------------------------------
 HashAggregate  (cost=10.77..10.87 rows=10 width=12) (actual time=0.043..0.044 rows=10
 loops=1)
   Group Key: foo
   Batches: 1  Memory Usage: 24kB
   ->  Index Scan using test_pkey on test  (cost=0.29..10.27 rows=99 width=8) (actual
 time=0.009..0.025 rows=99 loops=1)
         Index Cond: ((id > 100) AND (id < 200))
 Planning Time: 0.244 ms
 Execution Time: 0.073 ms
(7 rows)
```

Разумеется, конкретные числа, показанные здесь, зависят от фактического содержимого задействованных таблиц. Также учтите, что эти числа и даже выбранная стратегия выполнения запроса могут меняться от версии к версии Postgres Pro вследствие усовершенствования планировщика.

Кроме того, команда `ANALYZE` при обработке статистических данных производит случайные выборки, так что оценки стоимости могут меняться при каждом чистом запуске `ANALYZE`, даже когда фактическое распределение данных в таблице не меняется.

Обратите внимание, что в предыдущем примере показан «специализированный» план для запроса `EXECUTE` с определёнными значениями параметров. С помощью `GENERIC_PLAN` можно также посмотреть общий план для параметризованного запроса:

```
EXPLAIN (GENERIC_PLAN)
  SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate  (cost=26.79..26.89 rows=10 width=12)
   Group Key: foo
   ->  Index Scan using test_pkey on test  (cost=0.29..24.29 rows=500 width=8)
         Index Cond: ((id > $1) AND (id < $2))
(4 rows)
```

В этом случае анализатор запроса корректно предполагает, что `$1` и `$2` должны иметь тот же тип данных, что и `id`, так что отсутствие информации о типе данных из `PREPARE` не представляет проблемы. В других случаях может потребоваться явно указать типы для символов параметров, что можно сделать путём приведения типов, например:

```
EXPLAIN (GENERIC_PLAN)
  SELECT sum(bar) FROM test
    WHERE id > $1::integer AND id < $2::integer
    GROUP BY foo;
```

Совместимость

Оператор `EXPLAIN` отсутствует в стандарте SQL.

До версии Postgres Pro 9.0 использовался и по-прежнему поддерживается следующий синтаксис

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] оператор
```

Обратите внимание, что параметры должны быть заданы именно в указанном порядке.

См. также

ANALYZE

FETCH

FETCH — получить результат запроса через курсор

Синтаксис

```
FETCH [ направление ] [ FROM | IN ] имя_курсора

Здесь допускается направление:

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE число
    RELATIVE число
    число
    ALL
    FORWARD
    FORWARD число
    FORWARD ALL
    BACKWARD
    BACKWARD число
    BACKWARD ALL
```

Описание

`FETCH` получает строки через ранее созданный курсор.

Курсор связан с определённым положением, что и использует команда `FETCH`. Курсор может располагаться перед первой строкой результата запроса, на любой строке этого результата, либо после последней строки. При создании курсор оказывается перед первой строкой. Когда `FETCH` доходит до конца набора строк, курсор остаётся в положении после последней строки, либо перед первой, при движении назад. После команд `FETCH ALL` и `FETCH BACKWARD ALL` курсор всегда оказывается после последней строки или перед первой, соответственно.

Формы `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` и `RELATIVE` выбирают одну строку после соответствущего перемещения курсора. Если в этом положении строки не оказывается, возвращается пустой результат, а курсор остаётся в достигнутом положении перед первой строкой или после последней.

Формы `FORWARD` и `BACKWARD` получают указанное число строк, сдвигаясь соответственно вперёд или назад; в результате курсор оказывается на последней выданной строке (или перед/после всех строк, если *`число`* превышает количество доступных строк).

Формы `RELATIVE 0`, `FORWARD 0` и `BACKWARD 0` действуют одинаково — они считывают текущую строку, не перемещая курсор, то есть, повторно выбирая строку, выбранную последней. Эта операция будет успешна, только если курсор не расположен до первой или после последней строки; в этом случае строка возвращена не будет.

> **Примечание**
>
> На этой странице описывается применение курсоров на уровне команд SQL. Если вы попытаетесь использовать курсоры внутри функции PL/pgSQL, правила будут другими — см. Подраздел 41.7.3.

Параметры

`направление`

Параметр *`направление`* задаёт направление движения и число выбираемых строк. Он может принимать одно из следующих значений:

`NEXT`

Выбрать следующую строку. Это действие подразумевается по умолчанию, если *`направление`* опущено.

`PRIOR`

Выбрать предыдущую строку.

`FIRST`

Выбрать первую строку запроса (аналогично указанию `ABSOLUTE 1`).

`LAST`

Выбрать последнюю строку запроса (аналогично `ABSOLUTE -1`).

`ABSOLUTE` *`число`*

Выбрать строку под номером *`число`* с начала, либо под номером `abs(`*`число`*`)` с конца, если *`число`* отрицательно. Если *`число`* выходит за границы набора строк, курсор размещается перед первой или после последней строки; в частности, с `ABSOLUTE 0` курсор оказывается перед первой строкой.

`RELATIVE` *`число`*

Выбрать строку под номером *`число`*, считая со следующей вперёд, либо под номером `abs(`*`число`*`)`, считая с предыдущей назад, если *`число`* отрицательно. `RELATIVE 0` повторно считывает текущую строку, если таковая имеется.

`число`

Выбрать следующее *`число`* строк (аналогично `FORWARD` *`число`*).

`ALL`

Выбрать все оставшиеся строки (аналогично `FORWARD ALL`).

`FORWARD`

Выбрать следующую строку (аналогично `NEXT`).

`FORWARD` *`число`*

Выбрать следующее *`число`* строк. `FORWARD 0` повторно выбирает текущую строку.

`FORWARD ALL`

Выбрать все оставшиеся строки.

`BACKWARD`

Выбрать предыдущую строку (аналогично `PRIOR`).

`BACKWARD` *`число`*

Выбрать предыдущее *`число`* строк (с перемещением назад). `BACKWARD 0` повторно выбирает текущую строку.

`BACKWARD ALL`

Выбрать все предыдущие строки (с перемещением назад).

число

Здесь *число* — целочисленная константа, возможно со знаком, определяющая смещение или количество выбираемых строк. Для вариантов `FORWARD` и `BACKWARD` указание отрицательного *числа* равнозначно смене направления `FORWARD` на `BACKWARD` и наоборот.

имя_курсора

Имя открытого курсора.

Выводимая информация

При успешном выполнении `FETCH` возвращает метку команды вида

```
FETCH число
```

Здесь *count* — количество выбранных строк (может быть и нулевым). Заметьте, что в psql метка команды не выдаётся, так как вместо неё psql выводит выбранные строки.

Примечания

Если перемещение курсора в `FETCH` не ограничивается вариантами `FETCH NEXT` или `FETCH FORWARD` с положительным числом, курсор должен быть объявлен с указанием `SCROLL`. Для простых запросов Postgres Pro допускает обратное перемещение курсора, объявленного без `SCROLL`, но на это поведение лучше не рассчитывать. Если курсор объявлен с указанием `NO SCROLL`, перемещение назад запрещается.

Вариант `ABSOLUTE` нисколько не быстрее, чем перемещение к требуемой строке с относительным сдвигом: нижележащий механизм всё равно должен прочитать все промежуточные строки. Выборки по абсолютному отрицательному положению ещё хуже: сначала запрос необходимо прочитать до конца и найти последнюю строку, а затем вернуться назад к указанной строке. Однако перемотка к началу запроса (`FETCH ABSOLUTE 0`) выполняется быстро.

Определить курсор позволяет команда `DECLARE`, а переместить его, не читая данные, — команда `MOVE`.

Примеры

Следующий пример демонстрирует перемещение курсора в таблице:

```
BEGIN WORK;

-- Создание курсора:
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- Получение первых 5 строк через курсор liahona:
FETCH FORWARD 5 FROM liahona;

 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- Получение предыдущей строки:
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
```

```
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- Закрытие курсора и завершение транзакции:
CLOSE liahona;
COMMIT WORK;
```

Совместимость

В стандарте SQL команда `FETCH` определена только для встраиваемого SQL. Описанная здесь реализация `FETCH` возвращает данные подобно оператору `SELECT`, а не помещает их в переменные исполняющей среды. В остальном, `FETCH` полностью прямо-совместима со стандартом SQL.

Формы `FETCH` с `FORWARD` и `BACKWARD`, а также формы `FETCH` *`число`* и `FETCH ALL` (в которых `FORWARD` подразумевается) являются расширениями Postgres Pro.

В стандарте SQL перед именем курсора допускается только указание `FROM`; возможность указать `IN` или опустить оба указания относится к расширениям.

См. также

CLOSE, DECLARE, MOVE

GRANT

GRANT — определить права доступа

Синтаксис

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER |
 MAINTAIN }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] имя_таблицы [, ...]
         | ALL TABLES IN SCHEMA имя_схемы [, ...] }
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( имя_столбца [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( имя_столбца [, ...] ) }
    ON [ TABLE ] имя_таблицы [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE имя_последовательности [, ...]
         | ALL SEQUENCES IN SCHEMA имя_схемы [, ...] }
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE имя_бд [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN имя_домена [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER имя_сервера [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } имя_подпрограммы [ ( [ [ режим_аргумента ]
 [ имя_аргумента ] тип_аргумента [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA имя_схемы [, ...] }
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE имя_языка [, ...]
```

```
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT oid_БО [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { { SET | ALTER SYSTEM } [, ... ] | ALL [ PRIVILEGES ] }
    ON PARAMETER параметр_конфигурации [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA имя_схемы [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE табл_пространство [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE имя_типа [, ...]
    TO указание_роли [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY указание_роли ]

GRANT имя_роли [, ...] TO указание_роли [, ...]
    [ WITH { ADMIN | INHERIT | SET } { OPTION | TRUE | FALSE } ]
    [ GRANTED BY указание_роли ]

Здесь указание_роли:

    [ GROUP ] имя_роли
  | PUBLIC
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Описание

Команда `GRANT` имеет две основные разновидности: первая назначает права для доступа к объектам баз данных (таблицам, столбцам, представлениям, сторонним таблицам, последовательностям, базам данных, обёрткам сторонних данных, сторонним серверам, функциям, процедурам, процедурным языкам, большим объектам, параметрам конфигурации, схемам, табличным пространствам или типам), а вторая назначает одни роли членами других. Эти разновидности во многом похожи, но имеют достаточно отличий, чтобы рассматривать их отдельно.

GRANT для объектов баз данных

Эта разновидность команды `GRANT` даёт одной или нескольким ролям определённые права для доступа к объекту базы данных. Эти права добавляются к списку имеющихся, если роль уже наделена какими-то правами.

Ключевое слово `PUBLIC` означает, что права даются всем ролям, включая те, что могут быть созданы позже. `PUBLIC` можно воспринимать как неявно определённую группу, в которую входят все роли. Любая конкретная роль получит в сумме все права, данные непосредственно ей и ролям, членом которых она является, а также права, данные роли `PUBLIC`.

Если указано `WITH GRANT OPTION`, получатель права, в свою очередь, может давать его другим. Без этого указания распоряжаться своим правом он не сможет. Группе `PUBLIC` право передачи права дать нельзя.

Если указано `GRANTED BY`, заданным праводателем должен быть текущий пользователь. В настоящее время это предложение реализовано только ради совместимости со стандартом SQL.

Нет необходимости явно давать права для доступа к объекту его владельцу (обычно это пользователь, создавший объект), так как по умолчанию он имеет все права. (Однако владелец может лишить себя прав в целях безопасности.)

Право удалять объект или изменять его определение произвольным образом не считается назначаемым; оно неотъемлемо связано с владельцем, так что отозвать это право или дать его кому-то другому нельзя. (Однако похожий эффект можно получить, управляя членством в роли, владеющей объектом; см. ниже.) Владелец также неявно получает право распоряжения всеми правами для своего объекта.

Возможные права:

`SELECT`
`INSERT`
`UPDATE`
`DELETE`
`TRUNCATE`
`REFERENCES`
`TRIGGER`
`CREATE`
`CONNECT`
`TEMPORARY`
`EXECUTE`
`USAGE`
`SET`
`ALTER SYSTEM`
`MAINTAIN`

Определённые типы прав, описанные в Разделе 5.8.

`TEMP`

Альтернативное написание `TEMPORARY`.

`ALL PRIVILEGES`

Даёт целевой роли все права, применимые к данному объекту. Ключевое слово `PRIVILEGES` является необязательным в `Postgres Pro`, хотя в стандарте SQL оно требуется.

Синтаксис `FUNCTION` распространяется на обычные, агрегатные и оконные функции, но не на процедуры; для последних предназначен синтаксис `PROCEDURE`. Также можно использовать вариант с `ROUTINE`, охватывающий обычные, агрегатные и оконные функции, а также процедуры, вне зависимости от точного типа объекта.

Также можно дать роли некоторое право для всех объектов одного типа в одной или нескольких схемах. Эта функциональность в настоящее время поддерживается только для таблиц, последовательностей, функций и процедур. `ALL TABLES` распространяется и на представления со сторонними таблицами так же, как и вариант `GRANT` для конкретного объекта. `ALL FUNCTIONS` распространяется также на агрегатные и оконные функции, но не на процедуры, тоже подобно варианту `GRANT` для конкретного объекта. Чтобы охватить и процедуры, воспользуйтесь формой `ALL ROUTINES`.

GRANT для ролей

Эта разновидность команды `GRANT` включает роль в члены одной или нескольких других ролей, а также изменяет параметры членства `SET`, `INHERIT` и `ADMIN`. За более подробным описанием этих параметров обратитесь к Разделу 20.3. Членство в ролях играет важную роль, так как позволяет предоставить всем её членам права, данные роли, и потенциальную возможность изменять саму роль. Однако фактическое предоставление прав зависит от установленных параметров, связанных с их выдачей. Чтобы изменить параметры существующего членства, выдайте членство с обновлёнными значениями параметров.

Каждый из параметров, описанных ниже, может принимать значение `TRUE` или `FALSE`. Ключевое слово `OPTION` принимается в качестве синонима `TRUE`, то есть `WITH ADMIN OPTION` и `WITH ADMIN TRUE` являются синонимами. При изменении существующего членства отсутствие параметра приводит к сохранению текущего значения.

Получивший членство в роли с указанием `ADMIN` сможет, в свою очередь, включать в члены этой роли, а также исключать из неё другие роли. Без этого указания обычные пользователи не могут это делать. Считается, что роль не имеет права `WITH ADMIN OPTION` для самой себя. Суперпользователи баз данных могут включать или исключать любые роли из любых ролей. Значение по умолчанию — `FALSE`.

Параметр `INHERIT` управляет статусом наследования нового членства (за подробностями о наследовании обратитесь к Разделу 20.3). Если для этого параметра установлено значение `TRUE`, новый член наследует от выданной роли. Если установлено значение `FALSE`, новый член не наследует от выданной роли. Если при создании нового членства в роли этот параметр не указан, по умолчанию используется атрибут наследования добавляемого члена роли.

Если для `SET` указать `TRUE`, член роли сможет менять назначенную ему роль, используя команду `SET ROLE`. Если роль является опосредованным членом другой роли, то она сможет использовать `SET ROLE`, чтобы стать этой ролью только при условии, что существует цепочка членства в ролях, каждая из которых имеет атрибут `SET TRUE`. По умолчанию — `TRUE`.

Чтобы создать объект, владельцем которого будет другая роль, или назначить другой роли права владения существующим объектом, необходимо иметь право `SET ROLE` для этой роли, в противном случае команды `ALTER ... OWNER TO` или `CREATE DATABASE ... OWNER` выдадут ошибку. Однако пользователь, наследующий права роли, но не имеющий право `SET ROLE` для этой роли, может получить полный доступ к роли путём манипуляций с объектами, которыми владеет эта роль (например, переопределить существующую функцию с использованием «троянского кода»). Таким образом, если права роли должны наследоваться, но доступ к роли через `SET ROLE` предоставляться не должен, не следует выдавать ей права владения SQL-объектами.

С указанием `GRANTED BY` права выдаются от имени указанной роли. Это можно сделать, только если у пользователя есть права для этой роли. Роль, назначаемая праводателем, должна иметь право `ADMIN OPTION` для целевой роли (если только это не начальный суперпользователь). Если право выдаётся не от имени начального суперпользователя, а от другого праводателя, оно будет зависеть от праводателя с правом `ADMIN OPTION` для этой роли, так что при отзыве права `ADMIN OPTION` потребуется отозвать и зависимые права.

В отличие от прав, членство в ролях нельзя назначить группе `PUBLIC`. Заметьте также, что эта форма команды не принимает избыточное слово `GROUP` в *указании_роли*.

Примечания

Для лишения субъектов прав доступа применяется команда `REVOKE`.

Начиная с PostgreSQL версии 8.1, концепции пользователей и групп объединены в единую сущность, названную ролью. Таким образом, теперь нет необходимости добавлять ключевое слово `GROUP`, чтобы показать, что субъект является группой, а не пользователем. Слово `GROUP` всё ещё принимается этой командой, но оно лишено смысловой нагрузки.

Пользователь может выполнять `SELECT`, `INSERT` и подобные команды со столбцом таблицы, если он имеет такое право для данного столбца или для всей таблицы. Если назначить пользователю требуемое право на уровне таблицы, а затем отозвать его для одного из столбцов, это не даст эффекта, которого можно было бы ожидать: операция с правами на уровне столбцов не затронет право на уровне таблицы.

Если назначить право доступа к объекту (с помощью `GRANT`) попытается не владелец объекта, команда завершится ошибкой, если пользователь не имеет никаких прав для этого объекта. Если же пользователь имеет какие-то права, команда будет выполняться, но пользователь сможет давать другим только те права, которые даны ему с правом передачи. Формы `GRANT ALL PRIVILEGES` будут выдавать предупреждение, если у него вовсе нет таких прав, тогда как другие формы будут выдавать предупреждения, если пользователь не имеет прав распоряжаться именно правами, указанными в команде. (В принципе, эти утверждения применимы и к владельцу объекта, но ему разрешено распоряжаться всеми правами, поэтому такие ситуации невозможны.)

Следует отметить, что суперпользователи баз данных могут обращаться к любым объектам, вне зависимости от наличия каких-либо прав. Это сравнимо с привилегиями пользователя `root` в системе Unix. И так же, как `root`, роль суперпользователя следует использовать только когда это абсолютно необходимо.

Если суперпользователь решит выполнить команду `GRANT` или `REVOKE`, она будет выполнена, как если бы её выполнял владелец заданного объекта. В частности, права, назначенные такой командой, будут представлены как права, назначенные владельцем объекта. (Если так же установить членство в роли, оно будет представлено как назначенное начальным суперпользователем.)

`GRANT` и `REVOKE` также могут быть выполнены ролью, которая не является владельцем заданного объекта, но является членом роли-владельца, либо членом роли, имеющей права `WITH GRANT OPTION` для этого объекта. В этом случае права будут записаны как назначенные ролью, которая действительно владеет объектом, либо имеет право `WITH GRANT OPTION`. Например, если таблица `t1` принадлежит роли `g1`, членом которой является `u1`, то `u1` может дать права на использование `t1` роли `u2`, но эти права будут представлены, как назначенные непосредственно ролью `g1`. Отозвать эти права позже сможет любой член роли `g1`.

Если роль, выполняющая команду `GRANT`, получает требуемое право по нескольким путям членства, какая именно роль будет выбрана в качестве назначающей право, не определено. Если это важно, в таких случаях рекомендуется воспользоваться командой `SET ROLE` и переключиться на роль, которую хочется видеть в качестве выполняющей `GRANT`.

При назначении прав для доступа к таблице они автоматически не распространяются на последовательности, используемые этой таблицей, в том числе, на последовательности, связанные со столбцами `SERIAL`. Права доступа к последовательностям нужно назначать отдельно.

Подробнее о конкретных типах прав, а также о том, как просмотреть права, назначенные для объектов, рассказывается в Разделе 5.8.

Примеры

Следующая команда разрешает всем добавлять записи в таблицу `films`:

```
GRANT INSERT ON films TO PUBLIC;
```

Эта команда даёт пользователю `manuel` все права для представления `kinds`:

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

Учтите, что если её выполнит суперпользователь или владелец представления `kinds`, эта команда действительно даст субъекту все права, но если её выполнит обычный пользователь, субъект получит только те права, которые даны этому пользователю с правом передачи.

Включение в роль `admins` пользователя `joe`:

```
GRANT admins TO joe;
```

Совместимость

Согласно стандарту SQL, слово `PRIVILEGES` в указании `ALL PRIVILEGES` является обязательным. Стандарт SQL не позволяет назначать права сразу для нескольких объектов одной командой.

Postgres Pro позволяет владельцу объекта лишить себя своих обычных прав: например, владелец таблицы может разрешить себе только чтение таблицы, отозвав собственные права `INSERT`, `UPDATE`, `DELETE` и `TRUNCATE`. В стандарте SQL это невозможно. Это объясняется тем, что Postgres Pro воспринимает права владельца как назначенные им же себе; поэтому их можно и отозвать. В стандарте SQL права владельца даются ему предполагаемой сущностью «_SYSTEM». Так как владелец объекта отличается от «_SYSTEM», лишить себя этих прав он не может.

Согласно стандарту SQL, право с правом передачи можно дать субъекту `PUBLIC`; однако Postgres Pro может давать право с правом передачи только ролям.

Стандарт SQL допускает в предложении `GRANTED BY` только указание `CURRENT_USER` или `CURRENT_ROLE`. Другие варианты относятся к расширениям Postgres Pro.

В стандарте SQL право `USAGE` распространяется и на другие типы объектов: наборы символов, правила сортировки и преобразования.

В стандарте SQL право `USAGE` для последовательностей управляет использованием выражения `NEXT VALUE FOR`, которое равнозначно функции `nextval` в Postgres Pro. Права `SELECT` и `UPDATE` для последовательностей являются расширениями Postgres Pro. То, что право `USAGE` для последовательностей управляет использованием функции `currval`, так же относится к расширениям Postgres Pro (как и сама функция).

Права для баз данных, табличных пространств, схем, языков и параметров конфигурации относятся к расширениям Postgres Pro.

См. также

REVOKE, ALTER DEFAULT PRIVILEGES

IMPORT FOREIGN SCHEMA

IMPORT FOREIGN SCHEMA — импортировать определения таблиц со стороннего сервера

Синтаксис

```
IMPORT FOREIGN SCHEMA удалённая_схема
    [ { LIMIT TO | EXCEPT } ( имя_таблицы [, ...] ) ]
    FROM SERVER имя_сервера
    INTO локальная_схема
    [ OPTIONS ( параметр 'значение' [, ... ] ) ]
```

Описание

`IMPORT FOREIGN SCHEMA` создаёт сторонние таблицы, которые представляют таблицы, существующие на стороннем сервере. Новые сторонние таблицы будут принадлежать пользователю, выполняющему команду, и будут содержать корректные определения столбцов и параметры, соответствующие удалённым таблицам.

По умолчанию импортируются все таблицы и представления, существующие в определённой схеме на стороннем сервере. По желанию список таблиц можно ограничить некоторым подмножеством, или исключить из него конкретные таблицы. Новые сторонние таблицы создаются в целевой схеме, которая должна уже существовать.

Чтобы использовать `IMPORT FOREIGN SCHEMA`, необходимо иметь право `USAGE` для стороннего сервера, а также право `CREATE` в целевой схеме.

Параметры

удалённая_схема

Удалённая схема, из которой будут импортированы объекты. Что именно представляет собой удалённая схема, зависит от применяемой обёртки сторонних данных.

`LIMIT TO ( имя_таблицы [, ...] )`

Импортировать только сторонние таблицы с заданными именами. Другие таблицы, существующие в сторонней схеме, будут проигнорированы.

`EXCEPT ( имя_таблицы [, ...] )`

Исключить из импорта указанные сторонние таблицы. Данная команда импортирует все таблицы, существующие в сторонней схеме, за исключением перечисленных в этом предложении.

имя_сервера

Сторонний сервер, с которого импортируется схема.

локальная_схема

Схема, в которой будут созданы импортируемые сторонние таблицы.

`OPTIONS ( параметр 'значение' [, ...] )`

Параметры, которые должны применяться при импорте. Допустимые имена параметров и их значения зависят от обёртки сторонних данных.

Примеры

Импорт определений таблиц из удалённой схемы `foreign_films` на сервере `film_server` с созданием сторонних таблиц в локальной схеме `films`:

```
IMPORT FOREIGN SCHEMA foreign_films
    FROM SERVER film_server INTO films;
```

Та же операция, но импортируются только таблицы `actors` и `directors` (если они существуют):

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors, directors)
    FROM SERVER film_server INTO films;
```

Совместимость

Команда `IMPORT FOREIGN SCHEMA` соответствует стандарту SQL, за исключением параметра `OPTIONS`, являющегося расширением Postgres Pro.

См. также

CREATE FOREIGN TABLE, CREATE SERVER

INSERT

INSERT — добавить строки в таблицу

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
INSERT INTO имя_таблицы [ AS псевдоним ] [ ( имя_столбца [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    { DEFAULT VALUES | VALUES ( { выражение | DEFAULT } [, ...] ) [, ...] | query }
    [ ON CONFLICT [ объект_конфликта ] действие_при_конфликте ]
    [ RETURNING { * | выражение_результата [ [ AS ] имя_результата ] } [, ...] ]

Здесь допускается объект_конфликта:

    ( { имя_столбца_индекса | ( выражение_индекса ) } [ COLLATE правило_сортировки ]
 [ класс_операторов ] [, ...] ) [ WHERE предикат_индекса ]
    ON CONSTRAINT имя_ограничения

и действие_при_конфликте может быть следующим:

    DO NOTHING
    DO UPDATE SET { имя_столбца = { выражение | DEFAULT } |
                    ( имя_столбца [, ...] ) = [ ROW ] ( { выражение | DEFAULT }
 [, ...] ) |
                    ( имя_столбца [, ...] ) = ( вложенный_SELECT )
                  } [, ...]
              [ WHERE условие ]
```

Описание

`INSERT` добавляет строки в таблицу. Эта команда может добавить одну или несколько строк, сформированных выражениями значений, либо ноль или более строк, выданных дополнительным запросом.

Имена целевых столбцов могут перечисляться в любом порядке. Если список с именами столбцов отсутствует, по умолчанию целевыми столбцами становятся все столбцы заданной таблицы; либо первые *N* из них, если только *N* столбцов поступает от предложения `VALUES` или *запроса*. Значения, получаемые от предложения `VALUES` или *запроса*, связываются с явно или неявно определённым списком столбцов слева направо.

Все столбцы, не представленные в явном или неявном списке столбцов, получат значения по умолчанию, если для них заданы эти значения, либо NULL в противном случае.

Если выражение для любого столбца выдаёт другой тип данных, система попытается автоматически привести его к нужному.

Операция `INSERT` с таблицами без уникальных индексов не блокируется параллельно выполняемыми операциями. В таблицах с уникальными индексами эта операция может блокироваться, если в параллельных сеансах выполняются действия, которые блокируют или изменяют строки, совпадающие с вставляемыми значениями в уникальном индексе; подробнее см. Раздел 60.5. Предложение `ON CONFLICT` позволяет задать действие, заменяющее возникновение ошибки при нарушении ограничения уникальности или ограничения-исключения. (См. описание Предложение ON CONFLICT ниже.)

С необязательным предложением `RETURNING` команда `INSERT` вычислит и возвратит значения для каждой фактически добавленной строки (или изменённой, если применялось предложение `ON`

`CONFLICT DO UPDATE`). В основном это полезно для получения значений, присвоенных по умолчанию, например, последовательного номера записи. Однако в этом предложении можно задать любое выражение со столбцами таблицы. Список `RETURNING` имеет тот же синтаксис, что и список результатов `SELECT`. В результате будут возвращены те строки, которые были успешно вставлены или изменены. Например, если строка была заблокирована, но не изменена, из-за того, что *`условие`* в предложении `ON CONFLICT DO UPDATE ... WHERE` не удовлетворено, эта строка возвращена не будет.

Чтобы добавлять строки в таблицу, необходимо иметь право `INSERT` для неё. Если присутствует предложение `ON CONFLICT DO UPDATE`, также требуется иметь право `UPDATE` для этой таблицы.

Если указывается список столбцов, достаточно иметь право `INSERT` только для перечисленных столбцов. Аналогично, с предложением `ON CONFLICT DO UPDATE` достаточно иметь право `UPDATE` только для столбцов, которые будут изменены. Однако предложение `ON CONFLICT DO UPDATE` также требует наличия права `SELECT` для всех столбцов, значения которых считываются в выражениях `ON CONFLICT DO UPDATE` или в *`условии`*.

Для применения предложения `RETURNING` требуется право `SELECT` для всех столбцов, перечисленных в `RETURNING`. Если для добавления строк применяется *`запрос`*, для всех таблиц или столбцов, задействованных в этом запросе, разумеется, необходимо иметь право `SELECT`.

Параметры

Добавление

В этом разделе рассматриваются параметры, применяемые только при добавлении новых строк. Параметры, применяемые *исключительно* с предложением `ON CONFLICT`, описываются отдельно.

`запрос_WITH`

Предложение `WITH` позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе `INSERT`. Подробнее об этом см. Раздел 7.8 и SELECT.

Заданный *`запрос`* (оператор `SELECT`) также может содержать предложение `WITH`. В этом случае в *`запросе`* можно обращаться к обоим *`запросам_WITH`*, но второй будет иметь приоритет, так как он вложен ближе.

`имя_таблицы`

Имя существующей таблицы (возможно, дополненное схемой).

`псевдоним`

Альтернативное имя, заменяющее *`имя_таблицы`*. Когда указывается этот псевдоним, он полностью скрывает реальное имя таблицы. Это особенно полезно, когда в предложении `ON CONFLICT DO UPDATE` фигурирует таблица с именем `excluded`, так как без определения псевдонима это имя будет отдано специальной таблице, представляющей строку, предназначенную для добавления.

`имя_столбца`

Имя столбца в таблице *`имя_таблицы`*. Это имя столбца при необходимости может быть дополнено именем вложенного поля или индексом в массиве. (Когда данные вставляются только в некоторые поля столбца составного типа, в другие поля записывается NULL.) Обращаясь к столбцу в предложении `ON CONFLICT DO UPDATE`, включать имя таблицы в ссылку на целевой столбец не нужно. Например, запись `INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1` некорректна (это согласуется с общим поведением команды `UPDATE`).

`OVERRIDING SYSTEM VALUE`

Если указывается это предложение, то значения, предоставляемые для столбцов идентификации, переопределяют значения, выдаваемые последовательностью по умолчанию.

Для столбца идентификации, определённого со свойством `GENERATED ALWAYS`, считается ошибкой присваивание явного значения (кроме DEFAULT) без указания `OVERRIDING SYSTEM VALUE` или `OVERRIDING USER VALUE`. (Для столбца идентификации, определённого со свойством `GENERATED BY DEFAULT`, указание `OVERRIDING SYSTEM VALUE` соответствует обычному поведению и ни на что не влияет, но Postgres Pro допускает его как дополнение.)

`OVERRIDING USER VALUE`

Если указывается это предложение, то значения, предоставляемые для столбцов идентификации, игнорируются и вместо них применяются значения, выдаваемые последовательностью по умолчанию.

Это предложение полезно, например, при копировании значений между таблицами. Команда `INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1` скопирует из `tbl1` все столбцы, кроме столбцов идентификации в `tbl2`, а значения столбцов идентификации в `tbl2` будут сгенерированы последовательностями в `tbl2`.

`DEFAULT VALUES`

Все столбцы получат значения по умолчанию, как в случая явного указания `DEFAULT` для каждого столбца. (Предложение `OVERRIDING` в этой форме не допускается.)

`выражение`

Выражение или значение, которое будет присвоено соответствующему столбцу.

`DEFAULT`

Соответствующий столбец получит значение по умолчанию. Столбец идентификации получит новое значение, выданное связанной последовательностью. Для генерируемого столбца это указание допускается, но не меняет обычное поведение, то есть значение столбца вычисляется генерирующим выражением.

`query`

Запрос (оператор `SELECT`), который выдаст строки для добавления в таблицу. Его синтаксис описан в справке оператора SELECT.

`выражение_результата`

Выражение, которое будет вычисляться и возвращаться командой `INSERT` после добавления или изменения каждой строки. В этом выражении можно использовать имена любых столбцов таблицы *`имя_таблицы`*. Чтобы получить все столбцы, достаточно написать `*`.

`имя_результата`

Имя, назначаемое возвращаемому столбцу.

Предложение `ON CONFLICT`

Необязательное предложение `ON CONFLICT` задаёт действие, заменяющее возникновение ошибки при нарушении ограничения уникальности или ограничения-исключения. Для каждой отдельной строки, предложенной для добавления, добавление либо выполняется успешно, либо, если нарушается *решающее* ограничение или индекс, задаваемые как *`объект_конфликта`*, выполняется альтернативное *`действие_конфликта`*. Вариант `ON CONFLICT DO NOTHING` в качестве альтернативного действия просто отменяет добавление строки. Вариант `ON CONFLICT DO UPDATE` изменяет существующую строку, вызвавшую конфликт со строкой, предложенной для добавления.

Задаваемый *`объект_конфликта`* может *выбирать уникальный индекс*. Определение объекта, позволяющее выбрать индекс, включает один или несколько столбцов (их определяет *`имя_столбца_индекса`*) и/или *`выражение_индекса`* и необязательный *`предикат_индекса`*. Все уникальные индексы в таблице *`имя_таблицы`*, которые, без учёта порядка столбцов, содержат в точности столбцы/выражения, определяющие *`объект_конфликта`*, выбираются как решающие индексы. Если указывается *`предикат_индекса`*, он должен, в качестве дополнительного требования выбора, удовлетворять индексам. Заметьте, что это означает, что не частичный уникальный индекс (уникальный индекс без предиката) будет выбран (и будет использоваться в `ON CONFLICT`), если такой индекс удовлетворяет всем остальным критериям. Если попытка выбрать индекс оказывается неудачной, выдаётся ошибка.

`ON CONFLICT DO UPDATE` гарантирует атомарный результат команды `INSERT` или `UPDATE`; при отсутствии внешних ошибок гарантируется один из двух этих исходов, даже при большой параллельной активности. Эта операция также известна как *UPSERT* — «UPDATE или INSERT».

`объект_конфликта`

Определяет, для какого именно конфликта в `ON CONFLICT` будет предпринято альтернативное действие, устанавливая *решающие индексы*. Это указание позволяет осуществить *выбор уникального индекса* или явно задаёт имя ограничения. Для `ON CONFLICT DO NOTHING` *`объект_конфликта`* может не указываться; в этом случае игнорироваться будут все конфликты с любыми ограничениями (и уникальными индексами). Для `ON CONFLICT DO UPDATE` *`объект_конфликта`* *должен* указываться.

`действие_при_конфликте`

Параметр *`действие_при_конфликте`* задаёт альтернативное действие в случае конфликта. Это может быть либо `DO NOTHING` (не делать ничего), либо предложение `DO UPDATE` (произвести изменение), в котором указываются точные детали операции `UPDATE`, выполняемой в случае конфликта. Предложения `SET` и `WHERE` в `ON CONFLICT DO UPDATE` могут обращаться к существующей строке по имени таблицы (или псевдониму) или к строке, предлагаемой для добавления, используя специальную таблицу `excluded`. Для чтения столбцов `excluded` необходимо иметь право `SELECT` для соответствующих столбцов в целевой таблице.

Заметьте, что эффект действий всех триггеров уровня строк `BEFORE INSERT` отражается в значениях `excluded`, так как в результате этих действий строка может быть исключена из множества добавляемых.

`имя_столбца_индекса`

Имя столбца в таблице *`имя_таблицы`*. Используется для выбора решающих индексов. Задаётся в формате `CREATE INDEX`. Чтобы запрос выполнился, для столбца *`имя_столбца_индекса`* требуется право `SELECT`.

`выражение_индекса`

Подобно указанию *`имя_столбца_индекса`*, но применяется для выбора индекса по выражениям со столбцами таблицы *`имя_таблицы`*, фигурирующим в определениях индексов (не по простым столбцам). Задаётся в формате `CREATE INDEX`. Для всех столбцов, к которым обращается *`выражение_индекса`*, необходимо иметь право `SELECT`.

`правило_сортировки`

Когда задаётся, устанавливает, что соответствующие *`имя_столбца_индекса`* или *`выражение_индекса`* должны использовать определённый порядок сортировки, чтобы этот индекс мог быть выбран. Обычно это указание опускается, так как от правил сортировки чаще всего не зависит, произойдёт ли нарушение ограничений или нет. Задаётся в формате `CREATE INDEX`.

`класс_операторов`

Когда задаётся, устанавливает, что соответствующие *`имя_столбца_индекса`* или *`выражение_индекса`* должны использовать определённый класс, чтобы индекс мог быть выбран. Обычно это указание опускается, потому что семантика *равенства* часто всё равно одна и та же в разных классах операторов типа, или потому что достаточно рассчитывать на то, что заданные уникальные индексы имеют адекватное определение равенства. Задаётся в формате `CREATE INDEX`.

предикат_индекса

Используется для выбора частичных уникальных индексов. Выбраны могут быть любые индексы, удовлетворяющие предикату (при этом они могут не быть собственно частичными индексами). Задаётся в формате `CREATE INDEX`. Для всех столбцов, задействованных в *предикате_индекса*, требуется право `SELECT`.

имя_ограничения

Явно задаёт решающее *ограничение* по имени, что заменяет неявный выбор ограничения или индекса.

условие

Выражение, выдающее значение типа `boolean`. Изменены будут только те строки, для которых это выражение выдаст `true`, хотя при выборе действия `ON CONFLICT DO UPDATE` заблокируются все строки. Заметьте, что *условие* вычисляется в конце, после того как конфликт был признан претендующим на выполнение изменения.

Заметьте, что ограничения-исключения не могут быть решающими в `ON CONFLICT DO UPDATE`. Во всех случаях в качестве решающих поддерживаются только неоткладываемые (`NOT DEFERRABLE`) ограничения и уникальные индексы.

Команда `INSERT` с предложением `ON CONFLICT DO UPDATE` является «детерминированной». Это означает, что этой команде не разрешено воздействовать на любую существующую строку больше одного раза; в случае такой ситуации возникнет ошибка нарушения мощности множества. Строки, предлагаемые для добавления, не должны дублироваться с точки зрения атрибутов, ограничиваемых решающим индексом или ограничением.

Заметьте, что в настоящий момент не поддерживается ситуация, когда конструкция `ON CONFLICT DO UPDATE` команды `INSERT`, применяемой к секционированной таблице, изменяет ключ секционирования в конфликтующей строке так, что эта строка должна быть перенесена в новую секцию.

Подсказка

Часто предпочтительнее использовать неявный выбор уникального индекса вместо непосредственного указания ограничения в виде `ON CONFLICT ON CONSTRAINT` *имя_ограничения*. Выбор продолжит корректно работать, когда нижележащий индекс будет заменён другим более или менее равнозначным индексом методом наложения, например, с использованием `CREATE UNIQUE INDEX ... CONCURRENTLY` и последующим удалением заменяемого индекса.

Выводимая информация

В случае успешного завершения `INSERT` возвращает метку команды в виде

`INSERT` *oid число*

Здесь *число* представляет количество добавленных или изменённых строк. Поле *oid* всегда содержит 0 (раньше в нём выводился OID, присвоенный добавленной строке, когда *число* равнялось 1 и целевая таблица была создана с указанием `WITH OIDS`, а в противном случае — 0; теперь же создание таблицы с характеристикой `WITH OIDS` не поддерживается).

Если команда `INSERT` содержит предложение `RETURNING`, её результат будет похож на результат оператора `SELECT` (с теми же столбцами и значениями, что содержатся в списке `RETURNING`), полученный для строк, добавленных или изменённых этой командой.

Примечания

Если целевая таблица является секционированной, каждая строка перенаправляется в соответствующую секцию и вставляется в неё. Если целевая таблица является секцией и какая-либо из входных строк нарушает ограничение этой секции, происходит ошибка.

Вы также можете использовать команду `MERGE`, так как она позволяет объединить команды `INSERT`, `UPDATE` и `DELETE` в одном операторе. См. MERGE.

Примеры

Добавление одной строки в таблицу `films`:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

В этом примере столбец `len` опускается и, таким образом, получает значение по умолчанию:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

В этом примере для столбца с датой задаётся указание `DEFAULT`, а не явное значение:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

Добавление строки, полностью состоящей из значений по умолчанию:

```
INSERT INTO films DEFAULT VALUES;
```

Добавление нескольких строк с использованием многострочного синтаксиса `VALUES`:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

В этом примере в таблицу `films` вставляются некоторые строки из таблицы `tmp_films`, имеющей ту же структуру столбцов, что и `films`:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

Этот пример демонстрирует добавление данных в столбцы с типом массива:

```
-- Создание пустого поля 3x3 для игры в крестики-нолики
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- Указания индексов в предыдущей команде могут быть опущены
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

Добавление одной строки в таблицу `distributors` и получение последовательного номера, сгенерированного благодаря указанию `DEFAULT`:

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

Увеличение счётчика продаж для продавца, занимающегося компанией Acme Corporation, и сохранение всей изменённой строки вместе с текущим временем в таблице журнала:

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

Добавить дистрибьюторов или изменить существующие данные должным образом. Предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце `did`. Заметьте, что для обращения к значениям, изначально предлагаемым для добавления, используется специальная таблица `excluded`:

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
    ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

Добавить дистрибьютора или не делать ничего для строк, предложенных для добавления, если уже есть существующая исключающая строка (строка, содержащая конфликтующие значения в столбце или столбцах после срабатывания триггеров перед добавлением строки). В данном примере предполагается, что определён уникальный индекс, ограничивающий значения в столбце `did`:

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;
```

Добавить дистрибьюторов или изменить существующие данные должным образом. В данном примере предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце `did`. Предложение `WHERE` позволяет ограничить набор фактически изменяемых строк (однако любая существующая строка, не подлежащая изменению, всё же будет заблокирована):

```
-- Не менять данные существующих дистрибьюторов в зависимости от почтового индекса
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
    WHERE d.zipcode <> '21201';

-- Указать имя ограничения непосредственно в операторе (связанный индекс
-- применяется для принятия решения о выполнении действия DO NOTHING)
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

Добавить дистрибьютора, если возможно; в противном случае не делать ничего (`DO NOTHING`). В данном примере предполагается, что в таблице определён уникальный индекс, ограничивающий значения в столбце `did` по подмножеству строк, в котором логический столбец `is_active` содержит `true`:

```
-- Этот оператор может выбрать частичный уникальный индекс по "did"
-- с предикатом "WHERE is_active", а может и просто использовать
-- обычное ограничение уникальности по столбцу "did"
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
    ON CONFLICT (did) WHERE is_active DO NOTHING;
```

Совместимость

`INSERT` соответствует стандарту SQL, но предложение `RETURNING` относится к расширениям Postgres Pro, как и возможность применять `WITH` с `INSERT` и возможность задавать альтернативное действие с `ON CONFLICT`. Кроме того, ситуация, когда список столбцов опущен, но не все столбцы получают значения из предложения `VALUES` или *запроса*, стандартом не допускается. Если вы предпочитаете конструкции `ON CONFLICT` оператор, более соответствующий стандарту SQL, см. MERGE.

В стандарте SQL говорится, что предложение `OVERRIDING SYSTEM VALUE` может присутствовать, только если существует столбец идентификации, для которого всегда генерируется значение. Postgres Pro допускает это предложение в любом случае и игнорирует его в случае неприменимости.

Возможные ограничения предложения *запрос* описаны в справке SELECT.

LISTEN

LISTEN — ожидать уведомления

Синтаксис

```
LISTEN канал
```

Описание

`LISTEN` регистрирует текущий сеанс для получения уведомлений через канал с заданным именем (*канал*). Если текущий сеанс уже зарегистрирован и ожидает уведомлений через этот канал, ничего не происходит.

Когда вызывается команда `NOTIFY` *канал* (в текущем или другом сеансе, подключённом к той же базе данных), все сеансы, ожидающие уведомления через заданный канал, получают уведомление и каждый, в свою очередь, передаёт его подключённому клиентскому приложению.

Сеанс может отказаться от получения уведомлений через определённый канал с помощью команды `UNLISTEN`. Кроме того, подписка на любые уведомления автоматически отменяется при завершении сеанса.

Способ получения уведомлений клиентским приложением определяется программным интерфейсом Postgres Pro, который оно использует. Приложение, использующее библиотеку libpq, выполняет команду `LISTEN` как обычную команду SQL, а затем оно должно периодически вызывать функцию `PQnotifies`, чтобы проверить, не поступили ли новые уведомления. Другие интерфейсы, например libpgtcl, предоставляют более высокоуровневые методы для обработки событий уведомлений; на самом деле с libpgtcl разработчик приложения даже не должен непосредственно выполнять команды `LISTEN` и `UNLISTEN`. За дополнительными подробностями обратитесь к документации интерфейса, который вы используете.

Параметры

канал

Имя канала уведомлений (любой идентификатор).

Примечания

`LISTEN` начинает действовать при фиксировании транзакции. Если `LISTEN` или `UNLISTEN` выполняется в транзакции, которая затем откатывается, состояние подписки этого сеанса на уведомления не меняется.

Транзакция, в которой выполняется `LISTEN`, не может быть подготовлена для двухфазной фиксации.

Существует условие гонки в момент подписки на уведомления: если параллельно фиксируемые транзакции передают уведомления, какие именно получит только что подписавшийся сеанс? Ответ таков: этот сеанс получит все события, зафиксированные после момента фиксирования в нём транзакции, регистрирующей подписку. Но в этот момент состояние базы данных может несколько отличаться от того, что транзакция могла наблюдать в своих запросах. Таким образом, правильной будет следующая методика использования `LISTEN`: сначала выполните (и зафиксируйте) эту команду, затем в новой транзакции проанализируйте текущее состояние базы данных, как того требует логика приложения, и только после этого обрабатывайте уведомления, сообщающие о последующих изменениях состояния базы. Несколько первых уведомлений могут относиться к изменениям, уже наблюдавшимся при предварительном анализе базы, но обычно это не является проблемой.

В описании NOTIFY использование `LISTEN` и `NOTIFY` рассматривается более подробно.

Примеры

Демонстрация процедуры ожидания/получения уведомления в psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Совместимость

Оператор `LISTEN` отсутствует в стандарте SQL.

См. также

NOTIFY, UNLISTEN, max_notify_queue_pages

LOAD

LOAD — загрузить файл разделяемой библиотеки

Синтаксис

```
LOAD 'имя_файла'
```

Описание

Эта команда загружает файл разделяемой библиотеки в адресное пространство сервера Postgres Pro. Если указанный файл был загружен ранее, эта команда не делает ничего. Файлы библиотек, содержащие функции на C, загружаются автоматически при первом вызове любой из этих функций. Поэтому явно выполнять `LOAD` обычно требуется только для загрузки библиотек, которые изменяют поведение сервера, внедряя свои обработчики, а не предоставляют некоторый набор функций.

Имя файла библиотеки обычно задаётся собственно именем файла, а полный путь определяется при просмотре пути поиска библиотек сервера (задаваемого в dynamic_library_path). Также в качестве имени может быть передан непосредственно полный путь. В любом случае расширение имени файла, стандартное для файлов разделяемых библиотек на данной платформе, можно опустить. Дополнительную информацию по этой теме можно найти в Подразделе 36.10.1.

Обычные пользователи (не суперпользователи) могут использовать `LOAD` только для загрузки файлов библиотек, расположенных в `$libdir/plugins/` — заданное *`имя_файла`* должно начинаться именно с этой строки. (Ответственность за то, чтобы в этом каталоге находились только «безопасные» библиотеки, лежит на администраторе баз данных.)

Совместимость

`LOAD` является расширением Postgres Pro.

См. также

CREATE FUNCTION

LOCK

LOCK — заблокировать таблицу

Синтаксис

```
LOCK [ TABLE ] [ ONLY ] имя [ * ] [, ...] [ IN режим_блокировки MODE ] [ NOWAIT ]

Где режим_блокировки может быть следующим:

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Описание

`LOCK TABLE` получает блокировку на уровне таблицы, при необходимости ожидая освобождения таблицы от других конфликтующих блокировок. Если указано `NOWAIT`, `LOCK TABLE` не ждёт, пока таблица освободится: если блокировку нельзя получить немедленно, команда прерывается и выдаётся ошибка. Как только блокировка получена, она удерживается до завершения текущей транзакции. (Команды `UNLOCK TABLE` не существует; блокировки всегда освобождаются в конце транзакции.)

Когда блокируется представление, блокировка с тем же режимом рекурсивно распределяется на все отношения, фигурирующие в определяющем представление запросе.

Запрашивая автоматические блокировки для команд, работающих с таблицами, Postgres Pro всегда выбирает наименее ограничивающий режим блокировки из возможных. Оператор `LOCK TABLE` предназначен для случаев, когда требуется более сильная блокировка. Например, предположим, что приложение выполняет транзакцию на уровне изоляции `READ COMMITTED` и оно должно получать неизменные данные на протяжении всей транзакции. Для достижения этой цели можно получить для таблицы блокировку в режиме `SHARE`, прежде чем обращаться к ней. В результате параллельные изменения данных будут исключены и при последующих чтениях будет получено стабильное представление зафиксированных данных, так как режим блокировки `SHARE` конфликтует с блокировкой `ROW EXCLUSIVE`, запрашиваемой при записи, а `LOCK TABLE` *`имя`* `IN SHARE MODE` будет ждать, пока параллельные транзакции с блокировкой `ROW EXCLUSIVE` не будут зафиксированы или отменены. Таким образом, в момент получения такой блокировки не останется ни одной открытой незафиксированной операции записи; кроме того, никто не сможет записывать в таблицу, пока блокировка не будет снята.

Чтобы получить похожий эффект в транзакции на уровне изоляции `REPEATABLE READ` или `SERIALIZABLE`, необходимо выполнить оператор `LOCK TABLE` перед первым `SELECT` или оператором, изменяющим данные. Представление данных для транзакции уровня `REPEATABLE READ` или `SERIALIZABLE` будет заморожено в момент, когда начнёт выполняться этот запрос. Если команда `LOCK TABLE` выполняется в транзакции позже, она так же исключает параллельную запись, но не даёт гарантии, что транзакция будет читать последние зафиксированные данные.

Если в транзакции такого рода требуется изменять данные в таблице, для неё следует использовать режим блокировки `SHARE ROW EXCLUSIVE` вместо `SHARE`. Этот режим гарантирует, что в один момент времени будет выполняться только одна транзакция такого типа. Без этого ограничения возможна взаимоблокировка: две транзакции могут одновременно получить блокировки `SHARE`, после чего они не смогут получить блокировку `ROW EXCLUSIVE`, чтобы собственно выполнить изменения. (Заметьте, что собственные блокировки транзакции никогда не конфликтуют, так что транзакция может получить блокировку `ROW EXCLUSIVE`, когда она владеет блокировкой `SHARE` — но не тогда, когда блокировку `SHARE` удерживает другая транзакция.) Чтобы не допустить взаимоблокировок, убедитесь, что все транзакции запрашивают блокировки одних объектов в одинаковом порядке, и если для одного объекта запрашиваются блокировки в разных режимах, транзакции всегда должны запрашивать самую строгую блокировку.

Дополнительно о режимах и стратегиях блокировки можно узнать в Разделе 13.3.

Параметры

имя

Имя (возможно, дополненное схемой) существующей таблицы, для которой запрашивается блокировка. Если перед именем таблицы указано `ONLY`, блокируется только заданная таблица. Без `ONLY` блокируется и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание `*`, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы.

Команда `LOCK TABLE a, b;` равнозначна последовательности `LOCK TABLE a; LOCK TABLE b;`. Таблицы блокируются по одной в порядке, заданном в команде `LOCK TABLE`.

режим_блокировки

Режим блокировки определяет, с какой блокировкой будет конфликтовать данная. Режимы блокировок описаны в Разделе 13.3.

Если режим блокировки не указан, применяется самый строгий режим, `ACCESS EXCLUSIVE`.

`NOWAIT`

Указывает, что `LOCK TABLE` не должна ожидать освобождения конфликтующих блокировок: если запрошенная блокировка не может быть получена немедленно, транзакция прерывается.

Примечания

Чтобы заблокировать таблицу, используя *режимы_блокировки*, нужно иметь соответствующие права. Если у пользователя есть право `MAINTAIN`, `UPDATE`, `DELETE` или `TRUNCATE` для таблицы, может использоваться любой *режим_блокировки*. Если у пользователя есть право `INSERT` для таблицы, допускается `ROW EXCLUSIVE MODE` (или менее конфликтующий режим, что описано в Разделе 13.3). Если у пользователя есть право `SELECT` для таблицы, допускается `ACCESS SHARE MODE`.

Пользователь, выполняющий операцию блокировки представления, должен иметь соответствующее право для этого представления. Кроме того, владелец представления должен иметь сопутствующие права в нижележащих базовых отношениях, хотя пользователь, устанавливающий блокировку, может этих прав не иметь. Однако если для свойства представления `security_invoker` установлено значение `true` (см. `CREATE VIEW`), соответствующие права для нижележащих базовых отношений должен иметь пользователь, устанавливающий блокировку, а не владелец представления.

Вне блока транзакции команда `LOCK TABLE` бесполезна: блокировка сохранится только до завершения операции. Поэтому Postgres Pro выдаёт ошибку при попытке применить `LOCK` не в блоке транзакции. Чтобы определить блок транзакции, используйте `BEGIN` и `COMMIT` (или `ROLLBACK`).

`LOCK TABLE` может устанавливать только блокировки на уровне таблицы, так что все имена режимов, включающие слово `ROW` (строка), не совсем корректны. Следует воспринимать их так, что в этих режимах пользователь намеревается получать в заблокированной таблице блокировки уровня строк. Также учтите, что в режиме `ROW EXCLUSIVE` устанавливается разделяемая блокировка таблицы. Заметьте, что применительно к `LOCK TABLE` все режимы блокировки действуют одинаково, отличаются только правила, определяющие, какой режим с каким конфликтует. Чтобы узнать, как получить блокировку именно на уровне строк, обратитесь к Подразделу 13.3.2 и разделу Предложение блокировки в описании SELECT.

Примеры

Получение блокировки `SHARE` для первичного ключа таблицы при добавлении записи в подчинённую таблицу:

```
BEGIN WORK;
```

```
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Если запись не будет возвращена, произойдёт откат транзакции
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Установление блокировки `SHARE ROW EXCLUSIVE` в таблице первичного ключа перед выполнением операции удаления:

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Совместимость

Команда `LOCK TABLE` отсутствует в стандарте SQL, в нём уровни изоляции транзакции определяются командой `SET TRANSACTION`. Postgres Pro поддерживает и этот вариант; подробнее это описано в SET TRANSACTION.

За исключением `ACCESS SHARE`, `ACCESS EXCLUSIVE` и `SHARE UPDATE EXCLUSIVE`, режимы блокировки в Postgres Pro и синтаксис `LOCK TABLE` совместимы с теми, что представлены в СУБД Oracle.

MERGE

MERGE — добавить, изменить или удалить строки таблицы по условию

Синтаксис

```
[ WITH запрос_WITH [, ...] ]
MERGE INTO [ ONLY ] имя_целевой_таблицы [ * ] [ [ AS ] целевой_псевдоним ]
    USING источник_данных ON условие_соединения
    предложение_when [...]
    [ RETURNING { * | выражение_результата [ [ AS ] имя_результата ] } [, ...] ]

Здесь источник_данных:

    { [ ONLY ] имя_исходной_таблицы [ * ] | ( исходный_запрос ) }
 [ [ AS ] исходный_псевдоним ]

и предложение_when:

    { WHEN MATCHED [ AND условие ] THEN { изменение_при_объединении
 | удаление_при_объединении | DO NOTHING } |
      WHEN NOT MATCHED BY SOURCE [ AND условие ] THEN { изменение_при_объединении
 | удаление_при_объединении | DO NOTHING } |
      WHEN NOT MATCHED [ BY TARGET ] [ AND условие ] THEN { добавление_при_объединении
 | DO NOTHING } }

и добавление_при_объединении:

    INSERT [( имя_столбца [, ...] )]
        [ OVERRIDING { SYSTEM | USER } VALUE ]
        { VALUES ( { выражение | DEFAULT } [, ...] ) | DEFAULT VALUES }

и изменение_при_объединении:

    UPDATE SET { имя_столбца = { выражение | DEFAULT } |
                 ( имя_столбца [, ...] ) = [ ROW ] ( { выражение | DEFAULT } [, ...] )
 |
                 ( имя_столбца [, ...] ) = ( вложенный_SELECT )
               } [, ...]

и удаление_при_объединении:

    DELETE
```

Описание

Операция `MERGE` выполняет действия, которые меняют строки в целевой таблице с *`именем_целевой_таблицы`*, используя *`источник_данных`*. `MERGE` — это один SQL-оператор, который по условию выполняет со строками действия `INSERT`, `UPDATE` или `DELETE`; сделать то же самое без `MERGE` можно, только используя несколько операторов процедурного языка.

Сначала команда `MERGE` соединяет *`источник_данных`* с целевой таблицей, формируя ноль или более строк-кандидатов на изменение. Для каждой строки-кандидата устанавливается неизменяемый позже статус `MATCHED` (совпадает), `NOT MATCHED BY SOURCE` (не совпадает по источнику) или `NOT MATCHED [BY TARGET]`) (не совпадает по цели), после чего вычисляются условия `WHEN` в заданном порядке. Для каждой отдельной строки будет выполняться действие первого же предложения, условие которого выдаст true. При этом для каждой строки-кандидата может быть выполнено действие не более чем одного предложения `WHEN`.

Действия операции `MERGE` имеют тот же эффект, что и обычные одноимённые команды `UPDATE`, `INSERT` или `DELETE`. Синтаксис этих команд в `MERGE` отличается, в частности, отсутствием предложения `WHERE` и имени таблицы. Действия этих команд выполняются с целевой таблицей, хотя посредством триггеров могут быть изменены и другие таблицы.

С указанием `DO NOTHING` исходная строка пропускается. Поскольку применимость действий оценивается в заданном порядке, используя `DO NOTHING`, удобно пропускать исходные строки, не представляющие интерес, чтобы затем более детально обрабатывать остальные.

Предложение `RETURNING` указывает, что команда `MERGE` должна вычислить и вернуть значения для каждой вставленной, изменённой или удалённой строки. Вычислить в нём можно любое выражение со столбцами исходной или целевой таблицы, а также функцию `merge_action()`. При выполнении команды `INSERT` или `UPDATE` используются новые значения столбцов целевой таблицы. При выполнении команды `DELETE` используются старые значения столбцов целевой таблицы. Список `RETURNING` имеет тот же синтаксис, что и список результатов `SELECT`.

Для команды `MERGE` не предусмотрено отдельное право. Если пользователь указывает в ней действие `UPDATE`, у него должно быть право `UPDATE` для столбцов целевой таблицы, на которые ссылается предложение `SET`. Когда указывается действие `INSERT` или `DELETE`, у пользователя должно быть соответствующее право для целевой таблицы. Если пользователь указывает действие `DO NOTHING`, у него должно быть право `SELECT` хотя бы для одного столбца целевой таблицы. Кроме того, необходимо иметь право `SELECT` для любых столбцов *`источника_данных`* и целевой таблицы, которые фигурируют в `condition` (в том числе `join_condition`) или `expression`. Права проверяются один раз в начале выполнения оператора, вне зависимости от того, будут ли выполняться конкретные предложения `WHEN`.

Оператор `MERGE` не поддерживается для целевых таблиц, являющихся материализованными представлениями, сторонними таблицами, или если для них заданы какие-либо правила.

Параметры

`запрос_WITH`

Предложение `WITH` позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе `MERGE`. За подробностями обратитесь к Разделу 7.8 и SELECT. Обратите внимание, что предложение `WITH RECURSIVE` для команды `MERGE` не поддерживается.

`имя_целевой_таблицы`

Имя (возможно, дополненное схемой) целевой таблицы или представления, которые принимают результат объединения. Если перед именем таблицы добавлено `ONLY`, соответствующие строки изменяются или удаляются только в указанной таблице. Без `ONLY` соответствующие строки также изменяются или удаляются во всех таблицах, унаследованных от указанной таблицы. При желании, после имени таблицы можно указать `*`, чтобы явно обозначить, что операция затрагивает все дочерние таблицы. Ключевое слово `ONLY` и параметр `*` не влияют на действия `INSERT`, добавляющие строки только в указанную таблицу.

Если в *`имени_целевой_таблицы`* указано имя представления, оно должно быть либо автоматически изменено без триггеров `INSTEAD OF`, либо иметь триггеры `INSTEAD OF` для каждого действия (`INSERT`, `UPDATE` и `DELETE`), указанного в предложении `WHEN`. Представления, для которых заданы правила, не поддерживаются.

`целевой_псевдоним`

Альтернативное имя целевой таблицы. Когда это имя задаётся, настоящее имя таблицы полностью скрывается. Например, в запросе `MERGE INTO foo AS f` остальные компоненты оператора `MERGE` должны обращаться к целевой таблице по имени `f`, а не `foo`.

`имя_исходной_таблицы`

Имя (возможно, дополненное схемой) исходной таблицы, представления или переходной таблицы. Если перед именем таблицы добавлено `ONLY`, соответствующие строки берутся только из указанной таблицы. Без `ONLY` строки также берутся из всех таблиц, унаследованных от указанной. При желании, после имени таблицы можно указать `*`, чтобы явно обозначить, что операция затрагивает все дочерние таблицы.

`исходный_запрос`

Запрос (оператор `SELECT` или оператор `VALUES`), предоставляющий строки для объединения в целевой таблице. За информацией о синтаксисе обратитесь к описанию SELECT и VALUES.

`исходный_псевдоним`

Альтернативное имя для источника данных. Когда задаётся этот псевдоним, он полностью скрывает настоящее имя таблицы или тот факт, что это результат запроса.

`условие_соединения`

Задаваемое *`условие_соединения`* представляет собой выражение, выдающее значение типа `boolean` (как в предложении `WHERE`), которое определяет, какие строки в *`источнике_данных`* соответствуют строкам в целевой таблице.

Предупреждение

В *`условии_соединения`* должны фигурировать только столбцы целевой таблицы, по которым её строки сопоставляются со строками *`источника_данных`*. Подвыражения *`условия_соединения`*, ссылающиеся только на столбцы целевой таблицы, могут влиять на выполняемое действие, часто неожиданным образом.

Если указаны оба предложения `WHEN NOT MATCHED BY SOURCE` и `WHEN NOT MATCHED [BY TARGET]`, команда `MERGE` сделает полное соединение (`FULL JOIN`) *`источника_данных`* с целевой таблицей. Для корректной работы необходимо, чтобы хотя бы в одном подвыражении *`условия_соединения`* использовался оператор с поддержкой соединений по хешу, или чтобы во всех подвыражениях использовались операторы с поддержкой соединений слиянием.

`предложение_when`

В команде `MERGE` должно быть минимум одно предложение `WHEN`.

В предложении `WHEN` можно задать `WHEN MATCHED`, `WHEN NOT MATCHED BY SOURCE` или `WHEN NOT MATCHED [BY TARGET]`. Обратите внимание, что стандарт SQL определяет только `WHEN MATCHED` и `WHEN NOT MATCHED` (отсутствие соответствующей целевой строки). `WHEN NOT MATCHED BY SOURCE` является расширением стандарта SQL как способ совместно использовать `BY TARGET` и `WHEN NOT MATCHED` для более точного запроса.

Если в предложении `WHEN` указано `WHEN MATCHED` и строка-кандидат на изменение представляет собой строку из *`источника_данных`*, совпадающую со строкой целевой таблицы, то предложение `WHEN` выполняется, когда *`условие`* отсутствует или оценивается как `true`.

И наоборот, если в предложении `WHEN` указано `WHEN NOT MATCHED BY SOURCE` и строка-кандидат на изменение является строкой целевой таблицы, которая не соответствует строке в *`источнике_данных`*, предложение `WHEN` выполняется, когда *`условие`* отсутствует или оценивается как `true`.

Если в предложении `WHEN` указано `WHEN NOT MATCHED [BY TARGET]` и строка-кандидат на изменение является строкой в *`источнике_данных`*, которая не соответствует строке целевой таблицы, предложение `WHEN` выполняется, когда *`условие`* отсутствует или оценивается как `true`.

условие

Выражение, выдающее значение типа `boolean`. Если это выражение для предложения `WHEN` выдаёт `true`, для данной строки выполняется действие этого предложения.

Условие в предложении `WHEN MATCHED` может ссылаться на столбцы как исходного, так и целевого отношения. Условие в предложении `WHEN NOT MATCHED BY SOURCE` может ссылаться только на столбцы целевого отношения, поскольку соответствующей исходной строки нет по определению. Условие в предложении `WHEN NOT MATCHED [BY TARGET]` может ссылаться только на столбцы исходного отношения, поскольку соответствующей целевой строки нет по определению. В целевой таблице доступны только системные атрибуты.

добавление_при_объединении

Указание действия `INSERT`, добавляющего одну строку в целевую таблицу. Имена целевых столбцов могут перечисляться в любом порядке. Если список имён столбцов не задан вовсе, по умолчанию используются все столбцы таблицы в порядке объявления.

Все столбцы, не представленные в явном или неявном списке столбцов, получат значения по умолчанию, если для них заданы эти значения, либо NULL в противном случае.

Если целевая таблица является секционированной, каждая строка направляется в соответствующую секцию и добавляется в неё. Если целевая таблица является секцией и какая-либо входная строка нарушит ограничение секции, произойдёт ошибка.

Имена столбцов нельзя указывать более одного раза. Действия `INSERT` не могут содержать вложенные запросы `SELECT`.

Предложение `VALUES` может указываться только один раз. Ссылаться в нём можно только на столбцы исходного отношения, так как соответствующих целевых строк нет по определению.

изменение_при_объединении

Указание действия `UPDATE`, изменяющего текущую строку целевой таблицы. Имена столбцов нельзя указывать более одного раза.

Задавать имя таблицы и предложение `WHERE` здесь нельзя.

удаление_при_объединении

Указание действия `DELETE`, удаляющего текущую строку целевой таблицы. Задавать имя таблицы или какие-либо другие предложения, как в обычной команде DELETE, здесь нельзя.

имя_столбца

Имя столбца целевой таблицы. При необходимости имя столбца можно дополнить именем поля или индексом массива. (При добавлении данных лишь в некоторые поля составного типа другие поля будут содержать NULL.) Имя таблицы в указание целевого столбца добавлять не нужно.

`OVERRIDING SYSTEM VALUE`

Без этого предложения не допускается задание явного значения (отличного от `DEFAULT`) для столбца идентификации, определённого с характеристикой `GENERATED ALWAYS`. Данное предложение перекрывает это ограничение.

`OVERRIDING USER VALUE`

Если указывается это предложение, то значения, заданные для столбцов идентификации, которые определены с характеристикой `GENERATED BY DEFAULT`, игнорируются и вместо них применяются значения, выдаваемые последовательностями по умолчанию.

`DEFAULT VALUES`

Все столбцы получают значения по умолчанию. (Предложение `OVERRIDING` в этой форме не допускается.)

выражение

Выражение, результат которого присваивается столбцу. В выражениях предложений `WHEN MATCHED` могут использоваться значения из исходной строки целевой таблицы и значения из строки *источника_данных*. В выражениях предложений `WHEN NOT MATCHED BY SOURCE` могут использоваться значения только из исходной строки в целевой таблице. В выражениях предложений `WHEN NOT MATCHED [BY TARGET]` могут использоваться значения только из *источника_данных*.

`DEFAULT`

Присвоить столбцу значение по умолчанию (или `NULL`, если выражение по умолчанию для столбца не определено).

вложенный_SELECT

Подзапрос `SELECT`, выдающий столько выходных столбцов, сколько перечислено в предшествующем ему списке столбцов в скобках. При выполнении этого подзапроса должна быть получена максимум одна строка. Если он выдаёт одну строку, значения столбцов в нём присваиваются целевым столбцам; если же он не возвращает строку, целевым столбцам присваивается NULL. При использовании предложения `WHEN MATCHED` подзапрос может обращаться к значениям исходной строки в целевой таблице, а также значениям строки *источника_данных*. При использовании предложения `WHEN NOT MATCHED BY SOURCE` подзапрос может обращаться только к значениям исходной строки в целевой таблице.

выражение_результата

Выражение, вычисляемое и возвращаемое командой `MERGE` после изменения каждой строки (добавления, изменения или удаления). В этом выражении можно использовать имена любых столбцов исходной и целевой таблицы или функцию `merge_action()` для получения дополнительной информации о выполняемых действиях.

С указанием `*` возвращаются все столбцы исходной таблицы, а после них — все столбцы целевой. Обычно это приводит к большому количеству дубликатов, поскольку у исходной и целевой таблицы часто одинаковые столбцы. Этого можно избежать, указав при использовании `*` имена или псевдонимы исходной или целевой таблицы.

имя_результата

Имя, назначаемое возвращаемому столбцу.

Выводимая информация

При успешном выполнении команда `MERGE` возвращает метку команды в виде

`MERGE` *общее_число*

Здесь *общее_число* — суммарное количество изменённых строк (добавленных, изменённых или удалённых). Если *общее_число* равно 0, ни одна строка не была изменена.

Если команда `MERGE` содержит предложение `RETURNING`, её результат будет похож на результат `SELECT` (с теми же столбцами и значениями, что содержатся в списке `RETURNING`) для строк, добавленных, изменённых или удалённых этой командой.

Примечания

В ходе выполнения `MERGE` производятся следующие действия.

1. Вызываются все триггеры `BEFORE STATEMENT` для всех указанных действий, независимо от того, совпадают ли их предложения `WHEN`.
2. Выполняется соединение исходной таблицы с целевой. Полученный в результате запрос оптимизируется как обычно и выдаёт набор строк-кандидатов на изменение. Для каждой строки-кандидата на изменение:

   a. Для каждой строки определяется состояние: `MATCHED` (совпадает), `NOT MATCHED BY SOURCE` (не совпадает по источнику) или `NOT MATCHED [BY TARGET]` (не совпадает по цели).
   b. Проверяется каждое условие `WHEN` в заданном порядке, пока какое-либо не выдаст значение true.
   c. Если условие оценивается как `true`, происходит следующее:
      i. Вызываются все триггеры `BEFORE ROW`, соответствующие типу события выполняемого действия.
      ii. Выполняется указанное действие, при этом вызываются ограничения-проверки для целевой таблицы.
      iii. Вызываются все триггеры `AFTER ROW`, соответствующие типу события выполняемого действия.

      Если целевое отношение является представлением с триггерами `INSTEAD OF ROW`, соответствующими типу события выполняемого действия, то триггеры выполняют эти действия.
3. Выполняются все триггеры `AFTER STATEMENT` для всех указанных действий, независимо от того, выполнялись ли эти действия фактически. Это похоже на поведение оператора `UPDATE`, когда он не меняет ни одной строки.

То есть триггеры уровня оператора для некоторого события (скажем, `INSERT`) будут вызываться всегда, когда *указывается* действие такого типа. Триггеры уровня строк, напротив, вызываются только для определённого действия, которое *выполняется*. Таким образом, при выполнении `MERGE` могут вызываться триггеры уровня оператора как для `UPDATE`, так и для `INSERT`, даже если на уровне строк вызывались только триггеры `UPDATE`.

Следует позаботиться о том, чтобы для каждой целевой строки в результате соединения создавалось не более одной строки-кандидата на изменение. Другими словами, целевая строка не должна соединяться с более чем одной строкой источника данных. Если это не так, только одна из строк-кандидатов будет применяться для изменения целевой строки; последующие попытки изменить эту строку вызовут ошибку. Ошибка также может произойти, когда триггеры строк вносят изменения в целевую таблицу, а команда `MERGE` впоследствии воздействует на уже изменённые строки. Если повторится действие `INSERT`, это вызовет нарушение уникальности, а повторение `UPDATE` или `DELETE` вызовет ошибку «Нарушение количества»; последнее требуется стандартом SQL. Такое поведение отличается от поведения соединений в `UPDATE` и `DELETE`, традиционного для PostgreSQL, когда вторая и последующие попытки изменить одну и ту же строку просто игнорируются.

Если в предложении `WHEN` отсутствует дополнительное условие `AND`, оно становится последним достижимым предложением этого рода (`MATCHED`, `NOT MATCHED BY SOURCE` или `NOT MATCHED [BY TARGET]`). Если в команде встретится последующее предложение `WHEN` такого рода, оно гарантированно будет недостижимым, и это вызовет ошибку. В случае отсутствия последнего достижимого предложения любого рода возможна ситуация, когда для строки-кандидата на изменение не будет предпринято никаких действий.

Порядок, в котором строки выдаются из источника данных, по умолчанию не определён. Если необходим определённый порядок, например для предотвращения взаимоблокировок между параллельными транзакциями, его можно задать в *`исходном_запросе`*.

Когда `MERGE` выполняется одновременно с другими командами, изменяющими целевую таблицу, применяются обычные правила изоляции транзакций; поведение на каждом уровне изоляции описано в Разделе 13.2. В качестве альтернативы можно рассмотреть использование оператора `INSERT ... ON CONFLICT`, который предусматривает возможность выполнения команды `UPDATE`, если параллельно выполняется команда `INSERT`. Эти два типа операторов имеют ряд различий и особых ограничений, они не являются взаимозаменяемыми.

Примеры

Корректировка клиентских счетов (`customer_accounts`) с учётом новых транзакций (`recent_transactions`).

```
MERGE INTO customer_account ca
USING recent_transactions t
ON t.customer_id = ca.customer_id
WHEN MATCHED THEN
  UPDATE SET balance = balance + transaction_value
WHEN NOT MATCHED THEN
  INSERT (customer_id, balance)
  VALUES (t.customer_id, t.transaction_value);
```

Заметьте, что это полностью равнозначно следующему оператору, потому что статус `MATCHED` не меняется во время выполнения.

```
MERGE INTO customer_account ca
USING (SELECT customer_id, transaction_value FROM recent_transactions) AS t
ON t.customer_id = ca.customer_id
WHEN MATCHED THEN
  UPDATE SET balance = balance + transaction_value
WHEN NOT MATCHED THEN
  INSERT (customer_id, balance)
  VALUES (t.customer_id, t.transaction_value);
```

Попытка добавить новый продукт вместе с количеством. Если такая запись уже существует, увеличивается количество данного продукта в существующей записи. Позиции с нулевым количеством удаляются. Возвращается описание всех внесённых изменений.

```
MERGE INTO wines w
USING wine_stock_changes s
ON s.winename = w.winename
WHEN NOT MATCHED AND s.stock_delta > 0 THEN
  INSERT VALUES(s.winename, s.stock_delta)
WHEN MATCHED AND w.stock + s.stock_delta > 0 THEN
  UPDATE SET stock = w.stock + s.stock_delta
WHEN MATCHED THEN
  DELETE
RETURNING merge_action(), w.*;
```

Таблица `wine_stock_changes` может быть, например, временной таблицей, недавно загруженной в базу данных.

Изменяется `wines` в соответствии с новым списком, добавляются строки для нового товара, обновляются позиции изменённых товаров и удаляются все виды вин, отсутствующие в новом списке.

```
MERGE INTO wines w
USING new_wine_list s
ON s.winename = w.winename
WHEN NOT MATCHED BY TARGET THEN
  INSERT VALUES(s.winename, s.stock)
WHEN MATCHED AND w.stock != s.stock THEN
  UPDATE SET stock = s.stock
WHEN NOT MATCHED BY SOURCE THEN
  DELETE;
```

Совместимость

Эта команда соответствует стандарту SQL.

Предложение `WITH`, дополнения `BY SOURCE` и `BY TARGET` к действиям `WHEN NOT MATCHED`, `DO NOTHING`, а также предложение `RETURNING` являются расширениями стандарта SQL.

MOVE

MOVE — переместить курсор

Синтаксис

```
MOVE [ направление ] [ FROM | IN ] имя_курсора

Здесь допускается направление:

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE число
    RELATIVE число
    число
    ALL
    FORWARD
    FORWARD число
    FORWARD ALL
    BACKWARD
    BACKWARD число
    BACKWARD ALL
```

Описание

`MOVE` перемещает курсор, не получая данные. Команда `MOVE` работает точно так же, как `FETCH`, но она не возвращает данные строк, а только перемещает курсор.

Команда `MOVE` поддерживает те же параметры, что и `FETCH`; за подробным описанием её синтаксиса и использования обратитесь к FETCH.

Выводимая информация

В случае успешного завершения, `MOVE` возвращает метку команды в виде

```
MOVE число
```

Здесь *число* показывает количество строк, которое бы выдала команда `FETCH` с такими же параметрами (оно может быть нулевым).

Примеры

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Пропустить первые 5 строк:
MOVE FORWARD 5 IN liahona;
MOVE 5

-- Выбрать 6-ю строку из курсора liahona:
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)
```

```
-- Закрыть курсор liahona и завершить транзакцию:
CLOSE liahona;
COMMIT WORK;
```

Совместимость

Оператор `MOVE` отсутствует в стандарте SQL.

См. также

CLOSE, DECLARE, FETCH

NOTIFY

NOTIFY — сгенерировать уведомление

Синтаксис

```
NOTIFY канал [ , сообщение ]
```

Описание

Команда `NOTIFY` отправляет событие уведомления вместе с дополнительной строкой «сообщения» всем клиентским приложениям, которые до этого выполнили в текущей базе данных `LISTEN` *канал* с указанным именем канала. Уведомления видны всем пользователям.

`NOTIFY` предоставляет простой механизм межпроцессного взаимодействия для множества процессов, работающих с одной базой данных Postgres Pro. Вместе с уведомлением может быть передана строка сообщения, а передавая дополнительные данные через таблицы базы данных, можно создать более высокоуровневые механизмы обмена структурированными данными.

Информация, передаваемая клиенту с уведомлением, включает имя канала уведомлений, PID серверного процесса, управляющего сеансом, который выдал уведомление, и строку сообщения (она будет пустой, если сообщение не задано).

Выбор подходящих имён каналов и их назначения — дело проектировщика базы данных. Обычно имя канала совпадает с именем какой-либо таблицы в базе, а событие уведомления по сути означает «я изменила эту таблицу, посмотрите, что она содержит теперь». Однако команды `NOTIFY` и `LISTEN` не навязывают именно такой подход. Например, проектировщик базы данных может выбрать разные имена каналов, чтобы сигнализировать о разных типах изменений в одной таблице. Кроме того, строку сообщения тоже можно использовать для выделения различных событий.

Если требуется сигнализировать о факте изменений в определённой таблице, используя `NOTIFY`, можно применить полезный программный приём — поместить `NOTIFY` в триггер уровня оператора, который будет срабатывать при изменениях в таблице. При таком подходе уведомление будет выдаваться автоматически, так что прикладной программист не рискует случайно оставить какое-либо изменение без уведомления.

Транзакции оказывают значительное влияние на работу `NOTIFY`. Во-первых, если `NOTIFY` выполняется внутри транзакции, уведомления доставляются получателям после фиксирования транзакции и только в этом случае. Это разумно, так как в случае прерывания транзакции действие всех команд в ней аннулируется, включая `NOTIFY`. Однако это может обескуражить тех, кто ожидает, что уведомления будут приходить немедленно. Во-вторых, если ожидающий сеанс получает уведомление внутри транзакции, это событие не будет доставлено подключённому клиенту до завершения (фиксации или отката) транзакции. Это опять же объясняется тем, что если уведомление будет доставлено в рамках транзакции, которая затем будет прервана, может возникнуть желание как-то отменить его — но сервер не может «забрать назад» уведомление после того, как оно было отправлено клиенту. Поэтому уведомления доставляются только между транзакциями. Учитывая вышесказанное, в приложениях, применяющих `NOTIFY` для сигнализации в реальном времени, следует минимизировать размер транзакций.

Если в рамках одной транзакции в один канал поступило несколько уведомлений с одинаковым сообщением, подписчикам доставляется только один экземпляр уведомления. Если же сообщения различаются, уведомления будут всегда доставляться по отдельности. Так же уведомления, поступающие от разных транзакций, никогда не будут объединены в одно. Не считая фильтрации последующих экземпляров дублирующихся уведомлений, `NOTIFY` гарантирует, что уведомления от одной транзакции всегда доставляются в том же порядке, в каком были отправлены. Также гарантируется, что сообщения от разных транзакций поступают в порядке фиксирования этих транзакций.

Часто бывает, что клиент, выполнивший `NOTIFY`, ожидает уведомления на этом же канале. В этом случае он получит своё же уведомление, как и любой другой сеанс, ожидающий уведомления. В зависимости от логики приложения, это может привести к бессмысленным операциям, например, поиску изменений в таблице, которые и были внесены этим же сеансом. Этой дополнительной работы можно избежать, если проверить, не совпадает ли PID сигнализирующего процесса (указанный в данных события) с собственным PID сеанса (его можно узнать, обратившись к libpq). Если они совпадают, значит сеанс получил уведомление о собственных действиях, так что его можно игнорировать.

Параметры

`*канал*`

Имя канала для передачи уведомления (любой идентификатор).

`*сообщение*`

Строка «сообщения», которая будет передана вместе с уведомлением. Она должна задаваться простой текстовой константой. В стандартной конфигурации её длина должна быть меньше 8000 байт. (Если требуется передать двоичные данные или большой объём информации, лучше поместить их в таблицу базы данных и передать ключ этой записи.)

Примечания

Уведомления, которые были отправлены, но ещё не обработаны всеми ожидающими сеансами, содержатся в очереди. Если эта очередь переполняется, транзакции, в которых вызывается `NOTIFY`, будут завершены ошибкой при попытке фиксации. Очередь довольно велика (8 ГБ в стандартной конфигурации), так что её размера должно хватать практически во всех случаях, но если в сеансе выполняется `LISTEN`, а затем продолжается очень длительная транзакция, очередь не очищается. Как только эта очередь заполняется наполовину, в журнал записываются предупреждения, в которых указывается, какой сеанс препятствует очистке очереди. В этом случае следует добиться завершения текущей транзакции в указанном сеансе, чтобы очередь была очищена.

Функция `pg_notification_queue_usage` показывает, какой процент очереди в данный момент занят ожидающими уведомлениями. За дополнительными сведениями обратитесь к Разделу 9.27.

Транзакция, в которой выполняется `NOTIFY`, не может быть подготовлена для двухфазной фиксации.

pg_notify

Также отправить уведомление можно, используя функцию `pg_notify(text, text)`. Эта функция принимает в первом аргументе имя канала, а во втором текст сообщения. Гораздо удобнее использовать её, когда требуется работать с динамическими именами каналов и сообщениями.

Примеры

Демонстрация процедуры ожидания/получения уведомления в psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from
 server process with PID 8448.

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process
 with PID 14728.
```

Совместимость

Оператор `NOTIFY` отсутствует в стандарте SQL.

См. также

LISTEN, UNLISTEN, max_notify_queue_pages

PREPARE

PREPARE — подготовить оператор к выполнению

Синтаксис

```
PREPARE имя [ ( тип_данных [, ...] ) ] AS оператор
```

Описание

`PREPARE` создаёт подготовленный оператор. Подготовленный оператор представляет собой объект на стороне сервера, позволяющий оптимизировать производительность приложений. Когда выполняется `PREPARE`, указанный оператор разбирается, анализируется и переписывается. При последующем выполнении команды `EXECUTE` подготовленный оператор планируется и исполняется. Такое разделение труда исключает повторный разбор запроса, при этом позволяет выбрать наилучший план выполнения в зависимости от определённых значений параметров.

Подготовленные операторы могут принимать параметры — значения, которые подставляются в оператор, когда он собственно выполняется. При создании подготовленного оператора к этим параметрам можно обращаться по порядковому номеру, используя запись `$1`, `$2` и т. д. Дополнительно можно указать список соответствующих типов данных параметров. Если тип данных параметра не указан или объявлен как `unknown` (неизвестный), тип выводится из контекста при первом обращении к этому параметру (если это возможно). При выполнении оператора фактические значения параметров передаются команде `EXECUTE`. За подробностями обратитесь к EXECUTE.

Подготовленные операторы существуют только в рамках текущего сеанса работы с БД. Когда сеанс завершается, система забывает подготовленный оператор, так что его надо будет создать снова, чтобы использовать дальше. Это также означает, что один подготовленный оператор не может использоваться одновременно несколькими клиентами базы данных; но каждый клиент может создать собственный подготовленный оператор и использовать его. Освободить подготовленный оператор можно вручную, выполнив команду `DEALLOCATE`.

Подготовленные операторы потенциально дают наибольший выигрыш в производительности, когда в одном сеансе выполняется большое число однотипных операторов. Отличие в производительности особенно значительно, если операторы достаточно сложны для планирования или перезаписи, например, когда в запросе объединяется множество таблиц или необходимо применить несколько правил. Если оператор относительно прост в этом плане, но сложен для выполнения, выигрыш от использования подготовленных операторов будет менее заметным.

Параметры

`имя`

Произвольное имя, назначаемое данному подготовленному оператору. Оно должно быть уникальным в рамках одного сеанса; это имя затем используется для выполнения или освобождения ранее подготовленного оператора.

`тип_данных`

Тип данных параметра подготовленного оператора. Если тип данных конкретного параметра не задан или задан как `unknown`, он будет выводиться из контекста при первом обращении к этому параметру. Для обращения к параметрам в самом подготовленном операторе используется запись `$1`, `$2` и т. д.

`оператор`

Любой оператор `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` или `VALUES`.

Примечания

Подготовленный оператор может выполняться с использованием либо *общего плана*, либо *специализированного*. Общий план не меняется при последующих выполнениях, тогда как специализированный план строится для определённого выполнения с учётом значений параметров, переданных при данном вызове. Использование общего плана снижает издержки планирования, но в ряде случаев специализированный план будет выполняться гораздо эффективнее, так как планировщик может подстроиться под значения параметров. (Разумеется, если у подготовленного оператора нет параметров, специализированный план не имеет смысла, поэтому всегда используется общий план.)

По умолчанию (то есть когда plan_cache_mode имеет значение `auto`), сервер автоматически выбирает, использовать ли для подготовленного оператора с параметрами общий или специализированный план. На данный момент это происходит по следующему принципу — первые пять выполнений производятся со специализированными планами и вычисляется средняя стоимость этих планов. Затем строится общий план и его примерная стоимость сравнивается со средней стоимостью специализированных. При последующих выполнениях общий план будет использоваться, если его стоимость, по сравнению со стоимостью специализированных, не настолько велика, чтобы оправдать повторное планирование.

Эту логику можно переопределить, чтобы выбирались только общие или только специализированные планы, установив для параметра `plan_cache_mode` значение `force_generic_plan` или `force_custom_plan`, соответственно. Это полезно в первую очередь тогда, когда оценка стоимости общего плана по какой-то причине оказывается заниженной, и он выбирается даже когда фактически его использование обходится гораздо дороже, чем использование специализированных планов.

Узнать, какой план выполнения выбирает Postgres Pro для подготовленного оператора, можно, воспользовавшись командой `EXPLAIN`. Например:

```
EXPLAIN EXECUTE имя(значения_параметров);
```

Если применяется общий план, он будет содержать символы параметров `$n`, тогда как в специализированном плане будут подставлены фактические значения параметров.

Более подробно о планировании запросов и статистике, которую собирает Postgres Pro для этих целей, можно узнать в документации ANALYZE.

Хотя основной смысл подготовленных операторов в том, чтобы избежать многократного разбора и планирования оператора, Postgres Pro будет принудительно заново анализировать и планировать выполнение оператора всякий раз, когда объекты базы данных, задействованные в операторе, подвергаются изменениям определения (DDL) или в планировщике меняется их статистика со времени предыдущего использования подготовленного оператора. Кроме того, если от одного использования оператора к другому меняется значение search_path, оператор будет так же разобран заново с новым `search_path`. (Последнее поведение появилось в PostgreSQL 9.3.) С этими правилами использование подготовленного оператора по сути почти не отличается от выполнения одного и того же запроса снова и снова, но даёт выигрыш по скорости (если определения объектов не меняются), особенно если оптимальный план от раза к разу не меняется. Однако различия всё же могут проявиться — например, когда оператор обращается к таблице по неполному имени, а затем в схеме, стоящей в пути `search_path` раньше, создаётся другая таблица с таким же именем, автоматический пересмотр запроса не происходит, так как никакой объект в определении оператора не изменился. Однако если автоматический пересмотр произойдёт в результате других изменений, при последующем выполнении запроса будет задействована новая таблица.

Получить список всех доступных в сеансе подготовленных операторов можно, обратившись к системному представлению `pg_prepared_statements`.

Примеры

Создание подготовленного оператора для команды `INSERT`, который затем выполняется:

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Создание подготовленного оператора для команды `SELECT`, который затем выполняется:

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

В этом примере тип данных второго параметра не указывается, так что он выводится из контекста, в котором используется `$2`.

Совместимость

В стандарте SQL есть оператор `PREPARE`, но он предназначен только для применения во встраиваемом SQL. Эта версия оператора `PREPARE` имеет также несколько другой синтаксис.

См. также

DEALLOCATE, EXECUTE

PREPARE TRANSACTION

PREPARE TRANSACTION — подготовить текущую транзакцию для двухфазной фиксации

Синтаксис

```
PREPARE TRANSACTION id_транзакции
```

Описание

`PREPARE TRANSACTION` подготавливает текущую транзакцию для двухфазной фиксации. После этой команды транзакция перестаёт быть связанной с текущим сеансом; её состояние полностью сохраняется на диске, и есть очень большая вероятность, что она будет успешно зафиксирована, даже если до этого времени работа базы данных будет прервана аварийно.

Подготовленную транзакцию затем можно зафиксировать или отменить командами `COMMIT PREPARED` и `ROLLBACK PREPARED`, соответственно. Эти команды можно вызывать из любого сеанса, не только из того, в котором эта транзакция создавалась.

С точки зрения сеанса, выполняющего команду, `PREPARE TRANSACTION` не отличается от `ROLLBACK`: после её выполнения не активна никакая транзакция, а результат действия подготовленной транзакции становится невидимым (Он окажется видимым снова, если транзакция будет зафиксирована.)

Если при выполнении команды `PREPARE TRANSACTION` по какой-то причине происходит сбой, команда действует как `ROLLBACK`: текущая транзакция откатывается.

Параметры

`id_транзакции`

Произвольный идентификатор, по которому затем на эту транзакцию будут ссылаться команды `COMMIT PREPARED` или `ROLLBACK PREPARED`. Идентификатор должен задаваться строковой константой не длиннее 200 байт и должен отличаться от идентификаторов любых других подготовленных на данный момент транзакций.

Примечания

`PREPARE TRANSACTION` не предназначена для использования в приложениях или интерактивных сеансах. Её задача — дать возможность внешнему менеджеру транзакций выполнять атомарные глобальные транзакции, охватывающие несколько баз данных или другие транзакционные ресурсы. Обычно применять `PREPARE TRANSACTION` следует только при разработке собственного менеджера транзакций.

Эта команда должна выполняться внутри блока транзакции. Начинает блок транзакции команда `BEGIN`.

В настоящее время команда `PREPARE` неспособна подготавливать транзакции, в которых выполнялись какие-либо действия с временными таблицами или во временном пространстве имён сеанса, создавались курсоры `WITH HOLD` либо выполнялись команды `LISTEN`, `UNLISTEN` или `NOTIFY`. Эти функции слишком тесно связаны с текущим сеансом, так что в подготовленной транзакции они не были бы полезны.

Если транзакция меняет какие-либо параметры времени выполнения командой `SET` (без указания `LOCAL`), их значения сохраняются после `PREPARE TRANSACTION` и не зависят от последующих команд `COMMIT PREPARED` и `ROLLBACK PREPARED`. Так что в этом отношении `PREPARE TRANSACTION` больше похожа на `COMMIT`, чем на `ROLLBACK`.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении `pg_prepared_xacts`.

Внимание

Оставлять транзакции в подготовленном состоянии на долгое время не рекомендуется. Это повлияет на способность команды `VACUUM` высвобождать пространство, а в крайнем случае может привести к отключению базы данных для предотвращения зацикливания ID транзакций (см. Подраздел 23.1.5). Также учтите, что транзакция продолжит удерживать все свои блокировки. Это сделано с расчётом на то, что подготовленная транзакция будет зафиксирована или отменена как только внешний менеджер транзакций убедится, что все другие базы данных так же готовы к фиксации.

В отсутствие настроенного внешнего менеджера транзакций, который бы отслеживал подготовленные транзакции и своевременно закрывал их, лучше вовсе отключить поддержку подготовленных транзакций, установив max_prepared_transactions равным нулю. Это не позволит случайно создать подготовленные транзакции, которые могут быть забыты и в конце концов станут причиной проблем.

Примеры

Текущая транзакция подготавливается для двухфазной фиксации, при этом ей назначается идентификатор `foobar`:

```
PREPARE TRANSACTION 'foobar';
```

Совместимость

Оператор `PREPARE TRANSACTION` является расширением Postgres Pro. Он предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

См. также

COMMIT PREPARED, ROLLBACK PREPARED

REASSIGN OWNED

REASSIGN OWNED — сменить владельца объектов базы данных, принадлежащих заданной роли

Синтаксис

```
REASSIGN OWNED BY { старая_роль | CURRENT_ROLE | CURRENT_USER | SESSION_USER } [, ...]
               TO { новая_роль | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

Описание

`REASSIGN OWNED` указывает системе сменить владельца объектов баз данных, принадлежащих одной из *старых_ролей*, на *новую_роль*.

Параметры

старая_роль

Имя роли. Все объекты в текущей базе данных и все общие объекты (базы данных, табличные пространства), принадлежащие этой роли, станут принадлежать *новой_роли*.

новая_роль

Имя роли, которая станет новым владельцем затронутых объектов.

Примечания

`REASSIGN OWNED` часто применяется при подготовке к удалению одной или нескольких ролей. Так как команда `REASSIGN OWNED` затрагивает объекты только в текущей базе данных, обычно её нужно выполнять в каждой базе данных, которая содержит объекты, принадлежащие удаляемой роли.

Для выполнения `REASSIGN OWNED` необходимо быть членом и исходной, и целевой роли.

Команда `DROP OWNED` предлагает альтернативное решение, просто удаляя все объекты базы данных, принадлежащие одной или нескольким заданным ролям.

Команда `REASSIGN OWNED` не затрагивает никакие права, которые даны *старым_ролям* для объектов, им не принадлежащим. Также она не затрагивает права по умолчанию, установленные командой `ALTER DEFAULT PRIVILEGES`. Отозвать эти права можно, воспользовавшись командой `DROP OWNED`.

За подробностями обратитесь к Разделу 20.4.

Совместимость

Оператор `REASSIGN OWNED` является расширением Postgres Pro.

См. также

DROP OWNED, DROP ROLE, ALTER DATABASE

REFRESH MATERIALIZED VIEW

REFRESH MATERIALIZED VIEW — заменить содержимое материализованного представления

Синтаксис

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] имя
    [ WITH [ NO ] DATA ]
```

Описание

`REFRESH MATERIALIZED VIEW` полностью заменяет содержимое материализованного представления. Эту команду можно выполнить только с правом `MAINTAIN` для материализованного представления. Старое его содержимое при этом аннулируется. Если добавлено указание `WITH DATA` (или нет никакого), нижележащий запрос выполняется и выдаёт новые данные, так что материализованное представление остаётся в сканируемом состоянии. Если указано `WITH NO DATA`, новые данные не выдаются, и оно оказывается в несканируемом состоянии.

Указать `CONCURRENTLY` вместе с `WITH NO DATA` нельзя.

Параметры

`CONCURRENTLY`

Обновить материализованное представление, не блокируя параллельные выборки из него. Без данного параметра обновление, затрагивающее много строк, обычно задействует меньше ресурсов и выполнится быстрее, но может препятствовать чтению этого материализованного представления другими сеансами. При этом данный режим может быть быстрее при небольшом количестве строк.

Данный параметр допускается, только если в материализованном представлении есть хотя бы один индекс `UNIQUE`, построенный только по именам столбцов и включающий все строки (то есть это не должен быть индекс по выражению или индекс, содержащий `WHERE`).

Этот параметр нельзя использовать, когда материализованное представление ещё не наполнено.

Даже с этим параметром в один момент времени допускается только одно обновление (`REFRESH`) материализованного представления.

`имя`

Имя (возможно, дополненное схемой) материализованного представления, подлежащего обновлению.

Примечания

Если в запросе, определяющем материализованное представление, есть предложение `ORDER BY`, исходное содержимое материализованного представления будет упорядочено указанным образом; но `REFRESH MATERIALIZED VIEW` не гарантирует сохранения этого порядка.

При выполнении команды `REFRESH MATERIALIZED VIEW` search_path временно изменяется на `pg_catalog, pg_temp`.

Примеры

Эта команда заменяет содержимое материализованного представления `order_summary`, используя запрос из определения материализованного представления, и оставляет его в сканируемом состоянии:

```
REFRESH MATERIALIZED VIEW order_summary;
```

Эта команда освобождает пространство, связанное с материализованным представлением `annual_statistics_basis`, и оставляет это представление в нескануемом состоянии:

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

Совместимость

`REFRESH MATERIALIZED VIEW` — расширение Postgres Pro.

См. также

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

REINDEX

REINDEX — перестроить индексы

Синтаксис

```
REINDEX [ ( параметр [, ...] ) ] { INDEX | TABLE | SCHEMA } [ CONCURRENTLY ] имя
REINDEX [ ( параметр [, ...] ) ] { DATABASE | SYSTEM } [ CONCURRENTLY ] [ имя ]

Здесь допускается параметр:

    CONCURRENTLY [ boolean ]
    TABLESPACE новое_табл_пространство
    VERBOSE [ boolean ]
```

Описание

`REINDEX` перестраивает индекс, обрабатывая данные таблицы, к которой относится индекс, и в результате заменяет старую копию индекса. Команда `REINDEX` применяется в следующих ситуациях:

- Индекс был повреждён, его содержимое стало некорректным. Хотя в теории этого не должно случаться, на практике индексы могут испортиться из-за программных ошибок или аппаратных сбоев. В таких случаях `REINDEX` служит методом восстановления индекса.
- Индекс стал «раздутым», то есть в нём оказалось много пустых или почти пустых страниц. Это может происходить с B-деревьями в Postgres Pro при определённых, достаточно редких сценариях использования. `REINDEX` даёт возможность сократить объём, занимаемый индексом, записывая новую версию индекса без «мёртвых» страниц. За подробностями обратитесь к Разделу 23.2.
- Параметр хранения индекса (например, фактор заполнения) был изменён, и теперь требуется, чтобы это изменение вступило в силу в полной мере.
- Если построить индекс в режиме `CONCURRENTLY` не удаётся, индекс остаётся в «нерабочем» состоянии. Такие индексы бесполезны, но их можно легко перестроить, воспользовавшись командой `REINDEX`. Однако заметьте, что перестраивать индекс в неблокирующем режиме может только команда `REINDEX INDEX`.

Параметры

`INDEX`

Перестраивает указанный индекс. Эту форму `REINDEX` нельзя выполнить в блоке транзакции, если индекс является секционированным.

`TABLE`

Перестраивает все индексы указанной таблицы. Если таблица имеет вторичную таблицу «TOAST», она также будет переиндексирована. Эту форму `REINDEX` нельзя выполнить в блоке транзакции, если таблица является секционированной.

`SCHEMA`

Перестраивает все индексы в указанной схеме. Если таблица в этой схеме имеет вторичную таблицу «TOAST», она также будет переиндексирована. При этом обрабатываются и индексы в общих системных каталогах. Эту форму `REINDEX` нельзя выполнить в блоке транзакции.

`DATABASE`

Перестраивает все индексы в текущей базе данных. При этом индексы в системных каталогах не обрабатываются. Эту форму `REINDEX` нельзя выполнить в блоке транзакции.

`SYSTEM`

Перестраивает все индексы в системных каталогах текущей базы данных. При этом обрабатываются также индексы в общих системных каталогах, но индексы в таблицах пользователя не затрагиваются. Эту форму `REINDEX` нельзя выполнить в блоке транзакции.

`имя`

Имя определённого индекса, таблицы или базы данных, подлежащих переиндексации. В настоящее время `REINDEX DATABASE` и `REINDEX SYSTEM` могут переиндексировать только текущую базу данных, так что их параметр (необязательный) должен соответствовать имени текущей базы данных.

`CONCURRENTLY`

С этим указанием Postgres Pro перестроит индекс, не устанавливая никаких блокировок, которые бы предотвращали добавление, изменение или удаление записей в таблице, тогда как по умолчанию операция перестроения индекса блокирует запись (но не чтение) в таблице до своего завершения. При переиндексации в неблокирующем режиме есть ряд особенностей, о которых следует знать, — см. Неблокирующее перестроение индексов ниже.

Для временных таблиц `REINDEX` всегда выполняется более простым, неблокирующим способом, так как они не могут использоваться никакими другими сеансами.

`TABLESPACE`

Задаёт новое табличное пространство, в котором будут перестроены индексы.

`VERBOSE`

Выводит отчёт о прогрессе после переиндексации каждого индекса.

`boolean`

Включает или отключает заданный параметр. Для включения параметра можно написать `TRUE`, `ON` или `1`, а для отключения — `FALSE`, `OFF` или `0`. Значение *`boolean`* можно опустить, в этом случае подразумевается `TRUE`.

`новое_табл_пространство`

Табличное пространство, в котором будут перестраиваться индексы.

Примечания

В случае подозрений в повреждении индекса таблицы пользователя, этот индекс или все индексы таблицы можно перестроить, используя команду `REINDEX INDEX` или `REINDEX TABLE`.

Всё усложняется, если возникает необходимость восстановить повреждённый индекс системной таблицы. В этом случае важно, чтобы система сама не использовала этот индекс. (На самом деле в таких случаях вы, скорее всего, столкнётесь с падением процессов сервера в момент запуска, как раз вследствие испорченных индексов.) Чтобы надёжно восстановить рабочее состояние, сервер следует запускать с параметром `-P`, который отключает использование индексов при поиске в системных каталогах.

Один из вариантов сделать это — выключить сервер Postgres Pro и запустить его снова в однопользовательском режиме, с параметром `-P` в командной строке. Затем можно выполнить `REINDEX DATABASE`, `REINDEX SYSTEM`, `REINDEX TABLE` или `REINDEX INDEX`, в зависимости от того, что вы хотите восстановить. В случае сомнений выполните `REINDEX SYSTEM`, чтобы перестроить все системные индексы в базе данных. Затем завершите однопользовательский сеанс сервера и перезапустите сервер в обычном режиме. Чтобы подробнее узнать, как работать с сервером в однопользовательском интерфейсе, обратитесь к справочной странице postgres.

Можно так же запустить обычный экземпляр сервера, но добавить в параметры командной строки `-P`. В разных клиентах это может делаться по-разному, но во всех клиентах на базе libpq можно установить для переменной окружения `PGOPTIONS` значение `-P` до запуска клиента. Учтите, что хотя этот метод не препятствует работе других клиентов, всё же имеет смысл не позволять им подключаться к повреждённой базе данных до завершения восстановления.

Действие `REINDEX` подобно удалению и пересозданию индекса в том смысле, что содержимое индекса пересоздаётся с нуля, но блокировки при этом устанавливаются другие. `REINDEX` блокирует запись, но не чтение родительской таблицы индекса. В частности, планировщик пытается установить блокировку `ACCESS SHARE` для каждого индекса таблицы, независимо от запроса, поэтому `REINDEX` блокирует практически любые запросы, кроме некоторых подготовленных запросов, план которых был кеширован и которые не используют этот конкретный индекс. `DROP INDEX`, напротив, моментально устанавливает блокировку `ACCESS EXCLUSIVE` на родительскую таблицу, блокируя и запись, и чтение. Последующая команда `CREATE INDEX` блокирует запись, но не чтение; так как индекс отсутствует, обращений к нему ни при каком чтении не будет, что означает, что блокироваться чтение не будет, но выполняться оно будет как дорогостоящее последовательное сканирование.

На время выполнения команды `REINDEX` search_path временно меняется на `pg_catalog, pg_temp`.

Для перестраивания одного индекса или индексов таблицы необходимо иметь право `MAINTAIN` для этой таблицы. Обратите внимание, что хотя команды `REINDEX` для секционированного индекса или таблицы требуют наличия права `MAINTAIN` для секционированной таблицы, эти команды не проверяют наличие прав при обработке отдельных секций. Для переиндексирования схемы или базы данных необходимо быть владельцем этой схемы или базы или иметь права роли `pg_maintain`. Заметьте в частности, что вследствие этого не только суперпользователи могут перестраивать индексы таблиц, принадлежащих другим пользователям. Однако из этих правил есть исключение — команды `REINDEX DATABASE`, `REINDEX SCHEMA` и `REINDEX SYSTEM` пропускают индексы общих каталогов, если только у пользователя нет права `MAINTAIN` для каталога.

Перестроение секционированных индексов или индексов секционированных таблиц поддерживается соответственно командами `REINDEX INDEX` и `REINDEX TABLE`. Каждая секция указанного секционированного отношения переиндексируется в отдельной транзакции. Когда эти команды применяются для обработки секционированной таблицы или индекса, они не могут находиться в блоке транзакции.

Когда для команды `REINDEX`, обрабатывающей секционированный индекс или таблицу, применяется указание `TABLESPACE`, ссылки на табличное пространство меняются только в конечных секциях. Так как собственно секционированные индексы не модифицируются, для них рекомендуется отдельно выполнить `ALTER TABLE ONLY`, чтобы любые секции, присоединяемые в будущем, относились к новому табличному пространству. В случае сбоя эта команда может не перенести все индексы в новое табличное пространство. При повторном запуске она будет перестраивать все конечные секции и перенесёт ранее необработанные индексы в новое пространство.

Если указание `TABLESPACE` задаётся вместе с `SCHEMA`, `DATABASE` или `SYSTEM`, системные отношения пропускаются и выдаётся единственное предупреждение об этом. Индексы TOAST-таблиц перестраиваются, но не перемещаются в новое табличное пространство.

Неблокирующее перестроение индексов

Перестроение индекса может мешать обычной работе с базой данных. Обычно Postgres Pro блокирует запись в переиндексируемую таблицу и выполняет всю операцию построения индекса за одно сканирование таблицы. Другие транзакции могут продолжать читать таблицу, но при попытке вставить, изменить или удалить строки в таблице они будут заблокированы до завершения перестроения индекса. Это может оказать нежелательное влияние на работу производственной базы данных. Индексация очень больших таблиц может занимать много часов, и даже для маленьких таблиц перестроение индекса может заблокировать записывающие процессы на время, неприемлемое для производственной системы.

Postgres Pro поддерживает перестроение индексов в режиме минимизации блокировок записи. Этот режим включается указанием `CONCURRENTLY` команды `REINDEX`. С данным указанием Postgres Pro должен выполнить два сканирования таблицы для каждого индекса, который нужно перестроить, и должен дождаться завершения всех активных транзакций, которые могут использовать данный индекс. В связи с этим в неблокирующем режиме производится в целом больше действий, и длительность переиндексирования значительно увеличивается. Однако благодаря тому, что во время перестроения индекса могут выполняться другие обычные операции, этот режим полезен, когда требуется перестроить индексы в производственной среде. Разумеется, другие операции могут несколько замедлиться из-за дополнительной нагрузки на процессор, память и ввод-вывод, связанной с перестроением индекса.

В ходе неблокирующего переиндексирования производятся следующие действия (каждое в отдельной транзакции). Если переиндексированию подлежат несколько индексов, сначала для всех индексов полностью выполняется один этап, а затем другой.

1. В каталог `pg_index` добавляется переходное определение индекса, которое затем заменит старое. Для предотвращения каких-либо изменений в схеме во время операции обрабатываемые индексы, а также связанные с ними таблицы защищаются блокировкой `SHARE UPDATE EXCLUSIVE` на уровне сеанса.
2. Для каждого нового индекса выполняется первый проход, на котором строится индекс. Когда индекс построен, его флаг `pg_index.indisready` переходит в состояние «true», чтобы этот индекс был готов к добавлениям, и таким образом он становится видимым для других сеансов сразу после окончания построившей его транзакции. Это действие выполняется в отдельной транзакции для каждого индекса.
3. Затем выполняется второй проход, на котором в индекс вносятся кортежи, добавленные в таблицу во время первого прохода. Это действие также выполняется в отдельной транзакции для каждого индекса.
4. Все ограничения, ссылающиеся на индекс, переключаются на определение нового индекса, а также меняются имена индексов. В этот момент флаг `pg_index.indisvalid` нового индекса принимает значение «true», а старого — «false», и производится сброс кеша, в результате чего все сеансы, обращавшиеся к старому индексу, получают новую информацию.
5. Флаг `pg_index.indisready` старого индекса сбрасывается в «false» во избежание добавления в него новых кортежей, как только завершатся текущие запросы, которые могли обращаться к этому индексу.
6. Старые индексы удаляются. Блокировки `SHARE UPDATE EXCLUSIVE` уровня сеанса, установленные для индексов и таблиц, снимаются.

Если при перестроении индексов возникает проблема, например нарушение уникальности в уникальном индексе, `REINDEX` прерывается, но оставляет после себя «нерабочий» новый индекс в дополнение к уже существующему. Этот индекс будет игнорироваться запросами, так как он может быть неполным; тем не менее он будет обновляться при изменении данных, что повлечёт дополнительные издержки. Команда psql `\d` будет обозначать такой индекс как `INVALID` (нерабочий):

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
Indexes:
    "idx" btree (col)
    "idx_ccnew" btree (col) INVALID
```

Если имя индекса с пометкой `INVALID` оканчивается на `ccnew`, это переходный индекс созданный при параллельной операции, и для исправления ситуации рекомендуется удалить его, выполнив `DROP INDEX`, а затем попытаться ещё раз выполнить `REINDEX CONCURRENTLY`. Если же имя нерабочего индекса оканчивается на `ccold`, значит, это исходный индекс, удалить который по какой-то причине не получилось. Такой индекс рекомендуется просто удалить, так как нужный индекс был перестроен успешно. К суффиксам имён нерабочих индексов может быть добавлено ненулевое число для сохранения их уникальности, например `_ccnew1`, `_ccold2` и т. д.

Обычное построение индекса допускает одновременное построение других индексов для таблицы обычным методом, но неблокирующее построение для конкретной таблицы в один момент времени допускается только одно. Однако в любом случае никакие другие изменения схемы таблицы в это время не разрешаются. Другое отличие состоит в том, что в блоке транзакции может быть выполнена обычная команда `REINDEX TABLE` или `REINDEX INDEX`, но не `REINDEX CONCURRENTLY`.

Как и любая длительная транзакция, операция `REINDEX` с таблицей может повлиять на возможность удаления кортежей параллельной операцией `VACUUM` с какой-либо другой таблицей.

Команда `REINDEX SYSTEM` не поддерживает указание `CONCURRENTLY`, так как системные каталоги нельзя переиндексировать в неблокирующем режиме.

Более того, в неблокирующем режиме нельзя перестроить индексы, связанные с ограничениями-исключениями. Если явно указать имя такого индекса в команде, будет выдана ошибка. Когда в неблокирующем режиме переиндексируется таблица или база данных, содержащая такие индексы, эти индексы пропускаются. (Перестроить такие индексы можно в обычном режиме, без указания `CONCURRENTLY`.)

Каждый процесс, выполняющий операцию `REINDEX`, будет выдавать информацию о ходе её выполнения, отображаемую в представлении `pg_stat_progress_create_index`. За подробностями обратитесь к Подразделу 26.4.4.

Примеры

Перестроение одного индекса:

```
REINDEX INDEX my_index;
```

Перестроение всех индексов таблицы `my_table`:

```
REINDEX TABLE my_table;
```

Перестроение всех индексов в определённой базе данных, в предположении, что целостность системных индексов под сомнением:

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

Перестроение индексов таблицы, допускающее одновременные операции чтения и записи с затрагиваемыми в процессе переиндексации отношениями:

```
REINDEX TABLE CONCURRENTLY my_broken_table;
```

Совместимость

Команда `REINDEX` отсутствует в стандарте SQL.

См. также

CREATE INDEX, DROP INDEX, reindexdb, Подраздел 26.4.4

RELEASE SAVEPOINT

RELEASE SAVEPOINT — высвободить ранее определённую точку сохранения

Синтаксис

```
RELEASE [ SAVEPOINT ] имя_точки_сохранения
```

Описание

Команда `RELEASE SAVEPOINT` освобождает указанную точку сохранения и все активные точки сохранения, созданные после указанной, а также освобождает соответствующие ресурсы. Все изменения, сделанные с момента создания точки сохранения (но не отменённые), объединяются в транзакцию или в точку сохранения, активную на момент создания указанной точки сохранения. Изменения, сделанные после `RELEASE SAVEPOINT`, также включаются в активную транзакцию или точку сохранения.

Параметры

`имя_точки_сохранения`

Имя точки сохранения, подлежащей освобождению.

Примечания

Указание имени точки сохранения, не определённой ранее, считается ошибкой.

Освободить точку сохранения в транзакции, находящейся в прерванном состоянии, можно только при помощи ROLLBACK TO SAVEPOINT.

Если одно имя дано нескольким ранее определённым точкам сохранения, освобождена будет только последняя из них. Повторные команды будут последовательно освобождать более ранние точки сохранения.

Примеры

Этот пример показывает, как установить и затем освободить точку сохранения:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

Данная транзакция вставит значения 3 и 4.

Более сложный пример с несколькими вложенными подтранзакциями:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT sp1;
    INSERT INTO table1 VALUES (2);
    SAVEPOINT sp2;
    INSERT INTO table1 VALUES (3);
    RELEASE SAVEPOINT sp2;
    INSERT INTO table1 VALUES (4))); -- выдаст ошибку
```

В этом примере приложение запрашивает освобождение точки сохранения `sp2`, добавившей 3. При этом контекст добавляющей транзакции изменится на `sp1`. Когда оператор, добавляющий 4, выдаст ошибку, добавление значений 2 и 4 будет потеряно, поскольку они входят в отменённую точку сохранения, а значение 3 находится в том же транзакционном контексте. Поскольку остальные команды игнорируются, приложение может выбрать только одну из следующих двух команд:

```
ROLLBACK;
ROLLBACK TO SAVEPOINT sp1;
```

При выборе `ROLLBACK` все изменения будут отменены, включая добавление значения 1, а при выборе `ROLLBACK TO SAVEPOINT sp1` значение 1 сохранится, и выполнение транзакции продолжится.

Совместимость

Эта команда соответствует стандарту SQL. В стандарте говорится, что ключевое слово `SAVEPOINT` является обязательным, но Postgres Pro позволяет опускать его.

См. также

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

RESET

RESET — восстановить значение по умолчанию заданного параметра времени выполнения

Синтаксис

```
RESET параметр_конфигурации
RESET ALL
```

Описание

`RESET` сбрасывает параметры времени выполнения к значениям по умолчанию. `RESET` — альтернативное написание команды

```
SET параметр_конфигурации TO DEFAULT
```

За подробностями обратитесь к SET.

Значение по умолчанию определяется как значение, которое имел бы этот параметр, если бы для него не выполнялась команда `SET` в текущем сеансе. Фактическое значение может быть задано в компилируемом коде, файле конфигурации, параметрах командной строки или в параметрах по умолчанию для базы данных или пользователя. Если определить его как «значение, которое имеет параметр в начале сеанса», это будет не вполне корректно, так как оно будет сброшено к тому, что задано в файле конфигурации в данный момент. За подробностями обратитесь к Главе 18.

Транзакционное поведение команды `RESET` не отличается от `SET`: её действие будет отменено при откате транзакции.

Параметры

параметр_конфигурации

Имя устанавливаемого параметра конфигурации времени выполнения. Доступные параметры описаны в Главе 18 и на справочной странице SET.

`ALL`

Сбрасывает к значениям по умолчанию все устанавливаемые параметры времени выполнения.

Примеры

Сброс конфигурационной переменной `timezone` к значению по умолчанию:

```
RESET timezone;
```

Совместимость

`RESET` является расширением Postgres Pro.

См. также

SET, SHOW

REVOKE

REVOKE — отозвать права доступа

Синтаксис

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER |
 MAINTAIN }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] имя_таблицы [, ...]
         | ALL TABLES IN SCHEMA имя_схемы [, ...] }
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( имя_столбца [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( имя_столбца [, ...] ) }
    ON [ TABLE ] имя_таблицы [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE имя_последовательности [, ...]
         | ALL SEQUENCES IN SCHEMA имя_схемы [, ...] }
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE имя_бд [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN имя_домена [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER имя_обёртки_сторонних_данных [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER имя_сервера [, ...]
```

```
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } имя_функции [ ( [ [ режим_аргумента ]
 [ имя_аргумента ] тип_аргумента [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA имя_схемы [, ...] }
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE имя_языка [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT oid_БО [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SET | ALTER SYSTEM } [, ...] | ALL [ PRIVILEGES ] }
    ON PARAMETER параметр_конфигурации [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA имя_схемы [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE табл_пространство [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE имя_типа [, ...]
    FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
    [ CASCADE | RESTRICT ]

REVOKE [ { ADMIN | INHERIT | SET } OPTION FOR ]
    имя_роли [, ...] FROM указание_роли [, ...]
    [ GRANTED BY указание_роли ]
```

```
    [ CASCADE | RESTRICT ]
```

Здесь *указание_роли*:

```
    [ GROUP ] имя_роли
  | PUBLIC
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Описание

Команда `REVOKE` лишает одну или несколько ролей прав, назначенных ранее. Ключевое слово `PUBLIC` обозначает неявно определённую группу всех ролей.

Различные типы прав подробно рассматриваются в описании команды `GRANT`.

Заметьте, что любая конкретная роль получает в сумме права, данные непосредственно ей, права, данные любой роли, в которую она включена, а также права, данные группе `PUBLIC`. Поэтому, например, лишение `PUBLIC` права `SELECT` не обязательно будет означать, что все роли лишатся права `SELECT` для данного объекта: оно сохранится у тех ролей, которым оно дано непосредственно или косвенно, через другую роль. Подобным образом, лишение права `SELECT` какого-либо пользователя может не повлиять на его возможность пользоваться правом `SELECT`, если это право дано группе `PUBLIC` или другой роли, в которую он включён.

Если указано `GRANT OPTION FOR`, отзывается только право передачи права, но не само право. Без этого указания отзывается и право, и право распоряжаться им.

Если пользователь обладает правом с правом передачи и он дал его другим пользователям, последнее право считается зависимым. Когда первый пользователь лишается самого права или права передачи и существуют зависимые права, эти зависимые права также отзываются, если дополнительно указано `CASCADE`; в противном случае операция завершается ошибкой. Это рекурсивное лишение прав затрагивает только права, полученные через цепочку пользователей, которую можно проследить до пользователя, являющегося субъектом команды `REVOKE`. Таким образом, пользователи могут в итоге сохранить это право, если оно было также получено через других пользователей.

Когда отзывается право доступа к таблице, с ним вместе автоматически отзываются соответствующие права для каждого столбца таблицы (если такие права заданы). С другой стороны, если роли были даны права для таблицы, лишение роли таких же прав на уровне отдельных столбцов ни на что не влияет.

Когда пользователь лишается членства в роли, указание `GRANT OPTION` меняется на `ADMIN OPTION`, но в остальном поведение не отличается. Обратите внимание, что в версиях Postgres Pro до 16 при назначении членства в ролях зависимые права не отслеживались, то есть указание `CASCADE` при этом не действовало. Сейчас зависимые права отслеживаются. Заметьте также, что эта форма команды не принимает избыточное слово `GROUP` в *указании_роли*.

Можно отозвать право `INHERIT OPTION` или `SET OPTION` так же, как `ADMIN OPTION`. Это равнозначно присваиванию параметру значения `FALSE`.

Примечания

Пользователь может отзывать только те права, которые он дал другому непосредственно. Если, например, пользователь A дал право с правом передачи пользователю B, а пользователь B, в свою очередь, дал это право пользователю C, то пользователь A не сможет лишить этого права непосредственно C. Вместо этого, пользователь A может лишить права передачи права пользователя B и использовать параметр `CASCADE`, чтобы этого права по цепочке лишился пользователь C. Или же, например, если и A, и B дали одно и то же право C, то A сможет отозвать право, которое дал он, но не пользователь B, так что в результате C всё равно будет иметь это право.

Если отозвать право доступа к объекту (с помощью `REVOKE`) попытается не владелец объекта, команда завершится ошибкой, если пользователь не имеет никаких прав для этого объекта. Если же пользователь имеет какие-то права, команда будет выполняться, но пользователь сможет отозвать только те права, которые даны ему с правом распоряжения ими. Формы `REVOKE ALL PRIVILEGES` будут выдавать предупреждение, если у него вовсе нет таких прав, тогда как другие формы будут выдавать предупреждения, если пользователь не имеет права распоряжаться именно правами, указанными в команде. (В принципе, эти утверждения применимы и к владельцу объекта, но ему разрешено распоряжаться всем правами, поэтому такие ситуации невозможны.)

Если команду `GRANT` или `REVOKE` выполняет суперпользователь, эта команда выполняется так, как будто её выполняет владелец затрагиваемого объекта. (Поскольку у ролей владельцев нет, `GRANT` для назначения членства в роли выполняется, как от имени начального суперпользователя.) Так как все права в конце концов исходят от владельца объекта (возможно, косвенно по цепочке или через право распоряжением правом), суперпользователь может отозвать все права, но это может потребовать применения режима `CASCADE`, как описывалось выше.

`REVOKE` также может быть выполнена ролью, которая не является владельцем заданного объекта, но является членом роли-владельца, либо членом роли, имеющей права `WITH GRANT OPTION` для этого объекта. В этом случае команда будет выполнена, как если бы её выполняла содержащая роль, действительно владеющая объектом или имеющая права `WITH GRANT OPTION`. Например, если таблица `t1` принадлежит роли `g1`, членом которой является роль `u1`, то `u1` может отзывать права на использование `t1`, которые записаны как данные ролью `g1`. В том числе это могут быть права, данные ролью `u1`, а также другими членами роли `g1`.

Если роль, выполняющая команду `REVOKE`, получила указанные права косвенно по нескольким путям членства ролей, какая именно роль будет выбрана для выполнения команды, не определено. В таких случаях рекомендуется воспользоваться командой `SET ROLE` и переключиться на роль, которую хочется видеть в качестве выполняющей `REVOKE`. Если этого не сделать, могут быть отозваны не те права, что планировалось, либо отозвать права вообще не удастся.

Подробнее о конкретных типах прав, а также о том, как просмотреть права, назначенные для объектов, рассказывается в Разделе 5.8.

Примеры

Лишение группы public права добавлять данные в таблицу `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Лишение пользователя `manuel` всех прав для представления `kinds`:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

Заметьте, что на самом деле это означает «лишить всех прав, которые дал я».

Исключение из членов роли `admins` пользователя `joe`:

```
REVOKE admins FROM joe;
```

Совместимость

Замечания по совместимости, приведённые для команды `GRANT`, справедливы и для `REVOKE`. Стандарт требует обязательного указания ключевого слова `RESTRICT` или `CASCADE`, но Postgres Pro подразумевает `RESTRICT` по умолчанию.

См. также

GRANT, ALTER DEFAULT PRIVILEGES

ROLLBACK

ROLLBACK — прервать текущую транзакцию

Синтаксис

```
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Описание

`ROLLBACK` откатывает текущую транзакцию и приводит к аннулированию всех изменений, произведённых транзакцией.

Параметры

`WORK`
`TRANSACTION`

Необязательные ключевые слова, не оказывают никакого влияния.

`AND CHAIN`

Если добавляется указание `AND CHAIN`, сразу после окончания текущей транзакции начинается новая (не прерванная) с такими же характеристиками транзакции (см. SET TRANSACTION). В противном случае новая транзакция не начинается.

Примечания

Чтобы завершить и зафиксировать транзакцию, используйте `COMMIT`.

При выполнении команды `ROLLBACK` вне блока транзакции выдаётся предупреждение и больше ничего не происходит. Однако `ROLLBACK AND CHAIN` вне блока транзакции вызывает ошибку.

Примеры

Чтобы прервать все операции:

```
ROLLBACK;
```

Совместимость

Команда `ROLLBACK` соответствует стандарту SQL, а форма `ROLLBACK TRANSACTION` является расширением Postgres Pro.

См. также

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

ROLLBACK PREPARED

ROLLBACK PREPARED — отменить транзакцию, которая ранее была подготовлена для двухфазной фиксации

Синтаксис

```
ROLLBACK PREPARED id_транзакции
```

Описание

`ROLLBACK PREPARED` откатывает транзакцию в подготовленном состоянии.

Параметры

`id_транзакции`

Идентификатор транзакции, которую нужно откатить.

Примечания

Откатить подготовленную транзакцию может либо пользователь, выполнявший её изначально, либо суперпользователь. При этом не обязательно работать в том же сеансе, где выполнялась транзакция.

Эту команду нельзя выполнить внутри блока транзакции. Подготовленная транзакция откатывается немедленно.

Все существующие в текущий момент подготовленные транзакции показываются в системном представлении `pg_prepared_xacts`.

Примеры

Откат транзакции, имеющей идентификатор `foobar`:

```
ROLLBACK PREPARED 'foobar';
```

Совместимость

Оператор `ROLLBACK PREPARED` является расширением Postgres Pro. Он предназначен для использования внешними системами управления транзакциями, некоторые из которых работают по стандартам (например, X/Open XA), но сторона SQL в этих системах не стандартизирована.

См. также

PREPARE TRANSACTION, COMMIT PREPARED

ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT — откатиться к точке сохранения

Синтаксис

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] имя_точки_сохранения
```

Описание

Откатывает все команды, выполненные после установления точки сохранения, и начинает новую подтранзакцию на том же уровне транзакции. Точка сохранения остаётся действующей и при необходимости можно снова откатиться к ней позже.

`ROLLBACK TO SAVEPOINT` неявно уничтожает все точки сохранения, установленные после заданной точки.

Параметры

имя_точки_сохранения

Точка сохранения, к которой нужно откатиться.

Примечания

Чтобы уничтожить точку сохранения, не отменяя действия команд, выполненных после неё, применяется команда `RELEASE SAVEPOINT`.

Указание имени точки сохранения, не установленной ранее, считается ошибкой.

Курсоры проявляют не совсем транзакционное поведение применительно к точкам сохранения. Любой курсор, открытый внутри точки сохранения, будет закрыт при откате к этой точке. Если ранее открытый курсор был перемещён командой `FETCH` или `MOVE` внутри точки сохранения, к которой затем произошёл откат, курсор остаётся в той позиции, в которой он остался после `FETCH` (то есть, перемещение курсора, производимое командой `FETCH`, не откатывается). Также при откате не отменяется и закрытие курсора. Однако другие побочные эффекты, вызываемые запросом курсора (например, побочные действия изменчивых функций, вызываемых в запросе) *отменяются*, если они производятся после точки сохранения, к которой затем происходит откат. Курсор, выполнение которого приводит к прерыванию транзакции, переводится в нерабочее состояние, так что даже если восстановить транзакцию, выполнив `ROLLBACK TO SAVEPOINT`, этот курсор нельзя будет использовать.

Примеры

Отмена действия команд, выполненных после установки точки сохранения `my_savepoint`:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Откат к точке сохранения не отражается на положении курсора:

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1
```

```
ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

Совместимость

В стандарте SQL говорится, что ключевое слово `SAVEPOINT` является обязательным, но Postgres Pro и Oracle позволяют опускать его. SQL допускает `WORK`, но не `TRANSACTION`, в качестве избыточного слова после `ROLLBACK`. Кроме того, в SQL есть дополнительное предложение `AND [ NO ] CHAIN`, которое в настоящее время не поддерживается в Postgres Pro. В остальном эта команда соответствует стандарту SQL.

См. также

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

SAVEPOINT

SAVEPOINT — определить новую точку сохранения в текущей транзакции

Синтаксис

```
SAVEPOINT имя_точки_сохранения
```

Описание

`SAVEPOINT` устанавливает новую точку сохранения в текущей транзакции.

Точка сохранения — это специальная отметка внутри транзакции, которая позволяет откатить все команды, выполненные после неё, и восстановить таким образом состояние на момент установки этой точки.

Параметры

`имя_точки_сохранения`

Имя, назначаемое новой точке сохранения. Ранее существующая точка с таким именем оказывается недоступной, пока не будут освобождены установленные позже одноимённые точки.

Примечания

Для отката к установленной точке сохранения предназначена команда `ROLLBACK TO`. Чтобы уничтожить точку сохранения, сохраняя изменения, произведённые после того, как она была установлена, применяется команда `RELEASE SAVEPOINT`.

Точки сохранения могут быть установлены только внутри блока транзакции. В одной транзакции можно определить несколько точек сохранения.

Примеры

Установление точки сохранения и затем отмена действия всех команд, выполненных после установленной точки:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

Показанная транзакция вставит в таблицу значения 1 и 3, но не 2.

Этот пример показывает, как установить и затем уничтожить точку сохранения:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

Данная транзакция вставит значения 3 и 4.

Этот пример показывает, как использовать точки сохранения с одним именем:

```
BEGIN;
```

```
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);

    -- откат ко второй точке сохранения
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- вывод строк 1 и 2

    -- освобождение второй точки сохранения
    RELEASE SAVEPOINT my_savepoint;

    -- откат к первой точке сохранения
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- вывод только строки 1
COMMIT;
```

Данная транзакция сначала откатит строку 3, затем строку 2.

Совместимость

Стандарт SQL требует, чтобы точка сохранения уничтожалась автоматически, когда устанавливается другая точка сохранения с тем же именем. В Postgres Pro старая точка сохранения остаётся, хотя при откате или уничтожении будет выбираться только самая последняя. (После уничтожения последней точки командой `RELEASE SAVEPOINT` доступной для команд `ROLLBACK TO SAVEPOINT` и `RELEASE SAVEPOINT` становится следующая.) В остальном оператор `SAVEPOINT` полностью соответствует стандарту.

См. также

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT

SECURITY LABEL

SECURITY LABEL — определить или изменить метку безопасности, применённую к объекту

Синтаксис

```
SECURITY LABEL [ FOR провайдер ] ON
{
  TABLE имя_объекта |
  COLUMN имя_таблицы.имя_столбца |
  AGGREGATE имя_агрегатной_функции ( сигнатура_агр_функции ) |
  DATABASE имя_объекта |
  DOMAIN имя_объекта |
  EVENT TRIGGER имя_объекта |
  FOREIGN TABLE имя_объекта |
  FUNCTION имя_функции [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  LARGE OBJECT oid_большого_объекта |
  MATERIALIZED VIEW имя_объекта |
  [ PROCEDURAL ] LANGUAGE имя_объекта |
  PROCEDURE имя_процедуры [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  PUBLICATION имя_объекта |
  PROFILE имя_объекта |
  ROLE имя_объекта |
  ROUTINE имя_подпрограммы [ ( [ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента
 [, ...] ] ) ] |
  SCHEMA имя_объекта |
  SEQUENCE имя_объекта |
  SUBSCRIPTION имя_объекта |
  TABLESPACE имя_объекта |
  TYPE имя_объекта |
  VIEW имя_объекта
} IS { строковая_константа | NULL }

Здесь сигнатура_агр_функции:

* |
[ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] |
[ [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ] ] ORDER BY
 [ режим_аргумента ] [ имя_аргумента ] тип_аргумента [ , ... ]
```

Описание

`SECURITY LABEL` применяет метку безопасности к объекту базы данных. С определённым объектом может быть связано произвольное количество меток безопасности, по одной для каждого провайдера. Провайдеры меток представляют собой загружаемые модули, которые регистрируют себя, вызывая функцию `register_label_provider`.

> **Примечание**
>
> `register_label_provider` — это не SQL-функция; её можно вызывать только из скомпилированного кода C, загруженного сервером.

Провайдер меток определяет, допустима ли заданная метка и разрешено ли применять эту метку к указанному объекту. Какой смысл вкладывается в данную метку, тоже определяет провайдер меток. Postgres Pro не накладывает никаких ограничений на то, как провайдер должен интерпретировать метки безопасности; он просто обеспечивает механизм их хранения. На практике, этот механизм реализован для того, чтобы в базы данных можно было интегрировать системы мандатного управления доступом (MAC) на базе меток, такие как SELinux. Такие системы принимают все решения по ограничению доступа, учитывая метки объектов, а не традиционные сущности избирательного управления доступом (DAC), такие как пользователи и группы.

Чтобы использовать команду `SECURITY LABEL`, нужно быть владельцем объекта базы данных.

Параметры

имя_объекта
имя_таблицы.имя_столбца
имя_агрегатной_функции
имя_функции
имя_процедуры
имя_подпрограммы

Имя помечаемого объекта. Имена объектов, располагающихся в схемах, например, имена таблиц, функций и т. п., можно дополнить именем схемы.

провайдер

Имя провайдера, с которым будет связана эта метка. Указанный провайдер должен быть загружен и готов выполнять операцию размечивания. Если загружен всего один провайдер, его имя можно опустить для краткости.

режим_аргумента

Режим аргумента функции, процедуры или агрегата: `IN`, `OUT`, `INOUT` или `VARIADIC`. По умолчанию подразумевается `IN`. Заметьте, что `SECURITY LABEL` не учитывает аргументы `OUT`, так как для идентификации функции нужны только типы входных аргументов. Поэтому достаточно перечислить только аргументы `IN`, `INOUT` и `VARIADIC`.

имя_аргумента

Имя аргумента функции, процедуры или агрегата. Заметьте, что на самом деле `SECURITY LABEL` не обращает внимание на имена аргументов, так как для однозначной идентификации функции достаточно только типов аргументов.

тип_аргумента

Тип данных аргумента функции, процедуры или агрегата.

oid_большого_объекта

OID большого объекта.

`PROCEDURAL`

Это слово не несёт смысловой нагрузки.

строковая_константа

Новое значение метки безопасности, записанное в виде строковой константы.

`NULL`

Укажите `NULL`, если метку безопасности нужно удалить.

Примеры

Следующий пример показывает, как можно задать или изменить метку безопасности для таблицы.

```
SECURITY LABEL FOR selinux ON TABLE mytable IS 'system_u:object_r:sepgsql_table_t:s0';
```

Удалить метку можно так:

```
SECURITY LABEL FOR selinux ON TABLE mytable IS NULL;
```

Совместимость

Команда `SECURITY LABEL` отсутствует в стандарте SQL.

См. также

sepgsql

SELECT

SELECT, TABLE, WITH — получить строки из таблицы или представления

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( выражение [, ...] ) ] ]
    [ { * | выражение [ [ AS ] имя_результата ] } [, ...] ]
    [ FROM элемент_FROM [, ...] ]
    [ WHERE условие ]
    [ GROUP BY [ ALL | DISTINCT ] элемент_группирования [, ...] ]
    [ HAVING условие ]
    [ WINDOW имя_окна AS ( определение_окна ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] выборка ]
    [ ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
    [ LIMIT { число | ALL } ]
    [ OFFSET начало [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } { ONLY | WITH TIES } ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF from_reference [, ...] ]
 [ NOWAIT | SKIP LOCKED ] [...] ]

Здесь допускается элемент_FROM:

    [ ONLY ] имя_таблицы [ * ] [ [ AS ] псевдоним [ ( псевдоним_столбца [, ...] ) ] ]
                [ TABLESAMPLE метод_выборки ( аргумент [, ...] ) [ REPEATABLE
 ( затравка ) ] ]
    [ LATERAL ] ( выборка ) [ [ AS ] псевдоним [ ( псевдоним_столбца [, ...] ) ] ]
    имя_запроса_WITH [ [ AS ] псевдоним [ ( псевдоним_столбца [, ...] ) ] ]
    [ LATERAL ] имя_функции ( [ аргумент [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] псевдоним [ ( псевдоним_столбца
 [, ...] ) ] ]
    [ LATERAL ] имя_функции ( [ аргумент [, ...] ] ) [ AS ] псевдоним
 ( определение_столбца [, ...] )
    [ LATERAL ] имя_функции ( [ аргумент [, ...] ] ) AS ( определение_столбца [, ...] )
    [ LATERAL ] ROWS FROM( имя_функции ( [ аргумент [, ...] ] ) [ AS
 ( определение_столбца [, ...] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] псевдоним [ ( псевдоним_столбца
 [, ...] ) ] ]
    элемент_FROM тип_соединения элемент_FROM { ON условие_соединения | USING
 ( столбец_соединения [, ...] ) [ AS псевдоним_использования_соединения ] }
    элемент_FROM NATURAL тип_соединения элемент_FROM
    элемент_FROM CROSS JOIN элемент_FROM

и элемент_группирования может быть следующим:

    ( )
    выражение
    ( выражение [, ...] )
    ROLLUP ( { выражение | ( выражение [, ...] ) } [, ...] )
    CUBE ( { выражение | ( выражение [, ...] ) } [, ...] )
    GROUPING SETS ( элемент_группирования [, ...] )

и запрос_WITH:
```

```
    имя_запроса_WITH [ ( имя_столбца [, ...] ) ] AS [ [ NOT ] MATERIALIZED ] ( выборка
 | values | insert | update | delete | merge )
        [ SEARCH { BREADTH | DEPTH } FIRST BY имя_столбца [, ...]
 SET имя_столбца_послед_поиска ]
        [ CYCLE имя_столбца [, ...] SET имя_столбца_пометки_цикла
 [ TO значение_пометки_цикла DEFAULT пометка_цикла_по_умолчанию ]
 USING имя_столбца_пути_цикла ]

TABLE [ ONLY ] имя_таблицы [ * ]
```

Описание

`SELECT` получает строки из множества таблиц (возможно, пустого). Общая процедура выполнения `SELECT` следующая:

1. Выполняются все запросы в списке `WITH`. По сути они формируют временные таблицы, к которым затем можно обращаться в списке `FROM`. Запрос в `WITH` без указания `NOT MATERIALIZED` выполняется только один раз, даже когда он фигурирует в списке `FROM` неоднократно. (См. Предложение WITH ниже.)

2. Вычисляются все элементы в списке `FROM`. (Каждый элемент в списке `FROM` представляет собой реальную или виртуальную таблицу.) Если список `FROM` содержит несколько элементов, они объединяются перекрёстным соединением. (См. Предложение FROM ниже.)

3. Если указано предложение `WHERE`, все строки, не удовлетворяющие условию, исключаются из результата. (См. Предложение WHERE ниже.)

4. Если присутствует указание `GROUP BY`, либо в запросе вызываются агрегатные функции, вывод разделяется по группам строк, соответствующим одному или нескольким значениям, а затем вычисляются результаты агрегатных функций. Если добавлено предложение `HAVING`, оно исключает группы, не удовлетворяющие заданному условию. (См. Предложение GROUP BY и Предложение HAVING ниже.) Хотя столбцы вывода запроса номинально вычисляются на следующем шаге, на них также можно ссылаться (по имени или порядковому номеру) в предложении `GROUP BY`.

5. Вычисляются фактические выходные строки по заданным в `SELECT` выражениям для каждой выбранной строки или группы строк. (См. Список SELECT ниже.)

6. `SELECT DISTINCT` исключает из результата повторяющиеся строки. `SELECT DISTINCT ON` исключает строки, совпадающие по всем указанным выражениям. `SELECT ALL` (по умолчанию) возвращает все строки результата, включая дубликаты. (См. Предложение DISTINCT ниже.)

7. Операторы `UNION`, `INTERSECT` и `EXCEPT` объединяют вывод нескольких команд `SELECT` в один результирующий набор. Оператор `UNION` возвращает все строки, представленные в одном, либо обоих наборах результатов. Оператор `INTERSECT` возвращает все строки, представленные строго в обоих наборах. Оператор `EXCEPT` возвращает все строки, представленные в первом наборе, но не во втором. Во всех трёх случаях повторяющиеся строки исключаются из результата, если явно не указано `ALL`. Чтобы явно обозначить, что выдаваться должны только неповторяющиеся строки, можно добавить избыточное слово `DISTINCT`. Заметьте, что в данном контексте по умолчанию подразумевается `DISTINCT`, хотя в самом `SELECT` по умолчанию подразумевается `ALL`. (См. Предложение UNION+, Предложение INTERSECT и Предложение EXCEPT ниже.)

8. Если присутствует предложение `ORDER BY`, возвращаемые строки сортируются в указанном порядке. В отсутствие `ORDER BY` строки возвращаются в том порядке, в каком системе будет проще их выдать. (См. Предложение ORDER BY> ниже.)

9. Если указано предложение `LIMIT` (или `FETCH FIRST`) либо `OFFSET`, оператор `SELECT` возвращает только подмножество строк результата. (См. Предложение LIMIT ниже.)

10. Если указано `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` или `FOR KEY SHARE`, оператор `SELECT` блокирует выбранные строки, защищая их от одновременных изменений. (См. Предложение блокировки ниже.)

Для всех столбцов, задействованных в команде `SELECT`, необходимо иметь право `SELECT`. Применение блокировок `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` или `FOR KEY SHARE` требует также права `UPDATE` (как минимум для одного столбца в каждой выбранной для блокировки таблице).

Параметры

Предложение `WITH`

Предложение `WITH` позволяет задать один или несколько подзапросов, к которым затем можно обратиться по имени в основном запросе. Эти подзапросы по сути действуют как временные таблицы или представления в процессе выполнения главного запроса. Каждый подзапрос может представлять собой оператор `SELECT`, `TABLE`, `VALUES`, `INSERT`, `UPDATE`, `DELETE` или `MERGE`. При использовании в `WITH` оператора, изменяющего данные, (`INSERT`, `UPDATE`, `DELETE` или `MERGE`) обычно добавляется предложение `RETURNING`. Заметьте, что именно результат `RETURNING`, а *не* нижележащая таблица, изменяемая запросом, формирует временную таблицу, которую затем читает основной запрос. Если `RETURNING` опущено, оператор тем не менее выполняется, но не выдаёт никакого результата, так что на него нельзя сослаться как на таблицу в основном запросе.

Имя (без схемы) должно быть указано для каждого запроса `WITH`. Также можно задать необязательный список с именами столбцов; если он опущен, имена столбцов формируются из результата подзапроса.

Если указано `RECURSIVE`, подзапрос `SELECT` может ссылаться сам на себя по имени. Такой подзапрос должен иметь форму

```
нерекурсивная_часть UNION [ ALL | DISTINCT ] рекурсивная_часть
```

, где рекурсивная ссылка на сам запрос может находиться только справа от `UNION`. Для одного запроса допускается только одна рекурсивная ссылка на него же. Операторы, изменяющие данные, не могут быть рекурсивными, но результат рекурсивного запроса `SELECT` в таких операторах можно использовать. За примером обратитесь к Разделу 7.8.

Ещё одна особенность `RECURSIVE` в том, что запросы `WITH` могут быть неупорядоченными: запрос может ссылаться на другой, идущий в списке после него. (Однако циклические ссылки или взаимная рекурсия не поддерживаются.) Без `RECURSIVE` запрос в `WITH` может ссылаться только на запросы того же уровня в `WITH`, предшествующие ему в списке `WITH`.

Когда в предложении `WITH` задаются несколько запросов, `RECURSIVE` следует указывать только единожды, сразу после `WITH`. Это указание будет действовать на все запросы в предложении `WITH`, хотя оно никак не скажется на запросах, не использующих рекурсию или ссылки на последующие запросы.

Необязательное предложение `SEARCH` вычисляет *столбец последовательности поиска*, который может применяться для упорядочивания результатов рекурсивного запроса «сначала в ширину» или «сначала в глубину». Передаваемый список имён столбцов задаёт ключ строки, по которому будет отслеживаться, какие строки были посещены. Столбец с именем *`имя_столбца_послед_поиска`* будет добавлен в список результирующих столбцов запроса `WITH`. Для получения нужного порядка можно выполнить сортировку по этому столбцу во внешнем запросе. За примерами обратитесь к Подразделу 7.8.2.1.

Необязательное предложение `CYCLE` применяется для обнаружения циклов в рекурсивных запросах. Задаваемый в нём список имён столбцов определяет ключ строки, по которому будут отслеживаться посещённые строки. Столбец *`имя_столбца_пометки_цикла`* будет добавлен в список результирующих столбцов запроса `WITH`. В случае обнаружения цикла в этом столбце окажется *`значение_пометки_цикла`*, а иначе — *`пометка_цикла_по_умолчанию`*. Более того, в случае обнаружения цикла обработка рекурсивного объединения будет остановлена. Задаваемые *`значение_пометки_цикла`* и *`пометка_цикла_по_умолчанию`* должны быть константами и должны приводиться к общему типу данных, а для этого типа должен быть определён оператор неравенства. (Стандарт SQL требует, чтобы они были булевыми константами или символьными строками, но в Postgres Pro это необязательно.) По умолчанию подразумеваются соответственно константы `TRUE` и `FALSE` (типа `boolean`). Кроме того, в список результирующих столбцов запроса `WITH` будет добавлен столбец *`имя_столбца_пути_цикла`*. Этот столбец имеет внутреннее предназначение, в нём отслеживаются посещённые строки. За примерами обратитесь к Подразделу 7.8.2.2.

Предложения `SEARCH` и `CYCLE` работают только с рекурсивными запросами `WITH`. При этом *`запрос_WITH`* должен быть объединением (`UNION` или `UNION ALL`) двух `SELECT` (или равнозначных команд) без вложенных `UNION`. В случае применения обоих предложений столбец, добавляемый предложением `SEARCH`, размещается перед столбцами, добавляемыми предложением `CYCLE`.

Основной запрос и все запросы `WITH`, условно говоря, выполняются одновременно. Это значит, что действие оператора, изменяющего данные в `WITH`, не будут видеть другие части запроса, кроме как прочитав его вывод `RETURNING`. Если два таких оператора попытаются изменить одну строку, результат будет неопределённым.

Ключевое свойство запросов `WITH` состоит в том, что они обычно вычисляются один раз для всего основного запроса, даже если в основном запросе содержатся несколько ссылок на них. В частности, гарантируется, что операторы, изменяющие данные, будут выполняться ровно один раз, вне зависимости от того, будет ли их результат прочитан основным запросом и в каком объёме.

Однако от этой гарантии можно отказаться, добавив для запроса `WITH` пометку `NOT MATERIALIZED`. В этом случае запрос `WITH` может быть свёрнут в основной запрос, как если бы это был простой `SELECT` внутри предложения `FROM` основного запроса. В результате запрос `WITH` может вычисляться неоднократно, если основной запрос обращается к нему несколько раз. Но если при каждом таком обращении требуются лишь отдельные строки из всего результата запроса `WITH`, указание `NOT MATERIALIZED`, позволяющее оптимизировать запросы совместно, в целом может оказаться выгодным. Указание `NOT MATERIALIZED` игнорируется в запросе `WITH`, который имеет рекурсивный характер или не свободен от побочных эффектов (то есть когда это не простой `SELECT` без изменчивых функций).

По умолчанию запрос `WITH`, свободный от побочных эффектов, заворачивается в основной запрос, если он используется в его предложении `FROM` ровно один раз. Это позволяет совместно оптимизировать два уровня запросов в ситуациях, когда семантика запроса в целом сохраняется. Но это заворачивание можно предотвратить, пометив запрос `WITH` как `MATERIALIZED`. Это может быть полезно, например когда предложение `WITH` применяется как преграда для оптимизатора, не позволяющая ему выбрать неудачный план. В Postgres Pro до 12 версии такое заворачивание не выполнялось никогда, поэтому запросы, написанные для предыдущих версий, могли полагаться на то, что оптимизации будет препятствовать собственно предложение `WITH`.

За дополнительными сведениями обратитесь к Разделу 7.8.

Предложение `FROM`

В предложении `FROM` перечисляются одна или несколько таблиц, служащих источниками данных для `SELECT`. Если указано несколько источников, результатом будет декартово произведение (перекрёстное соединение) всех их строк. Но обычно в запрос добавляются уточняющие условия (в предложении `WHERE`), которые ограничивают набор строк небольшим подмножеством этого произведения.

Предложение `FROM` может содержать следующие элементы:

`имя_таблицы`

Имя (возможно, дополненное схемой) существующей таблицы или представления. Если перед именем таблицы указано `ONLY`, считывается только заданная таблица. Без `ONLY` считывается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание `*`, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы.

псевдоним

Альтернативное имя для элемента списка `FROM`. Этот псевдоним используется для краткости или для исключения неоднозначности с замкнутыми соединениями (когда одна таблица читается неоднократно). Когда задаётся псевдоним, он полностью скрывает настоящее имя таблицы или функции; например, при записи `FROM foo AS f`, в продолжении запроса `SELECT` к этому элементу `FROM` нужно обращаться по имени `f`, а не `foo`. Если задан псевдоним таблицы, за ним можно также написать список псевдонимов столбцов, который определит альтернативные имена для столбцов таблицы.

`TABLESAMPLE` *`метод_выборки`* `(` *`аргумент`* `[, ...] ) [ REPEATABLE (` *`затравка`* `) ]`

Предложение `TABLESAMPLE`, сопровождающее *`имя_таблицы`*, показывает, что для получения подмножества строк в этой таблице должен применяться указанный *`метод_выборки`*. Эта выборка предшествует применению любых других фильтров, например, в предложении `WHERE`. В стандартный дистрибутив Postgres Pro включены два метода выборки, `BERNOULLI` и `SYSTEM`; другие методы выборки можно установить в базу данных через расширения.

Методы выборки `BERNOULLI` и `SYSTEM` принимают единственный *`аргумент`*, определяющий, какой процент таблицы должен попасть в выборку, от 0 до 100. Этот аргумент может задаваться любым выражением со значением типа `real`. (Другие методы выборки могут принимать дополнительные или другие параметры.) Оба этих метода возвращают случайную выборку таблицы, содержащую примерно указанный процент строк таблицы. Метод `BERNOULLI` сканирует всю таблицу и выбирает или игнорирует отдельные строки независимо, с заданной вероятностью. Метод `SYSTEM` строит выборку на уровне блоков, определяя для каждого блока шанс его задействовать, и возвращает все строки из каждого задействуемого блока. Метод `SYSTEM` работает значительно быстрее `BERNOULLI`, когда выбирается небольшой процент строк, но он может выдавать менее случайную выборку таблицы из-за эффектов кучности.

В необязательном предложении `REPEATABLE` задаётся *`затравка`* — число или выражение, задающее отправное значение для генератора случайных чисел в методе выборки. Значением затравки может быть любое отличное от NULL число с плавающей точкой. Два запроса, в которых указаны одинаковые значения затравки и *`аргумента`*, выдадут одну и ту же выборку таблицы при условии неизменности содержимого таблицы. Но с разными значениями затравки выборки обычно получаются разными. В отсутствие предложения `REPEATABLE` для каждого запроса выдаётся новая случайная выборка, в зависимости от затравки, сгенерированной системой. Заметьте, что некоторые дополнительные методы выборки не принимают предложение `REPEATABLE` и выдают разные выборки при каждом использовании.

выборка

Предложение `FROM` может содержать вложенный запрос `SELECT`. Можно считать, что из его результата создаётся временная таблица на время выполнения основной команды `SELECT`. Заметьте, что вложенный запрос `SELECT` должен заключаться в скобки и для него может задаваться псевдоним так же, как для таблицы. Здесь также можно использовать команду `VALUES`.

имя_запроса_WITH

На запрос `WITH` можно ссылаться по имени, как если бы имя запроса представляло имя таблицы. (На самом деле запрос `WITH` скрывает любую реальную таблицу с тем же именем для основного запроса. Если необходимо обратиться к одноимённой реальной таблице, можно дополнить имя этой таблицы именем схемы.) Для этого имени можно задать псевдоним, так же, как и для имени таблицы.

имя_функции

В предложении `FROM` могут содержаться вызовы функций. (Это особенно полезно для функций, возвращающих множества, но в принципе можно использовать любые функции.) Можно считать, что для результата функции создаётся временная таблица на время выполнения текущей команды `SELECT`. Если функция возвращает результат составного типа (это касается и функций с несколькими аргументами `OUT`), каждый атрибут результата помещается в отдельный столбец этой неявной таблицы.

Когда вызов функции дополняется необязательным предложением `WITH ORDINALITY`, к столбцам результата функции добавляется столбец типа `bigint`. В этом столбце нумеруются строки набора результатов функции, начиная с 1. По умолчанию ему присваивается имя `ordinality`.

Псевдоним для функции можно задать так же, как и для таблицы. Если этот псевдоним задан, за ним можно также написать список псевдонимов столбцов, который определит альтернативные имена для атрибутов составного типа результата функции, включая имя столбца нумерации (если он присутствует).

Несколько вызовов функций можно объединить в одном элементе предложения `FROM`, заключив их в конструкцию `ROWS FROM( ... )`. Выводом такого элемента будет соединение первых строк всех функций, затем вторых строк и т. д. Если одни функции выдают меньше строк, чем другие, недостающие данные заменяются значениями NULL, так что общее число возвращаемых строк всегда будет равняться максимальному числу строк из возвращённых всеми функциями.

Если функция определена как возвращающая тип данных `record`, для неё нужно указать псевдоним или ключевое слово `AS`, за которым должен идти список определений столбцов в форме `(` *`имя_столбца тип_данных`* `[, ... ])`. Список определений столбцов должен соответствовать фактическому количеству и типу столбцов, возвращаемых функцией.

Если при использовании синтаксиса `ROWS FROM( ... )` одна из функций требует наличия списка определений столбцов, этот список лучше разместить после вызова функции внутри `ROWS FROM( ... )`. Список определений столбцов можно поместить после конструкции `ROWS FROM( ... )`, только если вызывается всего одна функция, а предложение `WITH ORDINALITY` отсутствует.

Чтобы использовать `ORDINALITY` со списком определений столбцов, необходимо применить запись `ROWS FROM( ... )` и поместить список с определениями столбцов внутрь `ROWS FROM( ... )`.

`тип_соединения`

Один из следующих вариантов:

- `[ INNER ] JOIN`
- `LEFT [ OUTER ] JOIN`
- `RIGHT [ OUTER ] JOIN`
- `FULL [ OUTER ] JOIN`

Для типов соединений `INNER` и `OUTER` необходимо указать условие соединения, а именно одно из предложений `ON` *`условие_соединения`*, `USING (`*`столбец_соединения`* `[, ...])`, или `NATURAL`. Эти предложения описываются ниже.

Предложение `JOIN` объединяет два элемента списка `FROM`, которые мы для простоты дальше будем называть «таблицами», хотя на самом деле это может быть любой объект, допустимый в качестве элемента `FROM`. Для определения порядка вложенности при необходимости следует использовать скобки. В отсутствие скобок предложения `JOIN` обрабатывается слева направо. В любом случае `JOIN` связывает элементы сильнее, чем запятые, разделяющие элементы в списке `FROM`. Все возможности `JOIN` существуют просто для удобства записи, они не дают ничего такого, чего нельзя было бы получить, применяя обычные предложения `FROM` и `WHERE`.

`LEFT OUTER JOIN` возвращает все строки ограниченного декартова произведения (т. е. все объединённые строки, удовлетворяющие условию соединения) плюс все строки в таблице слева, для которых не находится строк в таблице справа, удовлетворяющих условию. Строка, взятая из таблицы слева, дополняется до полной ширины объединённой таблицы значениями NULL в столбцах таблицы справа. Заметьте, что для определения, какие строки двух таблиц соответствуют друг другу, проверяется только условие самого предложения `JOIN`. Внешние условия проверяются позже.

`RIGHT OUTER JOIN`, напротив, возвращает все соединённые строки плюс одну строку для каждой строки справа, не имеющей соответствия слева (эта строка дополняется значениями NULL влево). Это предложение введено исключительно для удобства записи, так как его можно легко свести к `LEFT OUTER JOIN`, поменяв левую и правую таблицы местами.

`FULL OUTER JOIN` возвращает все соединённые строки плюс все строки слева, не имеющие соответствия справа, (дополненные значениями NULL вправо) плюс все строки справа, не имеющие соответствия слева (дополненные значениями NULL влево).

`ON` *`условие_соединения`*

Задаваемое *`условие_соединения`* представляет собой выражение, выдающее значение типа `boolean` (как в предложении `WHERE`), которое определяет, какие строки считаются соответствующими при соединении.

`USING (` *`столбец_соединения`* `[, ...] ) [ AS` *`псевдоним_использования_соединения`* `]`

Предложение вида `USING ( a, b, ... )` представляет собой сокращённую форму записи `ON таблица_слева.a = таблица_справа.a AND таблица_слева.b = таблица_справа.b ....` Кроме того, `USING` подразумевает, что в результат соединения будет включён только один из пары равных столбцов, но не оба.

Если задан *`псевдоним_использования_соединения`*, он определяет псевдоним таблицы со столбцами соединения. По этому имени можно будет обращаться только к столбцам соединения, перечисленным в предложении `USING`. В отличие от обычного *`псевдонима`*, он не скрывает имена соединяемых таблиц от остального запроса. Также в отличие от *`псевдонима`*, с ним нельзя задать список псевдонимов столбцов — выходными именами столбцов соединения будут те же имена, что фигурируют в списке `USING`.

`NATURAL`

`NATURAL` представляет собой краткую запись `USING` со списком, в котором перечисляются все столбцы двух таблиц, имеющие одинаковые имена. Если одинаковых имён нет, указание `NATURAL` равнозначно `ON TRUE`.

`CROSS JOIN`

Предложение `CROSS JOIN` равнозначно `INNER JOIN ON (TRUE)`, то есть с ним никакие строки по условию не удаляются. Оба эти предложения формируют простое декартово произведение — тот же результат получится, если указать две таблицы на верхнем уровне `FROM` и добавить в качестве ограничения условие соединения (если оно есть).

`LATERAL`

Ключевое слово `LATERAL` может предварять вложенный запрос `SELECT` в списке `FROM`. Оно позволяет обращаться в этом вложенном `SELECT` к столбцам элементов `FROM`, предшествующим ему в списке `FROM`. (Без `LATERAL` все вложенные подзапросы `SELECT` обрабатываются независимо и не могут ссылаться на другие элементы списка `FROM`.)

Слово `LATERAL` можно также добавить перед вызовом функции в списке `FROM`, но в этом случае оно будет избыточным, так как выражения с функциями могут ссылаться на предыдущие элементы списка `FROM` в любом случае.

Элемент `LATERAL` может находиться на верхнем уровне списка `FROM` или в дереве `JOIN`. В последнем случае он может также ссылаться на любые элементы в левой части `JOIN`, справа от которого он находится.

Когда элемент `FROM` содержит ссылки `LATERAL`, запрос выполняется следующим образом: сначала для строки элемента `FROM` с целевыми столбцами, или набора строк из нескольких элементов `FROM`, содержащих целевые столбцы, вычисляется элемент `LATERAL` со значениями этих столбцов. Затем результирующие строки обычным образом соединяются со строками, из которых они были вычислены. Эта процедура повторяется для всех строк исходных таблиц.

Таблица, служащая источником столбцов, должна быть связана с элементом `LATERAL` соединением `INNER` или `LEFT`, в противном случае не образуется однозначно определяемый набор строк, из которого можно будет получать наборы строк для элемента `LATERAL`. Таким образом, хотя конструкция `X RIGHT JOIN LATERAL Y` синтаксически правильная, `Y` в ней не может обращаться к `X`.

Предложение `WHERE`

Необязательное предложение `WHERE` имеет общую форму

```
WHERE условие
```

, где *`условие`* — любое выражение, выдающее результат типа `boolean`. Любая строка, не удовлетворяющая этому условию, исключается из результата. Строка удовлетворяет условию, если оно возвращает true при подстановке вместо ссылок на переменные фактических значений из этой строки.

Предложение `GROUP BY`

Необязательное предложение `GROUP BY` имеет общую форму

```
GROUP BY [ ALL | DISTINCT ] элемент_группирования [, ...]
```

`GROUP BY` собирает в одну строку все выбранные строки, выдающие одинаковые значения для выражений группировки. В качестве *`выражения`* внутри *`элемента_группирования`* может выступать имя входного столбца, либо имя или порядковый номер выходного столбца (из списка элементов `SELECT`), либо произвольное значение, вычисляемое по значениям входных столбцов. В случае неоднозначности имя в `GROUP BY` будет восприниматься как имя входного, а не выходного столбца.

Если в элементе группирования задаётся `GROUPING SETS`, `ROLLUP` или `CUBE`, предложение `GROUP BY` в целом определяет некоторое число независимых *`наборов группирования`*. Это даёт тот же эффект, что и объединение подзапросов (с `UNION ALL`) с отдельными наборами группирования в их предложениях `GROUP BY`. Необязательное предложение `DISTINCT` исключает повторяющиеся наборы перед обработкой, но оно *не* преобразует `UNION ALL` в `UNION DISTINCT`. Подробнее использование наборов группирования описывается в Подразделе 7.2.4.

Агрегатные функции, если они используются, вычисляются по всем строкам, составляющим каждую группу, и в итоге выдают отдельное значение для каждой группы. (Если агрегатные функции используются без предложения `GROUP BY`, запрос выполняется как с одной группой, включающей все выбранные строки.) Набор строк, поступающих в каждую агрегатную функцию, можно дополнительно отфильтровать, добавив предложение `FILTER` к вызову агрегатной функции; за дополнительными сведениями обратитесь к Подразделу 4.2.7. С предложением `FILTER` на вход агрегатной функции поступают только те строки, которые соответствуют заданному фильтру.

Когда в запросе присутствует предложение `GROUP BY` или какая-либо агрегатная функция, выражения в списке `SELECT` не могут обращаться к негруппируемым столбцам, кроме как в агрегатных функциях или в случае функциональной зависимости, так как иначе в негруппируемом столбце нужно было бы вернуть более одного возможного значения. Функциональная зависимость образуется, если группируемые столбцы (или их подмножество) составляют первичный ключ таблицы, содержащей негруппируемый столбец.

Имейте в виду, что все агрегатные функции вычисляются перед «скалярными» выражениями в предложении `HAVING` или списке `SELECT`. Это значит, что например, с помощью выражения `CASE` нельзя обойти вычисление агрегатной функции; см. Подраздел 4.2.14.

В настоящее время указания `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE` нельзя задать вместе с `GROUP BY`.

Предложение `HAVING`

Необязательное предложение `HAVING` имеет общую форму

```
HAVING условие
```

Здесь *`условие`* задаётся так же, как и для предложения `WHERE`.

`HAVING` исключает из результата строки групп, не удовлетворяющих условию. `HAVING` отличается от `WHERE`: `WHERE` фильтрует отдельные строки до применения `GROUP BY`, а `HAVING` фильтрует строки групп, созданных предложением `GROUP BY`. Каждый столбец, фигурирующий в *`условии`*, должен однозначно ссылаться на группируемый столбец, за исключением случаев, когда эта ссылка находится внутри агрегатной функции или негруппируемый столбец функционально зависит от группируемых.

В присутствие `HAVING` запрос превращается в группируемый, даже если `GROUP BY` отсутствует. То же самое происходит, когда запрос содержит агрегатные функции, но не предложение `GROUP BY`. Все выбранные строки считаются формирующими одну группу, а в списке `SELECT` и предложении `HAVING` можно обращаться к столбцам таблицы только из агрегатных функций. Такой запрос будет выдавать единственную строку, если результат условия `HAVING` — true, и ноль строк в противном случае.

В настоящее время указания `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE` нельзя задать вместе с `HAVING`.

Предложение `WINDOW`

Необязательное предложение `WINDOW` имеет общую форму

```
WINDOW имя_окна AS ( определение_окна ) [, ...]
```

Здесь *`имя_окна`* — это имя, на которое можно ссылаться из предложений `OVER` или последующих определений окон, а *`определение_окна`* имеет следующий вид:

```
[ имя_существующего_окна ]
[ PARTITION BY выражение [, ...] ]
[ ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ]
  [, ...] ]
[ предложение_рамки ]
```

Если указано *`имя_существующего_окна`*, оно должно ссылаться на предшествующую запись в списке `WINDOW`; новое окно копирует предложение секционирования из этой записи, а также предложение сортировки, если оно присутствует. В этом случае для нового окна нельзя задать собственное предложение `PARTITION BY`, а `ORDER BY` можно указать, только если его не было у копируемого окна. Новое окно всегда использует собственное предложение рамки; в копируемом окне оно задаваться не должно.

Элементы списка `PARTITION BY` интерпретируется во многом так же, как и элементы списка `GROUP BY`, за исключением того, что это всегда простые выражения, но не имя или номер выходного столбца. Другое различие состоит в том, что эти выражения могут содержать вызовы агрегатных функций, которые не допускаются в обычном предложении `GROUP BY`. Здесь они допускаются потому, что формирование окна происходит после группировки и агрегирования.

Подобным образом, элементы списка `ORDER BY` интерпретируются во многом так же, как и элементы предложения `ORDER BY` на уровне оператора, за исключением того, что они всегда воспринимаются как простые выражения, но не как имя или номер выходного столбца.

Необязательное *`предложение_рамки`* определяет *рамку окна* для оконных функций, которые зависят от рамки (не все функции таковы). Рамка окна — это набор связанных строк для каждой строки запроса (называемой *текущей строкой*). В качестве *`предложения_рамки`* может задаваться

```
{ RANGE | ROWS | GROUPS } начало_рамки [ исключение_рамки ]
```

```
{ RANGE | ROWS | GROUPS } BETWEEN начало_рамки AND конец_рамки [ исключение_рамки ]
```

Здесь *начало_рамки* и *конец_рамки* может задаваться как

```
UNBOUNDED PRECEDING
смещение PRECEDING
CURRENT ROW
смещение FOLLOWING
UNBOUNDED FOLLOWING
```

и *исключение_рамки* может быть таким:

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

Если *конец_рамки* опущен, по умолчанию подразумевается `CURRENT ROW`. В качестве *начала_рамки* нельзя задать `UNBOUNDED FOLLOWING`, в качестве *конца_рамки* не допускается `UNBOUNDED PRECEDING`, и *конец_рамки* не может идти в показанном выше списке указаний *начало_рамки* AND *конец_рамки* перед *началом_рамки* — например, синтаксис `RANGE BETWEEN CURRENT ROW AND` *смещение* `PRECEDING` не допускается.

По умолчанию рамка образуется предложением `RANGE UNBOUNDED PRECEDING`, что по сути то же, что `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`; оно устанавливает рамку так, что она включает все строки от начала раздела до последней строки, *родственной* текущей (строки, которые согласно указанному для окна предложению `ORDER BY` считаются равными текущей; если `ORDER BY` отсутствует, все строки считаются родственными). Вообще `UNBOUNDED PRECEDING` означает, что рамка начинается с первой строки раздела, а `UNBOUNDED FOLLOWING` означает, что рамка заканчивается на последней строке раздела, независимо от режима `RANGE`, `ROWS` или `GROUPS`. В режиме `ROWS` указание `CURRENT ROW` означает, что рамка начинается или заканчивается текущей строкой; но в режиме `RANGE` или `GROUPS` оно означает, что рамка начинается или заканчивается первой или последней строкой, родственной текущей, согласно порядку `ORDER BY`. Варианты *смещение* `PRECEDING` и *смещение* `FOLLOWING` означают разное в зависимости от режима рамки. В режиме `ROWS` целочисленное *смещение* определяет сдвиг, с которым начало рамки позиционируется перед текущей строкой, а конец рамки — после текущей строки. В режиме `GROUPS` целочисленное *смещение* аналогичным образом определяет сдвиг относительно группы строк, родственных текущей, где *группа родственных строк* — группа строк, считающихся равными согласно предложению `ORDER BY` для данного окна. В режиме `RANGE` для указания *смещения* необходимо присутствие в определении окна ровно одного столбца `ORDER BY`. Тогда рамка будет содержать те строки, в которых значение упорядочивающего столбца не более чем на *смещение* меньше (для `PRECEDING`) или больше (для `FOLLOWING`) значения упорядочивающего столбца в текущей строке. В этом случае тип данных выражения *смещение* зависит от типа данных упорядочивающего столбца. Для числовых столбцов это обычно тот же числовой тип, а для столбцов с типом дата/время — тип `interval`. Во всех этих случаях значение *смещения* должно быть отличным от NULL и неотрицательным. Кроме того, хотя *смещение* не обязательно должно быть простой константой, оно не может содержать переменные, агрегатные или оконные функции.

Дополнение *исключение_рамки* позволяет исключить из рамки строки, которые окружают текущую строку, даже если они должны быть включены согласно указаниям, определяющим начало и конец рамки. `EXCLUDE CURRENT ROW` исключает из рамки текущую строку. `EXCLUDE GROUP` исключает из рамки текущую строку и родственные ей согласно порядку сортировки. `EXCLUDE TIES` исключает из рамки все родственные строки для текущей, но не собственно текущую строку. `EXCLUDE NO OTHERS` просто явно выражает поведение по умолчанию — не исключает ни текущую строку, ни родственные ей.

Учтите, что в режиме `ROWS` могут выдаваться непредсказуемые результаты, если согласно порядку, заданному в `ORDER BY`, строки сортируются неоднозначно. Режимы `RANGE` и `GROUPS` предусмотрены для того, чтобы строки, являющиеся родственными в порядке `ORDER BY`, обрабатывались одинаково: все строки определённой группы попадут в одну рамку или будут исключены из неё.

Предложение `WINDOW` применяется для управления поведением *оконных функций*, фигурирующих в запросе, в списке `SELECT` или предложении `ORDER BY`. Эти функции могут обращаться к элементам `WINDOW` по именам в своих предложениях `OVER`. При этом элементы `WINDOW` не обязательно задействовать в запросе; если они не используются, они просто игнорируются. Оконные функции можно использовать вовсе без элементов `WINDOW`, так как в вызове оконной функции можно задать определение окна непосредственно в предложении `OVER`. Однако предложение `WINDOW` позволяет сократить текст запроса, когда одно и то же определение окна применяется при вызове нескольких оконных функций.

В настоящее время указания `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE` нельзя задать вместе с `WINDOW`.

Оконные функции подробно описываются в Разделе 3.5, Подразделе 4.2.8 и Подразделе 7.2.5.

Список SELECT

Список `SELECT` (между ключевыми словами `SELECT` и `FROM`) содержит выражения, которые формируют выходные строки оператора `SELECT`. Эти выражения могут обращаться (и обычно обращаются) к столбцам, вычисленным в предложении `FROM`.

Так же, как в таблице, каждый выходной столбец `SELECT` имеет имя. В простом предложении `SELECT` это имя просто помечает столбец при выводе, но когда `SELECT` представляет собой подзапрос большого запроса, это имя большой запрос видит как имя столбца виртуальной таблицы, созданной подзапросом. Чтобы задать имя для выходного столбца, нужно написать `AS` *`выходное_имя`* после выражения столбца. (Слово `AS` можно опустить, но только если желаемое выходное имя не совпадает с каким-либо ключевым словом Postgres Pro (см. Приложение C). Чтобы не зависеть от появления новых ключевых слов в будущем, рекомендуется всегда писать `AS`, либо заключать имя в двойные кавычки.) Если имя столбца не задать, Postgres Pro выберет его автоматически. Если выражение столбца представляет собой просто ссылку на столбец, то выбранное таким образом имя будет совпадать с именем столбца. В более сложных случаях может использоваться имя функции или типа, либо в отсутствие других вариантов система может сгенерировать имя вроде `?column?`.

По имени выходного столбца можно обратиться к его значению в предложениях `ORDER BY` и `GROUP BY`, но не в `WHERE` или `HAVING`; в них вместо имени надо записывать всё выражение.

Вместо выражения в выходном списке можно указать `*`, что будет обозначать все столбцы выбранных строк. Кроме того, можно записать *`имя_таблицы`*`.*` как краткое обозначение всех столбцов, получаемых из данной таблицы. В этих случаях нельзя задать новые имена столбцов с помощью `AS`; именами выходных столбцов будут имена столбцов в таблице.

Согласно стандарту SQL, выражения в выходном списке должны вычисляться до применения `DISTINCT`, `ORDER BY` или `LIMIT`. Это, очевидно, необходимо для `DISTINCT`, так как иначе не будет ясно, какие значения должны выдаваться как уникальные. Однако во многих случаях выходные выражения удобнее вычислять после `ORDER BY` и `LIMIT`; в частности, если в выходном списке содержатся изменчивые или дорогостоящие функции. В этом случае порядок вычисления функций оказывается более интуитивным, а для строк, которые не попадут в результат, не будут производиться вычисления. Postgres Pro фактически будет вычислять выходные выражения после сортировки и ограничения их количества, если эти выражения не фигурируют в `DISTINCT`, `ORDER BY` или `GROUP BY`. (Например, в запросе `SELECT f(x) FROM tab ORDER BY 1` функция `f(x)`, несомненно, должна вычисляться перед сортировкой.) Выходные выражения, содержащие функции, возвращающие множества, фактически вычисляются после сортировки и до ограничения количества строк, так что `LIMIT` будет отбрасывать строки, выдаваемые функцией, возвращающей множество.

Примечание

В Postgres Pro до версии 9.6 никакой порядок вычисления выходных выражений по отношению к сортировке или ограничениям количества не гарантировался; он зависел от формы выбранного плана запроса.

Предложение `DISTINCT`

Если указано `SELECT DISTINCT`, все повторяющиеся строки исключаются из результирующего набора (из каждой группы дубликатов остаётся одна строка). `SELECT ALL` делает противоположное: сохраняет все строки; это поведение по умолчанию.

`SELECT DISTINCT ON (` *`выражение`* `[, ...] )` сохраняет только первую строку из каждого набора строк, для которого данное выражение даёт одинаковые значения. Выражения `DISTINCT ON` обрабатываются по тем же правилам, что и выражения `ORDER BY` (см. выше). Заметьте, что «первая строка» каждого набора непредсказуема, если только не применяется предложение `ORDER BY`, определяющее, какие строки должны быть первыми. Например:

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

возвращает самую последнюю сводку погоды для каждого местоположения. Но если бы мы не добавили `ORDER BY`, чтобы значения времени убывали, мы бы получили сводки по местоположениям от непредсказуемого времени.

Выражения `DISTINCT ON` должны соответствовать самым левым выражениям в `ORDER BY`. Предложение `ORDER BY` обычно содержит и другие выражения, которые определяют желаемый порядок строк в каждой группе `DISTINCT ON`.

В настоящее время указания `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE` нельзя задать вместе с `DISTINCT`.

Предложение `UNION`+

Предложение `UNION` имеет следующую общую форму:

`оператор_SELECT` `UNION [ ALL | DISTINCT ]` *`оператор_SELECT`*

Здесь *`оператор_SELECT`* — это любой подзапрос `SELECT` без предложений `ORDER BY`, `LIMIT`, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE`. (`ORDER BY` и `LIMIT` можно добавить к вложенному выражению, если оно заключено в скобки. Без скобок эти предложения будут восприняты как применяемые к результату `UNION`, а не к выражению в его правой части.)

Оператор `UNION` вычисляет объединение множеств всех строк, возвращённых заданными запросами `SELECT`. Строка оказывается в объединении двух наборов результатов, если она присутствует минимум в одном наборе. Два оператора `SELECT`, представляющие прямые операнды `UNION`, должны выдавать одинаковое число столбцов, а типы соответствующих столбцов должны быть совместимыми.

Результат `UNION` не будет содержать повторяющихся строк, если не указан параметр `ALL`. `ALL` предотвращает исключение дубликатов. (Таким образом, `UNION ALL` обычно работает значительно быстрее, чем `UNION`; поэтому, везде, где возможно, следует указывать `ALL`.) `DISTINCT` можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе `SELECT` нескольких операторов `UNION` они вычисляются слева направо, если иной порядок не определяется скобками.

В настоящее время указания `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE` нельзя задать ни для результата `UNION`, ни для любого из подзапросов `UNION`.

Предложение `INTERSECT`

Предложение `INTERSECT` имеет следующую общую форму:

`оператор_SELECT` `INTERSECT [ ALL | DISTINCT ]` *`оператор_SELECT`*

Здесь *оператор_SELECT* — это любой подзапрос `SELECT` без предложений `ORDER BY`, `LIMIT`, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE`.

Оператор `INTERSECT` вычисляет пересечение множеств всех строк, возвращённых заданными запросами `SELECT`. Строка оказывается в пересечении двух наборов результатов, если она присутствует в обоих наборах.

Результат `INTERSECT` не будет содержать повторяющихся строк, если не указан параметр `ALL`. С параметром `ALL` строка, повторяющаяся *m* раз в левой таблице и *n* раз в правой, будет выдана в результирующем наборе min(*m*,*n*) раз. `DISTINCT` можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе `SELECT` нескольких операторов `INTERSECT` они вычисляются слева направо, если иной порядок не диктуется скобками. `INTERSECT` связывает свои подзапросы сильнее, чем `UNION`. Другими словами, `A UNION B INTERSECT C` будет восприниматься как `A UNION (B INTERSECT C)`.

В настоящее время указания `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE` нельзя задать ни для результата `INTERSECT`, ни для любого из подзапросов `INTERSECT`.

Предложение `EXCEPT`

Предложение `EXCEPT` имеет следующую общую форму:

```
оператор_SELECT EXCEPT [ ALL | DISTINCT ] оператор_SELECT
```

Здесь *оператор_SELECT* — это любой подзапрос `SELECT` без предложений `ORDER BY`, `LIMIT`, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE`.

Оператор `EXCEPT` вычисляет набор строк, которые присутствуют в результате левого запроса `SELECT`, но отсутствуют в результате правого.

Результат `EXCEPT` не будет содержать повторяющихся строк, если не указан параметр `ALL`. С параметром `ALL` строка, повторяющаяся *m* раз в левой таблице и *n* раз в правой, будет выдана в результирующем наборе max(*m*-*n*,0) раз. `DISTINCT` можно записать явно, чтобы обозначить, что дублирующиеся строки должны удаляться (это поведение по умолчанию).

При использовании в одном запросе `SELECT` нескольких операторов `EXCEPT` они вычисляются слева направо, если иной порядок не диктуется скобками. `EXCEPT` связывает свои подзапросы так же сильно, как `UNION`.

В настоящее время указания `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` и `FOR KEY SHARE` нельзя задать ни для результата `EXCEPT`, ни для любого из подзапросов `EXCEPT`.

Предложение `ORDER BY`

Необязательное предложение `ORDER BY` имеет следующую общую форму:

```
ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ] [, ...]
```

Предложение `ORDER BY` указывает, что строки результата должны сортироваться согласно заданным выражениям. Если две строки дают равные значения для самого левого выражения, проверяется следующее выражение и т. д. Если их значения оказываются равными для всех заданных выражений, строки возвращаются в порядке, определяемом реализацией.

В качестве *выражения* может задаваться имя или порядковый номер выходного столбца (элемента списка `SELECT`), либо произвольное выражение со значениями входных столбцов.

Порядковым номером в данном случае считается последовательный номер (при нумерации слева направо) позиции выходного столбца. Возможность указать порядковый номер позволяет выполнить сортировку по столбцу, не имеющему уникального имени. В принципе это не абсолютно необходимо, так как выходному столбцу всегда можно присвоить имя, воспользовавшись предложением `AS`.

В предложении `ORDER BY` также можно использовать произвольные выражения, в том числе, и со столбцами, отсутствующими в списке результатов `SELECT`. Таким образом, следующий оператор вполне корректен:

```
SELECT name FROM distributors ORDER BY code;
```

Однако если `ORDER BY` применяется к результату `UNION`, `INTERSECT` или `EXCEPT`, в нём можно задать только имя или номер выходного столбца, но не выражение.

Если в качестве выражения `ORDER BY` задано простое имя, которому соответствует и выходной, и входной столбец, то `ORDER BY` будет воспринимать его как имя выходного столбца. Этот выбор противоположен тому, что делает `GROUP BY` в такой же ситуации. Такая несогласованность допущена для соответствия стандарту SQL.

Дополнительно после любого выражения в предложении `ORDER BY` можно добавить ключевое слово `ASC` (по возрастанию) или `DESC` (по убыванию). По умолчанию подразумевается `ASC`. Кроме того, можно задать имя специфического оператора сортировки в предложении `USING`. Оператор сортировки должен быть членом «меньше» или «больше» некоторого семейства операторов B-дерева. `ASC` обычно равнозначно `USING <` и `DESC` обычно равнозначно `USING >`. (Хотя создатель нестандартного типа данных может определить по-другому порядок сортировки по умолчанию и поставить ему в соответствие операторы с другими именами.)

Если указано `NULLS LAST`, значения NULL при сортировке оказываются после значений не NULL; с указанием `NULLS FIRST` значения NULL оказываются перед значениями не NULL. Если не указано ни то, ни другое, по умолчанию подразумевается `NULLS LAST` при явно или неявно выбранном порядке `ASC`, либо `NULLS FIRST` при порядке `DESC` (то есть по умолчанию считается, что значения NULL больше значений не NULL). С предложением `USING` порядок NULL по умолчанию зависит от того, является ли указанный оператор оператором «меньше» или «больше».

Заметьте, что параметры сортировки применяются только к тому выражению, за которым они следуют; в частности, `ORDER BY x, y DESC` означает не то же самое, что `ORDER BY x DESC, y DESC`.

Данные символьных строк сортируются согласно правилу сортировки, установленному для сортируемого столбца. При необходимости это правило можно переопределить, добавив предложение `COLLATE` в *выражение*, например так: `ORDER BY mycolumn COLLATE "en_US"`. За дополнительными сведениями обратитесь к Подразделу 4.2.10 и Разделу 22.2.

Предложение `LIMIT`

Предложение `LIMIT` состоит из двух независимых вложенных предложений:

```
LIMIT { число | ALL }
OFFSET начало
```

Параметр *число* определяет максимальное количество строк, которое должно быть выдано, тогда как *начало* определяет, сколько строк нужно пропустить, прежде чем начать выдавать строки. Когда указаны оба значения, сначала строки пропускаются в количестве, заданном значением *начало*, а затем следующие строки выдаются в количестве, не превышающем значения *число*.

Если результатом выражения *число* оказывается NULL, предложение воспринимается как `LIMIT ALL`, т. е. число строк не ограничивается. Если *начало* принимает значение NULL, предложение воспринимается как `OFFSET 0`.

SQL:2008 вводит другой синтаксис для получения того же результата, и его так же поддерживает Postgres Pro. Он выглядит так:

```
OFFSET начало { ROW | ROWS }
FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } { ONLY | WITH TIES }
```

В этом синтаксисе значение *начало* или *число* в соответствии со стандартом должно быть буквальной константой, параметром или именем переменной; Postgres Pro позволяет использовать и другие выражения, но их обычно нужно заключать в скобки во избежание неоднозначности. Если *`число`* опускается в предложении `FETCH`, оно принимает значение 1. С указанием `WITH TIES` будут возвращены дополнительные строки, с точки зрения `ORDER BY` совпадающие с последней строкой набора результатов; в этом случае предложение `ORDER BY` обязательно, а `SKIP LOCKED` — недопустимо. Слова `ROW` и `ROWS`, а также `FIRST` и `NEXT` являются незначащими и не влияют на поведение этих предложений. Согласно стандарту предложение `OFFSET` должно идти перед `FETCH`, если они присутствуют вместе; но Postgres Pro менее строг и допускает любой порядок.

Применяя `LIMIT`, имеет смысл использовать также предложение `ORDER BY`, чтобы строки результата выдавались в определённом порядке. Иначе будут возвращаться непредсказуемые подмножества строк запроса — вы можете запросить строки с десятой по двадцатую, но какой порядок вы имеете в виду? Порядок будет неизвестен, если не добавить `ORDER BY`.

Планировщик запроса учитывает ограничение `LIMIT`, строя план выполнения запроса, поэтому, вероятнее всего, планы (а значит и порядок строк) будут меняться при разных `LIMIT` и `OFFSET`. Таким образом, различные значения `LIMIT`/`OFFSET`, выбирающие разные подмножества результатов запроса, *приведут к несогласованности результатов*, если не установить предсказуемую сортировку с помощью `ORDER BY`. Это не ошибка, а неизбежное следствие того, что SQL не гарантирует вывод результатов запроса в некотором порядке, если порядок не определён явно предложением `ORDER BY`.

Возможно даже, что при повторном выполнении одного и того же запроса с `LIMIT` будут получены разные подмножества строк таблицы, если предложение `ORDER BY` не диктует выбор определённого подмножества. Опять же, это не ошибка; в данном случае детерминированность результата просто не гарантируется.

Предложение блокировки

Предложения блокировки включают в себя `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` и `FOR KEY SHARE`; они влияют на то, как `SELECT` блокирует строки, получаемые из таблицы.

Предложение блокировки имеет следующую общую форму:

```
FOR вариант_блокировки [ OF имя_таблицы [, ...] ] [ NOWAIT | SKIP LOCKED ]
```

Здесь *`вариант_блокировки`* может быть следующим:

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

`имя_таблицы` должно представлять собой *`псевдоним`* или нескрытое *`имя_таблицы`*, используемое в предложении `FROM`. Подробнее о каждом режиме блокировки на уровне строк можно узнать в Подразделе 13.3.2.

Чтобы операция не ждала завершения других транзакций, к блокировке можно добавить указание `NOWAIT` или `SKIP LOCKED`. С `NOWAIT` оператор выдаёт ошибку, а не ждёт, если выбранную строку нельзя заблокировать немедленно. С указанием `SKIP LOCKED` выбранные строки, которые нельзя заблокировать немедленно, пропускаются. При этом формируется несогласованное представление данных, так что этот вариант не подходит для общего применения, но может использоваться для исключения блокировок при обращении множества потребителей к таблице типа очереди. Заметьте, что указания `NOWAIT` и `SKIP LOCKED` применяются только к блокировкам на уровне строк — необходимая блокировка `ROW SHARE` уровня таблицы запрашивается обычным способом (см. Главу 13). Если требуется запросить блокировку уровня таблицы без ожидания, можно сначала выполнить `LOCK` с указанием `NOWAIT`.

Если в предложении блокировки указаны определённые таблицы, блокироваться будут только строки, получаемые из этих таблиц; другие таблицы, задействованные в `SELECT`, будут прочитаны как обычно. Предложение блокировки без списка таблиц затрагивает все таблицы, задействованные в этом операторе. Если предложение блокировки применяется к представлению или подзапросу, оно затрагивает все таблицы, которые используются в представлении или подзапросе. Однако эти предложения не применяются к запросам `WITH`, к которым обращается основной запрос. Если требуется установить блокировку строк в запросе `WITH`, предложение блокировки нужно указать непосредственно в этом запросе `WITH`.

В случае необходимости задать для разных таблиц разное поведение блокировки, в запрос можно добавить несколько предложений. Если при этом одна и та же таблица упоминается (или неявно затрагивается) в нескольких предложениях блокировки, блокировка устанавливается так, как если бы было указано только одно, самое сильное из них. Подобным образом, если в одном из предложений указано `NOWAIT`, для этой таблицы блокировка будет запрашиваться без ожидания. В противном случае она будет обработана в режиме `SKIP LOCKED`, если он выбран в любом из затрагивающих её предложений.

Предложения блокировки не могут применяться в контекстах, где возвращаемые строки нельзя чётко связать с отдельными строками таблицы; например, блокировка неприменима при агрегировании.

Когда предложение блокировки находится на верхнем уровне запроса `SELECT`, блокируются именно те строки, которые возвращаются запросом; в случае с запросом объединения, блокировке подлежат строки, из которых составляются возвращаемые строки объединения. В дополнение к этому, заблокированы будут строки, удовлетворяющие условиям запроса на момент создания снимка запроса, хотя они не будут возвращены, если с момента снимка они изменятся и перестанут удовлетворять условиям. Если применяется `LIMIT`, блокировка прекращается, как только будет получено достаточное количество строк для удовлетворения лимита (но заметьте, что строки, пропускаемые указанием `OFFSET`, будут блокироваться). Подобным образом, если предложение блокировки применяется в запросе курсора, блокироваться будут только строки, фактически полученные или пройденные курсором.

Когда предложение блокировки находится в подзапросе `SELECT`, блокировке подлежат те строки, которые будет получены внешним запросом от подзапроса. Таких строк может оказаться меньше, чем можно было бы предположить, проанализировав только сам подзапрос, так как условия из внешнего запроса могут способствовать оптимизации выполнения подзапроса. Например, запрос

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

заблокирует только строки, в которых `col1 = 5`, при том, что в такой записи условие не относится к подзапросу.

Предыдущие версии не могли сохранить блокировку, которая была повышена последующей точкой сохранения. Например, этот код:

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

не мог сохранить блокировку `FOR UPDATE` после `ROLLBACK TO`. Это было исправлено в версии 9.3.

Внимание

Возможно, что команда `SELECT`, работающая на уровне изоляции `READ COMMITTED` и применяющая предложение `ORDER BY` вместе с блокировкой, будет возвращать строки не по порядку. Это связано с тем, что `ORDER BY` выполняется в первую очередь. Эта команда отсортирует результат, но затем может быть заблокирована, пытаясь получить блокировку одной или нескольких строк. К моменту, когда блокировка `SELECT` будет снята, некоторые из сортируемых столбцов могут уже измениться, в результате чего их порядок может быть нарушен (хотя они были упорядочены для исходных значений). При необходимости обойти эту проблему, можно поместить `FOR UPDATE/SHARE` в подзапрос, например так:

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;
```

Заметьте, что в результате это приведёт к блокированию всех строк в `mytable`, тогда как указание `FOR UPDATE` на верхнем уровне могло бы заблокировать только фактически возвращаемые строки. Это может значительно повлиять на производительность, особенно в сочетании `ORDER BY` с `LIMIT` или другими ограничениями. Таким образом, этот приём рекомендуется, только если ожидается параллельное изменение сортируемых столбцов, а результат должен быть строго отсортирован.

На уровнях изоляции `REPEATABLE READ` и `SERIALIZABLE` это приведёт к ошибке сериализации (с `SQLSTATE '40001'`), так что на этих уровнях получить строки не по порядку невозможно.

Команда `TABLE`

Команда

```
TABLE имя
```

равнозначна

```
SELECT * FROM имя
```

Её можно применять в качестве команды верхнего уровня или как более краткую запись внутри сложных запросов. С командой `TABLE` могут использоваться только предложения `WITH`, `UNION`, `INTERSECT`, `EXCEPT`, `ORDER BY`, `LIMIT`, `OFFSET`, `FETCH` и предложения блокировки `FOR`; предложение `WHERE` и какие-либо формы агрегирования не поддерживаются.

Примеры

Соединение таблицы `films` с таблицей `distributors`:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d JOIN films f USING (did);

       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

Суммирование значений столбца `len` (продолжительность) для всех фильмов и группирование результатов по столбцу `kind` (типу фильма):

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;

   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

Суммирование значений столбца `len` для всех фильмов, группирование результатов по столбцу `kind` и вывод только тех групп, общая продолжительность которых меньше 5 часов:

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';

   kind   | total
----------+-------
```

```
 Comedy   | 02:58
 Romantic | 04:38
```

Следующие два запроса демонстрируют равнозначные способы сортировки результатов по содержимому второго столбца (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;

 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

Следующий пример показывает объединение таблиц `distributors` и `actors`, ограниченное именами, начинающимися с буквы W в каждой таблице. Интерес представляют только неповторяющиеся строки, поэтому ключевое слово `ALL` опущено.

```
distributors:               actors:
 did |     name              id |     name
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                         ...

SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

Этот пример показывает, как использовать функцию в предложении `FROM`, со списком определений столбцов и без него:

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;
```

```
SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

Пример функции с добавленным столбцом нумерации:

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH ORDINALITY;
 unnest | ordinality
--------+----------
 a      |        1
 b      |        2
 c      |        3
 d      |        4
 e      |        5
 f      |        6
(6 rows)
```

Этот пример показывает, как использовать простое предложение `WITH`:

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t;
         x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

Заметьте, что запрос `WITH` выполняется всего один раз, поэтому мы получаем два одинаковых набора по три случайных значения.

В этом примере `WITH RECURSIVE` применяется для поиска всех подчинённых Мери (непосредственных или косвенных) и вывода их уровня косвенности в таблице с информацией только о непосредственных подчинённых:

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
```

```
SELECT distance, employee_name FROM employee_recursive;
```

Заметьте, что это типичная форма рекурсивных запросов: начальное условие, последующий `UNION`, а затем рекурсивная часть запроса. Убедитесь в том, что рекурсивная часть запроса в конце концов перестанет возвращать строки, иначе запрос окажется в бесконечном цикле. (За другими примерами обратитесь к Разделу 7.8.)

В этом примере используется `LATERAL` для применения функции `get_product_names()`, возвращающей множество, для каждой строки таблицы `manufacturers`:

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;
```

Производители, с которыми в данный момент не связаны никакие продукты, не попадут в результат, так как это внутреннее соединение. Если бы мы захотели включить названия и этих производителей, мы могли бы сделать так:

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;
```

Совместимость

Разумеется, оператор `SELECT` совместим со стандартом SQL. Однако не все описанные в стандарте возможности реализованы, а некоторые, наоборот, являются расширениями.

Необязательное предложение `FROM`

Postgres Pro разрешает опустить предложение `FROM`. Это позволяет очень легко вычислять результаты простых выражений:

```
SELECT 2+2;

 ?column?
----------
        4
```

Некоторые другие базы данных SQL не допускают этого, требуя задействовать в `SELECT` фиктивную таблицу с одной строкой.

Пустые списки `SELECT`

Список выходных выражений после `SELECT` может быть пустым, что в результате даст таблицу без столбцов. Стандарт SQL не считает такой синтаксис допустимым, но Postgres Pro допускает его, так как это согласуется с возможностью иметь таблицы с нулём столбцов. Однако когда используется `DISTINCT`, пустой список не допускается.

Необязательное ключевое слово `AS`

В стандарте SQL необязательное ключевое слово `AS` можно опустить перед именем выходного столбца, если это имя является допустимым именем столбца (то есть не совпадает с каким-либо зарезервированным ключевым словом). Postgres Pro несколько более строг: `AS` требуется, если имя столбца совпадает с любым ключевым словом, зарезервированным или нет. Тем не менее рекомендуется использовать `AS` или заключать имена выходных столбцов в кавычки, во избежание конфликтов, возможных при появлении в будущем новых ключевых слов.

В списке `FROM` и стандарт, и Postgres Pro позволяют опускать `AS` перед псевдонимом, который является незарезервированным ключевым словом. Но для имён выходных столбцов это не подходит из-за синтаксической неоднозначности.

Неиспользование псевдонимов вложенных `SELECT` в предложении `FROM`

Согласно стандарту SQL для любого вложенного запроса `SELECT` в списке `FROM` должен быть псевдоним. В Postgres Pro псевдонимы можно не использовать.

`ONLY` и наследование

Стандарт SQL требует заключать в скобки имя таблицы после `ONLY`, например `SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...`. Postgres Pro считает эти скобки необязательными.

Postgres Pro позволяет добавлять в конце `*`, чтобы явно обозначить, что дочерние таблицы включаются в рассмотрение, в отличие от поведения с `ONLY`. Стандарт не позволяет этого.

(Эти соображения в равной степени касаются всех SQL-команд, поддерживающих параметр `ONLY`.)

Ограничения предложения `TABLESAMPLE`

Предложение `TABLESAMPLE` в настоящий момент принимается только для обычных таблиц и материализованных представлений. Однако согласно стандарту SQL оно должно применяться к любым элементам списка `FROM`.

Вызовы функций в предложении `FROM`

Postgres Pro позволяет записать вызов функции непосредственно в виде элемента списка `FROM`. В стандарте SQL такой вызов функции требуется помещать во вложенный `SELECT`; то есть, запись `FROM` *`функция`*`(...)` *`псевдоним`* примерно равнозначна записи `FROM LATERAL (SELECT` *`функция`*`(...))` *`псевдоним`*. Заметьте, что указание `LATERAL` считается неявным; это связано с тем, что стандарт требует поведения `LATERAL` для элемента `UNNEST()` в предложении `FROM`. Postgres Pro обрабатывает `UNNEST()` так же, как и другие функции, возвращающие множества.

Пространства имён в `GROUP BY` и `ORDER BY`

В стандарте SQL-92 предложение `ORDER BY` может содержать ссылки только на выходные столбцы по именам или номерам, тогда как `GROUP BY` может содержать выражения с именами только входных столбцов. Postgres Pro расширяет оба эти предложения, позволяя также применять другие варианты (но если возникает неоднозначность, он разрешает её согласно стандарту). Postgres Pro также позволяет задавать произвольные выражения в обоих предложениях. Заметьте, что имена, фигурирующие в выражениях, всегда будут восприниматься как имена входных, а не выходных столбцов.

В SQL:1999 и более поздних стандартах введено несколько другое определение, которое не полностью совместимо с SQL-92. Однако в большинстве случаев Postgres Pro будет интерпретировать выражение `ORDER BY` или `GROUP BY` так, как требует SQL:1999.

Функциональные зависимости

Postgres Pro распознаёт функциональную зависимость (что позволяет опускать столбцы в `GROUP BY`), только когда первичный ключ таблицы присутствует в списке `GROUP BY`. В стандарте SQL оговариваются дополнительные условия, которые следует учитывать.

`LIMIT` и `OFFSET`

Предложения `LIMIT` и `OFFSET` относятся к специфическим особенностям Postgres Pro и поддерживаются также в MySQL. В стандарте SQL:2008 для той же цели вводятся предложения `OFFSET ... FETCH {FIRST|NEXT} ...`, рассмотренные ранее в Предложение LIMIT. Этот синтаксис также используется в IBM DB2. (Приложения, написанные для Oracle, часто применяют обходной способ и получают эффект этих предложений, задействуя автоматически генерируемый столбец `rownum`, который отсутствует в Postgres Pro.)

`FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE`, `FOR KEY SHARE`

Хотя указание `FOR UPDATE` есть в стандарте SQL, стандарт позволяет использовать его только в предложении `DECLARE CURSOR`. Postgres Pro допускает его использование в любом запросе `SELECT`, а также в подзапросах `SELECT`, но это является расширением. Варианты `FOR NO KEY UPDATE`, `FOR SHARE` и `FOR KEY SHARE`, а также указания `NOWAIT` и `SKIP LOCKED` в стандарте отсутствуют.

Изменение данных в `WITH`

Postgres Pro разрешает использовать `INSERT`, `UPDATE`, `DELETE` и `MERGE` в качестве запросов `WITH`. Стандарт SQL этого не предусматривает.

Нестандартные предложения

`DISTINCT ON ( ... )` — расширение стандарта SQL.

`ROWS FROM( ... )` — расширение стандарта SQL.

Указания `MATERIALIZED` и `NOT MATERIALIZED` предложения `WITH` относятся к расширениям стандарта SQL.

SELECT INTO

SELECT INTO — создать таблицу из результатов запроса

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( выражение [, ...] ) ] ]
    [ { * | выражение [ [ AS ] имя_результата ] } [, ...] ]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] новая_таблица
    [ FROM элемент_FROM [, ...] ]
    [ WHERE условие ]
    [ GROUP BY выражение [, ...] ]
    [ HAVING условие ]
    [ WINDOW имя_окна AS ( определение_окна ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] выборка ]
    [ ORDER BY выражение [ ASC | DESC | USING оператор ] [ NULLS { FIRST | LAST } ]
 [, ...] ]
    [ LIMIT { число | ALL } ]
    [ OFFSET начало [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF имя_таблицы [, ...] ] [ NOWAIT ] [...] ]
```

Описание

`SELECT INTO` создаёт новую таблицу и заполняет её данными, полученными из запроса. Данные не передаются клиенту, как с обычной командой `SELECT`. Столбцы новой таблицы получают имена и типы данных, связанные с выходными столбцами `SELECT`.

Параметры

`TEMPORARY` или `TEMP`

Если указано, создаваемая таблица будет временной. За подробностями обратитесь к CREATE TABLE.

`UNLOGGED`

Если указано, создаваемая таблица будет нежурналируемой. За подробностями обратитесь к CREATE TABLE.

`новая_таблица`

Имя создаваемой таблицы (возможно, дополненное схемой).

Все другие параметры подробно описываются в SELECT.

Примечания

Команда `SELECT INTO` действует подобно `CREATE TABLE AS`, но рекомендуется использовать `CREATE TABLE AS`, так как `SELECT INTO` не поддерживается в ECPG и PL/pgSQL вследствие того, что они воспринимают предложение `INTO` по-своему. К тому же, `CREATE TABLE AS` предоставляет больший набор возможностей, чем `SELECT INTO`.

В отличие от `CREATE TABLE AS`, команда `SELECT INTO` не позволяет задать свойства таблицы, например выбрать метод доступа с помощью указания `USING` *`метод`* или табличное пространство с помощью `TABLESPACE` *`табл_пространство`*. Если это требуется, используйте команду `CREATE TABLE AS`. Таким образом, для новой таблицы выбирается метод доступа к таблицам по умолчанию. За дополнительными сведениями обратитесь к default_table_access_method.

Примеры

Создание таблицы `films_recent`, содержащей только последние записи из таблицы `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

Совместимость

В стандарте SQL команда `SELECT INTO` применяется для передачи скалярных значений клиентской программе, но не для создания новой таблицы. Именно это применение имеет место в ECPG (см. Главу 34) и в PL/pgSQL (см. Главу 41). В Postgres Pro команда `SELECT INTO` связана с созданием таблицы по историческим причинам. Таким же образом работает `SELECT INTO` и в некоторых другие реализациях SQL (но в большинстве реализаций поддерживается конструкция `CREATE TABLE AS`). Если эти соображения совместимости не важны, в новом коде для этих целей лучше использовать `CREATE TABLE AS`.

См. также

CREATE TABLE AS

SET

SET — изменить параметр времени выполнения

Синтаксис

```
SET [ SESSION | LOCAL ] параметр_конфигурации { TO | = } { значение | 'значение' |
 DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { значение | 'значение' | LOCAL | DEFAULT }
```

Описание

Команда `SET` изменяет конфигурационные параметры времени выполнения. С помощью `SET` можно «на лету» изменить многие из параметров, перечисленных в Главе 18. (Но некоторые параметры могут изменять только суперпользователи и пользователя, имеющие право `SET` для них, а другие нельзя изменять после запуска сервера или сеанса.) `SET` влияет на значение параметра только в рамках текущего сеанса.

Если команда `SET` (или равнозначная `SET SESSION`) выполняется внутри транзакции, которая затем прерывается, эффект команды `SET` пропадает, когда транзакция откатывается. Если же окружающая транзакция фиксируется, этот эффект сохраняется до конца сеанса, если его не переопределит другая команда `SET`.

Действие `SET LOCAL` продолжается только до конца текущей транзакции, независимо от того, фиксируется она или нет. Особый случай представляет использование `SET` с последующей `SET LOCAL` в одной транзакции: значение, заданное `SET LOCAL`, будет сохраняться до конца транзакции, но после этого (если транзакция фиксируется) восстановится значение, заданное командой `SET`.

Действия `SET` или `SET LOCAL` также отменяются при откате к точке сохранения, установленной до выполнения этих команд.

Если `SET LOCAL` применяется в функции, параметр `SET` для которой устанавливает значение той же переменной (см. CREATE FUNCTION), действие команды `SET LOCAL` прекращается при выходе из функции; то есть, в любом случае восстанавливается значение, существовавшее при вызове функции. Это позволяет использовать `SET LOCAL` для динамических и неоднократных изменений параметра в рамках функции, и при этом иметь удобную возможность использовать параметр `SET` для сохранения и восстановления значения, полученного извне. Однако обычная команда `SET` переопределяет любой параметр `SET` окружающей функции; её действие сохраняется, если не происходит откат транзакции.

Примечание

В PostgreSQL с версии 8.0 до 8.2 действие `SET LOCAL` могло отменяться при освобождении ранее установленной точки сохранения или при успешном выходе из блока исключения PL/pgSQL. Затем это поведение было признано неинтуитивным, и было изменено.

Параметры

`SESSION`

Указывает, что команда действует в рамках текущего сеанса. (Это поведение по умолчанию, если не указано ни `SESSION`, ни `LOCAL`.)

`LOCAL`

Указывает, что команда действует только в рамках текущей транзакции. После `COMMIT` или `ROLLBACK` в силу вновь вступает значение, определённое на уровне сеанса. При выполнении такой команды вне блока транзакции выдаётся предупреждение и больше ничего не происходит.

`параметр_конфигурации`

Имя устанавливаемого параметра времени выполнения. Доступные параметры описаны в Главе 18 и ниже.

`значение`

Новое значение параметра. Параметры могут задаваться в виде строковых констант, идентификаторов, чисел или списков перечисленных типов через запятую, в зависимости от конкретного параметра. Указание `DEFAULT` в данном контексте позволяет сбросить параметр к значению по умолчанию (то есть, к тому значению, которое он имел бы, если в текущем сеансе не выполнялись бы команды `SET`).

Помимо конфигурационных параметров, описанных в Главе 18, есть ещё несколько параметров, которые можно изменить только командой `SET` или которые имеют особый синтаксис:

`SCHEMA`

`SET SCHEMA 'значение'` — альтернативное написание команды `SET search_path TO значение`. Такой синтаксис позволяет указать только одну схему.

`NAMES`

`SET NAMES значение` — альтернативное написание команды `SET client_encoding TO значение`.

`SEED`

Устанавливает внутреннее начальное значение для генератора случайных чисел (функции `random`). В качестве значения допускаются числа с плавающей точкой от -1 до 1 включительно.

Это начальное значение также можно установить, вызвав функцию `setseed`:

```
SELECT setseed(значение);
```

`TIME ZONE`

`SET TIME ZONE 'значение'` — альтернативное написание команды `SET timezone TO 'значение'`. Синтаксис `SET TIME ZONE` позволяет указывать часовой пояс в специальном формате. Например, допускаются следующие значения:

`'America/Los_Angeles'`

Часовой пояс Беркли, штат Калифорния.

`'Europe/Rome'`

Часовой пояс Италии.

`-7`

Часовой пояс, сдвинутый от UTC на 7 часов к западу (равнозначен PDT). Положительные значения означают сдвиг от UTC к востоку.

`INTERVAL '-08:00' HOUR TO MINUTE`

Часовой пояс, сдвинутый от UTC на 8 часов к западу (равнозначен PST).

`LOCAL`
`DEFAULT`

Устанавливает в качестве часового пояса местный часовой пояс (то есть, значение серверного параметра `timezone` по умолчанию).

Значения часового пояса, заданные в виде чисел или интервалов, внутри переводятся в формат часовых поясов POSIX. Например, после `SET TIME ZONE -7`, команда `SHOW TIME ZONE` покажет `<-07>+07`.

Аббревиатуры часовых поясов команда `SET` не поддерживает; узнать о часовых поясах подробнее можно в Подразделе 8.5.3.

Примечания

Также изменить значение параметра можно с помощью функции `set_config`; см. Подраздел 9.28.1. Кроме того, выполнив UPDATE в системном представлении `pg_settings`, можно произвести то же действие, что выполняет `SET`.

Примеры

Установка пути поиска схем:

```
SET search_path TO my_schema, public;
```

Установка традиционного стиля даты POSTGRES с форматом ввода «день, месяц, год»:

```
SET datestyle TO postgres, dmy;
```

Установка часового пояса для Беркли, штат Калифорния:

```
SET TIME ZONE 'America/Los_Angeles';
```

Установка часового пояса Италии:

```
SET TIME ZONE 'Europe/Rome';
```

Совместимость

`SET TIME ZONE` расширяет синтаксис, определённый в стандарте SQL. Стандарт допускает только числовые смещения часовых поясов, тогда как Postgres Pro позволяет задавать часовой пояс более гибко. Все другие функции `SET` являются расширениями Postgres Pro.

См. также

RESET, SHOW

SET CONSTRAINTS

SET CONSTRAINTS — установить время проверки ограничений для текущей транзакции

Синтаксис

```
SET CONSTRAINTS { ALL | имя [, ...] } { DEFERRED | IMMEDIATE }
```

Описание

`SET CONSTRAINTS` определяет, когда будут проверяться ограничения в текущей транзакции. Ограничения `IMMEDIATE` проверяются в конце каждого оператора, а ограничения `DEFERRED` откладываются до фиксации транзакции. Режим `IMMEDIATE` или `DEFERRED` задаётся для каждого ограничения независимо.

При создании ограничение получает одну из следующих характеристик: `DEFERRABLE INITIALLY DEFERRED` (откладываемое, изначально отложенное), `DEFERRABLE INITIALLY IMMEDIATE` (откладываемое, изначально немедленное) или `NOT DEFERRABLE` (неоткладываемое). Третий вариант всегда подразумевает `IMMEDIATE` и на него команда `SET CONSTRAINTS` не влияет. Первые два варианта запускаются в каждой транзакции в указанном режиме, но их поведение можно изменить в рамках транзакции командой `SET CONSTRAINTS`.

`SET CONSTRAINTS` со списком имён ограничений меняет режим только этих ограничений (все они должны быть откладываемыми). Имя любого ограничения можно дополнить схемой. Если имя схемы не указано, в поисках первого подходящего имени будет просматриваться текущий путь поиска схем. `SET CONSTRAINTS ALL` меняет режим всех откладываемых ограничений.

Когда `SET CONSTRAINTS` меняет режим ограничения с `DEFERRED` на `IMMEDIATE`, новый режим начинает действовать в обратную сторону: все изменения данных, ожидающие проверки в конце транзакции, вместо этого проверяются в момент выполнения команды `SET CONSTRAINTS`. Если какое-либо ограничение нарушается, при выполнении `SET CONSTRAINTS` происходит ошибка (и режим проверки не меняется). Таким образом, с помощью `SET CONSTRAINTS` можно принудительно проверить ограничения в определённом месте транзакции.

В настоящее время это распространяется только на ограничения `UNIQUE`, `PRIMARY KEY`, `REFERENCES` (внешний ключ) и `EXCLUDE`. Ограничения `NOT NULL` и `CHECK` всегда проверяются немедленно в момент добавления или изменения строки (*не* в конце оператора). Ограничения уникальности и ограничения-исключения, объявленные без указания `DEFERRABLE`, так же проверяются немедленно.

Срабатывание триггеров, объявленных как «триггеры ограничений» так же зависит от этой команды — они срабатывают в момент, когда должно проверяться соответствующее ограничение.

Примечания

Так как Postgres Pro не требует, чтобы имена ограничений были уникальны в схеме (достаточно уникальности в таблице), возможно, что для заданного имени найдётся несколько соответствующих ограничений. В этом случае `SET CONSTRAINTS` подействует на все эти ограничения. Для имён без указания схемы, её действие будет распространяться только на ограничение(я), найденное в первой из схем; другие схемы просматриваться не будут.

Эта команда меняет поведение ограничений только в текущей транзакции. При выполнении этой команды вне блока транзакции выдаётся предупреждение и больше ничего не происходит.

Совместимость

Эта команда реализует поведение, описанное в стандарте SQL, с одним исключением — в Postgres Pro она не влияет на проверку ограничений `NOT NULL` и `CHECK`. Кроме того, Postgres Pro проверяет неоткладываемые ограничения уникальности немедленно, а не в конце оператора, как предлагает стандарт.

SET ROLE

SET ROLE — установить идентификатор текущего пользователя в рамках сеанса

Синтаксис

```
SET [ SESSION | LOCAL ] ROLE имя_роли
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

Описание

Эта команда меняет идентификатор текущего пользователя в активном сеансе SQL на *имя_роли*. Имя роли может быть записано в виде идентификатора или строковой константы. После `SET ROLE`, права доступа для команд SQL проверяются так, как если бы сеанс изначально был установлен с этим именем роли. Имейте в виду, что команды `SET ROLE` и `SET SESSION AUTHORIZATION` — исключения: для проверки прав доступа продолжают использоваться текущий и начальный пользователи сеанса (*аутентифицированный пользователь*) соответственно.

Текущий пользователь должен либо непосредственно обладать атрибутом `SET` для указанного *имени_роли*, либо опосредованно по цепочке членства в роли с атрибутом `SET`. (Если пользователь сеанса является суперпользователем, он может выбрать любую роль.)

Указания `SESSION` и `LOCAL` действуют на эту команду так же, как и на обычную команду `SET`.

`SET ROLE NONE` устанавливает в качестве идентификатора текущего пользователя идентификатор текущего пользователя сеанса, выдаваемый функцией `session_user`. `RESET ROLE` устанавливает в качестве идентификатора текущего пользователя значение, заданное во время подключения параметрами командной строки либо командой `ALTER ROLE` или `ALTER DATABASE`. Если же такое значение не задано, в качестве идентификатора текущего пользователя так же устанавливается идентификатор текущего пользователя сеанса. Эти формы могут выполняться любыми пользователями.

Примечания

С помощью этой команды можно как добавить права, так и ограничить их. Если роль пользователя сеанса получила членство в роли с `WITH INHERIT TRUE`, она автоматически получает права такой роли. В этом случае `SET ROLE` убирает все права, кроме принадлежащих этой роли и наследуемых ей. С другой стороны, если роль получила членство с `WITH INHERIT FALSE`, по умолчанию она не получает доступ к правам назначенной роли. Однако если членство в роли было выдано с `WITH SET TRUE`, пользователь сеанса может воспользоваться `SET ROLE`, чтобы убрать права, назначенные непосредственно пользователю сеанса, и вместо них назначить права, которые имеет указанная роль. Если роль получила членство с `WITH INHERIT FALSE, SET FALSE`, доступ к правам назначенной роли нельзя получить и с `SET ROLE`.

`SET ROLE` оказывает действие, сравнимое с `SET SESSION AUTHORIZATION`, но проверка прав выполняется по-другому. Также `SET SESSION AUTHORIZATION` определяет, какие роли разрешены для последующей `SET ROLE`, тогда как при смене ролей командой `SET ROLE` набор ролей, допустимых для последующей команды `SET ROLE`, не меняется.

`SET ROLE` не обрабатывает сеансовые переменные, указанные в свойствах роли (`ALTER ROLE`); они устанавливаются только при подключении.

`SET ROLE` нельзя использовать внутри функции с характеристикой `SECURITY DEFINER`.

Примеры

```
SELECT SESSION_USER, CURRENT_USER;
```

```
 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

Совместимость

Postgres Pro принимает идентификаторы (`"`*`имя_роли`*`"`), тогда как стандарт SQL требует, чтобы имя роли записывалось в виде строковой константы. Стандарт не позволяет выполнять эту команду в транзакции; в Postgres Pro такого ограничения нет, так как для него нет причины. Указания `SESSION` и `LOCAL` относятся к расширениям Postgres Pro, так же, как и синтаксис `RESET`.

См. также

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION — установить идентификатор пользователя сеанса и идентификатор текущего пользователя в рамках сеанса

Синтаксис

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION имя_пользователя
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Описание

Эта команда меняет идентификатор пользователя сеанса и идентификатор текущего пользователя в рамках активного сеанса SQL на *`имя_пользователя`*. Имя пользователя может быть записано в виде идентификатора или строковой константы. С помощью этой команды, можно, например, временно переключиться на непривилегированного пользователя, сохранив возможность затем стать суперпользователем.

Идентификатором пользователя сеанса изначально принимается имя пользователя, введённое клиентом (возможно, прошедшее проверку подлинности). Идентификатор текущего пользователя обычно совпадает с идентификатором пользователя сеанса, но может временно меняться в функциях с характеристикой `SECURITY DEFINER` и подобными механизмами; также его можно изменить командой `SET ROLE`. Идентификатор текущего пользователя принимается во внимание при проверке разрешений.

Идентификатор пользователя сеанса можно изменить, только если начальный пользователь сеанса (*аутентифицированный пользователь*) имеет права суперпользователя. В противном случае эта команда разрешается, только если в ней указывается имя аутентифицированного пользователя.

Указания `SESSION` и `LOCAL` действуют на эту команду так же, как и на обычную команду `SET`.

Формы `DEFAULT` и `RESET` сбрасывают идентификаторы текущего пользователя и пользователя сеанса, так что текущим становится пользователь, изначально проходивший проверку подлинности. Эти формы могут выполняться любым пользователем.

Примечания

`SET SESSION AUTHORIZATION` нельзя использовать в функции с характеристикой `SECURITY DEFINER`.

Примеры

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

Совместимость

Стандарт SQL позволяет вместо строковой константы *`имя_пользователя`* указывать некоторые другие выражения, но на практике это не очень полезно. Postgres Pro допускает синтаксис идентификаторов (`"`*`имя_пользователя`*`"`), а стандарт SQL — нет. Стандарт не позволяет выполнять эту команду в транзакции; в Postgres Pro такого ограничения нет, так как для него нет причины. Указания `SESSION` и `LOCAL` относятся к расширениям Postgres Pro, так же, как и синтаксис `RESET`.

Набор прав, требуемых для выполнения этой команды, согласно стандарту, определяется реализацией.

См. также

SET ROLE

SET TRANSACTION

SET TRANSACTION — установить характеристики текущей транзакции

Синтаксис

```
SET TRANSACTION режим_транзакции [, ...]
SET TRANSACTION SNAPSHOT id_снимка
SET SESSION CHARACTERISTICS AS TRANSACTION режим_транзакции [, ...]

Где режим_транзакции может быть следующим:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Описание

Команда `SET TRANSACTION` устанавливает характеристики текущей транзакции. На последующие транзакции она не влияет. `SET SESSION CHARACTERISTICS` устанавливает характеристики транзакции по умолчанию для последующих транзакций в рамках сеанса. Заданные по умолчанию характеристики затем можно переопределить для отдельных транзакций командой `SET TRANSACTION`.

К характеристикам транзакции относится уровень изоляции транзакции, режим доступа транзакции (чтение/запись или только чтение) и допустимость её откладывания. В дополнение к ним можно выбрать снимок, но только для текущей транзакции, не для сеанса по умолчанию.

Уровень изоляции транзакции определяет, какие данные может видеть транзакция, когда параллельно с ней выполняются другие транзакции:

`READ COMMITTED`

Оператор видит только те строки, которые были зафиксированы до начала его выполнения. Этот уровень устанавливается по умолчанию.

`REPEATABLE READ`

Все операторы текущей транзакции видят только те строки, которые были зафиксированы перед первым запросом на выборку или изменение данных, выполненным в этой транзакции.

`SERIALIZABLE`

Все операторы текущей транзакции видят только те строки, которые были зафиксированы перед первым запросом на выборку или изменение данных, выполненным в этой транзакции. Если наложение операций чтения и записи параллельных сериализуемых транзакций может привести к ситуации, невозможной при последовательном их выполнении (когда одна транзакция выполняется за другой), произойдёт откат одной из транзакций с ошибкой `serialization_failure` (сбой сериализации).

В стандарте SQL определён ещё один уровень, `READ UNCOMMITTED`. В Postgres Pro уровень `READ UNCOMMITTED` обрабатывается как `READ COMMITTED`.

Уровень изоляции транзакции нельзя изменить после выполнения первого запроса на выборку или изменение данных (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `MERGE`, `FETCH` или `COPY`) в текущей транзакции. За дополнительными сведениями об изоляции транзакций и управлении параллельным доступом обратитесь к Главе 13.

Режим доступа транзакции определяет, будет ли транзакция только читать данные или будет и читать, и писать. По умолчанию подразумевается чтение/запись. В транзакции без записи запрещаются следующие команды SQL: `INSERT`, `UPDATE`, `DELETE`, `MERGE` и `COPY FROM`, если только целевая таблица не временная; любые команды `CREATE`, `ALTER` и `DROP`, а также `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE`; кроме того, запрещаются `EXPLAIN ANALYZE` и `EXECUTE`, если команда, которую они должны выполнить, относится к вышеперечисленным. Это высокоуровневое определение режима только для чтения, которое в принципе не исключает запись на диск.

Свойство `DEFERRABLE` оказывает влияние, только если транзакция находится также в режимах `SERIALIZABLE` и `READ ONLY`. Когда для транзакции установлены все три этих свойства, транзакция может быть заблокирована при первой попытке получить свой снимок данных, после чего она сможет выполняться без дополнительных усилий, обычных для режима `SERIALIZABLE`, и без риска привести к сбою сериализации или пострадать от него. Этот режим подходит для длительных операций, например для построения отчётов или резервного копирования.

Команда `SET TRANSACTION SNAPSHOT` позволяет выполнить новую транзакцию со *снимком данных*, который имеет уже существующая. Эта ранее созданная транзакция должна экспортировать этот снимок с помощью функции `pg_export_snapshot` (см. Подраздел 9.28.5). Эта функция возвращает идентификатор снимка, который и нужно передать команде `SET TRANSACTION SNAPSHOT` в качестве идентификатора импортируемого снимка. В данной команде этот идентификатор должен записываться в виде строковой константы, например `'00000003-0000001B-1'`. `SET TRANSACTION SNAPSHOT` можно выполнить только в начале транзакции, до первого запроса на выборку или изменение данных (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `MERGE`, `FETCH` или `COPY`) в текущей транзакции. Более того, для транзакции уже должен быть установлен уровень изоляции `SERIALIZABLE` или `REPEATABLE READ` (в противном случае снимок будет сразу же потерян, так как на уровне `READ COMMITTED` для каждой команды делается новый снимок). Если импортирующая транзакция работает на уровне изоляции `SERIALIZABLE`, то транзакция, экспортирующая снимок, также должна работать на этом уровне. Кроме того, транзакции в режиме чтение/запись не могут импортировать снимок из транзакции в режиме «только чтение».

Примечания

Если команде `SET TRANSACTION` не предшествует `START TRANSACTION` или `BEGIN`, она выдаёт предупреждение и больше ничего не делает.

Без `SET TRANSACTION` можно обойтись, задав требуемые *`режимы_транзакции`* в операторах `BEGIN` или `START TRANSACTION`. Но для `SET TRANSACTION SNAPSHOT` такой возможности не предусмотрено.

Режимы транзакции для сеанса по умолчанию можно также задать или прочитать в конфигурационных переменных default_transaction_isolation, default_transaction_read_only и default_transaction_deferrable. (На практике, `SET SESSION CHARACTERISTICS` — это просто более многословная альтернатива изменению этих переменных командой `SET`.) Это значит, что значения этих переменных по умолчанию можно задать в файле конфигурации, с помощью команды `ALTER DATABASE` и т. д. За дополнительными сведениями обратитесь к Главе 18.

Режимы текущей транзакции можно также задать или прочитать в конфигурационных переменных default_transaction_isolation, transaction_read_only и transaction_deferrable. Присвоение значения одному из этих параметров равнозначна использованию `SET TRANSACTION` с теми же ограничениями по применению. Однако эти параметры нельзя задать в конфигурационном файле или каким-то ещё образом, кроме как непосредственно в SQL.

Примеры

Чтобы начать новую транзакцию со снимком данных, который получила уже существующая транзакция, его нужно сначала экспортировать из первой транзакции. При этом будет получен идентификатор снимка, например:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
```

```
 00000003-0000001B-1
(1 row)
```

Затем этот идентификатор нужно передать команде `SET TRANSACTION SNAPSHOT` в начале новой транзакции:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

Совместимость

Эти команды определены в стандарте SQL, за исключением режима транзакции `DEFERRABLE` и формы `SET TRANSACTION SNAPSHOT`, которые являются расширениями Postgres Pro.

В стандарте уровнем изоляции по умолчанию является `SERIALIZABLE`. В Postgres Pro уровнем по умолчанию обычно считается `READ COMMITTED`, но его можно изменить, как описано выше.

В стандарте SQL есть ещё одна характеристика транзакции, которую нельзя задать этими командами: размер диагностической области. Эта специфическая концепция встраиваемого SQL, поэтому в сервере Postgres Pro она не реализована.

Стандарт SQL требует, чтобы последовательные *`режимы_транзакций`* разделялись запятыми, но по историческим причинам Postgres Pro позволяет опустить запятые.

SHOW

SHOW — показать значение параметра времени выполнения

Синтаксис

```
SHOW имя
SHOW ALL
```

Описание

`SHOW` выводит текущие значения параметров времени выполнения. Эти переменные можно установить, воспользовавшись оператором `SET`, отредактировав файл конфигурации `postgresql.conf`, передав в переменной окружения `PGOPTIONS` (при использовании psql или приложения на базе libpq) либо в параметрах командной строки при запуске сервера `postgres`. За подробностями обратитесь к Главе 18.

Параметры

`имя`

Имя параметра времени выполнения. Доступные параметры описаны в Главе 18 и на странице справки SET. Кроме того, есть несколько параметров, которые можно просмотреть, но нельзя изменить:

`SERVER_VERSION`

Показывает номер версии сервера.

`SERVER_ENCODING`

Показывает кодировку набора символов на стороне сервера. В настоящее время этот параметр можно узнать, но нельзя изменить, так как кодировка определяется в момент создания базы данных.

`IS_SUPERUSER`

Возвращает true, если текущая роль обладает правами суперпользователя.

`ALL`

Показать значения всех конфигурационных параметров с описаниями.

Примечания

Ту же информацию выдаёт функция `current_setting`; см. Подраздел 9.28.1. Кроме того, эту информацию можно получить через системное представление `pg_settings`.

Примеры

Просмотр текущего значения параметра `DateStyle`:

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

Просмотр текущего значения параметра `geqo`:

```
SHOW geqo;
 geqo
```

```
------
 on
(1 row)
```

Просмотр всех параметров:

```
SHOW ALL;
            name         | setting |                description
-------------------------+---------+-------------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in implicit parsing ...
 zero_damaged_pages      | off     | Continues processing past damaged page headers.
(196 rows)
```

Совместимость

Команда `SHOW` является расширением Postgres Pro.

См. также

SET, RESET

START TRANSACTION

START TRANSACTION — начать блок транзакции

Синтаксис

```
START TRANSACTION [ режим_транзакции [, ...] ]

Где режим_транзакции может быть следующим:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
 UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Описание

Эта команда начинает новый блок транзакции. Если указан уровень изоляции, режим чтения/записи или допустимость откладывания транзакции, новая транзакция получит эти характеристики, как при выполнении команды `SET TRANSACTION`. Данная команда равнозначна команде `BEGIN`.

Параметры

За описанием параметров этого оператора обратитесь к SET TRANSACTION.

Совместимость

Согласно стандарту, выполнять `START TRANSACTION`, чтобы начать блок транзакции, необязательно: блок неявно начинает любая команда SQL. Поведение Postgres Pro можно представить как неявное выполнение `COMMIT` после каждой команды, которой не предшествует `START TRANSACTION` (или `BEGIN`), и поэтому такое поведение часто называется «автофиксацией». Другие реляционные СУБД тоже могут предлагать автофиксацию как удобную возможность.

Значение `DEFERRABLE` параметра *`режим_транзакции`* является языковым расширением Postgres Pro.

Стандарт SQL требует, чтобы последовательные *`режимы_транзакций`* разделялись запятыми, но по историческим причинам Postgres Pro позволяет опустить запятые.

См. также сведения о совместимости в описании SET TRANSACTION.

См. также

BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION

TRUNCATE

TRUNCATE — опустошить таблицу или набор таблиц

Синтаксис

```
TRUNCATE [ TABLE ] [ ONLY ] имя [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

Описание

Команда `TRUNCATE` быстро удаляет все строки из набора таблиц. Она действует так же, как безусловная команда `DELETE` для каждой таблицы, но гораздо быстрее, так как она фактически не сканирует таблицы. Более того, она немедленно высвобождает дисковое пространство, так что выполнять операцию `VACUUM` после неё не требуется. Наиболее полезна она для больших таблиц.

Параметры

`имя`

Имя таблицы (возможно, дополненное схемой), подлежащей опустошению. Если перед именем таблицы указано `ONLY`, очищается только заданная таблица. Без `ONLY` очищается и заданная таблица, и все её потомки (если таковые есть). После имени таблицы можно также добавить необязательное указание `*`, чтобы явно обозначить, что блокировка затрагивает и все дочерние таблицы.

`RESTART IDENTITY`

Автоматически перезапускать последовательности, связанные со столбцами опустошаемой таблицы.

`CONTINUE IDENTITY`

Не изменять значения последовательностей. Это поведение по умолчанию.

`CASCADE`

Автоматически опустошать все таблицы, ссылающиеся по внешнему ключу на заданные таблицы, или на таблицы, затронутые в результате действия `CASCADE`.

`RESTRICT`

Отказать в опустошении любых таблиц, на которые по внешнему ключу ссылаются другие таблицы, не перечисленные в этой команде. Это поведение по умолчанию.

Примечания

Чтобы опустошить таблицу, необходимо иметь право `TRUNCATE` для этой таблицы.

Команда `TRUNCATE` запрашивает блокировку `ACCESS EXCLUSIVE` для каждой таблицы, которую она обрабатывает. Когда указано `RESTART IDENTITY`, все последовательности, которые должны быть перезапущены, также блокируются исключительно. В случаях, когда требуется обеспечить параллельный доступ к таблице, следует использовать `DELETE`.

`TRUNCATE` нельзя использовать с таблицей, на которую по внешнему ключу ссылаются другие таблицы, если только и эти таблицы не опустошаются этой же командой. Проверка допустимости очистки в таких случаях потребовала бы сканирования таблицы, а главная идея данной команды в том, чтобы не делать этого. Для автоматической обработки всех зависимых таблиц можно использовать указание `CASCADE` — но будьте очень осторожны с ним, иначе вы можете потерять данные, которые не собирались удалять! В частности, заметьте, что когда опустошаемая таблица является секцией, соседние секции эта операция не затрагивает, но затрагивает все таблицы, на которые ссылается целевая, или которые являются её секциями.

При выполнении `TRUNCATE` не срабатывают никакие триггеры `ON DELETE`, которые могут быть настроены для таблиц. Однако при этом срабатывают триггеры `ON TRUNCATE`. Если триггеры `ON TRUNCATE` определены для любых из этих таблиц, то все триггеры `BEFORE TRUNCATE` срабатывают до того, как происходит опустошение, а все триггеры `AFTER TRUNCATE` срабатывают после того, как завершается опустошение последней таблицы и все последовательности сбрасываются. Триггеры срабатывают по порядку обработки таблиц (сначала для таблиц, перечисленных в команде, затем для тех, что затрагиваются каскадно).

Команда `TRUNCATE` небезопасна с точки зрения MVCC. После опустошения таблицы она будет выглядеть пустой для параллельных транзакций, если они работают со снимком, полученным до опустошения. За подробностями обратитесь к Разделу 13.6.

`TRUNCATE` является надёжной транзакционной операцией в отношении данных в таблицах: опустошение будет безопасно отменено, если окружающая транзакция не будет зафиксирована.

С указанием `RESTART IDENTITY` подразумеваемые операции `ALTER SEQUENCE RESTART` также выполняются транзакционно; то есть, они будут отменены, если окружающая транзакция не будет зафиксирована. Учтите, что если до того, как транзакция отменится, будут выполнены какие-либо дополнительные операции с последовательностями, эффект этих операций также будет отменён, но не их влияние на значение `currval()`; то есть после транзакции `currval()` продолжит возвращать последнее значение последовательности, полученное внутри прерванной транзакции, хотя сама последовательность уже может быть несогласованной с ним. Подобным образом обычно ведёт себя `currval()` после сбоя транзакции.

Команда `TRUNCATE` может работать со сторонними таблицами, если это поддерживает обёртка сторонних данных, как например postgres_fdw.

Примеры

Опустошение таблиц `bigtable` и `fattable`:

```
TRUNCATE bigtable, fattable;
```

Та же операция и сброс всех связанных генераторов последовательностей:

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

Опустошение таблицы `othertable` и каскадная обработка всех таблиц, ссылающихся на `othertable` по ограничениям внешнего ключа:

```
TRUNCATE othertable CASCADE;
```

Совместимость

Стандарт SQL:2008 включает команду `TRUNCATE` с синтаксисом `TRUNCATE TABLE` *`имя_таблицы`*. Предложения `CONTINUE IDENTITY`/`RESTART IDENTITY` также описаны в стандарте, но с небольшими отличиями, хотя их назначение похоже. Поведение этой команды при параллельных операциях, согласно стандарту, отчасти определяются реализацией, так что приведённые выше замечания при необходимости следует учитывать и сопоставлять с другими реализациями.

См. также

DELETE

UNLISTEN

UNLISTEN — прекратить ожидание уведомления

Синтаксис

```
UNLISTEN { канал | * }
```

Описание

`UNLISTEN` применяется для отмены существующей подписки на получение событий `NOTIFY`. `UNLISTEN` отменяет существующую подписку в текущем сеансе Postgres Pro на канал уведомлений с именем *канал*. Специальный знак `*` отменяет все подписки в текущем сеансе.

В описании NOTIFY использование `LISTEN` и `NOTIFY` рассматривается более подробно.

Параметры

канал

Имя канала уведомлений (любой идентификатор).

`*`

Отменяются все текущие подписки на уведомления для активного сеанса.

Примечания

Вы можете также попытаться отменить подписку на канал, на который не подписаны; предупреждений или ошибки при этом не будет.

`UNLISTEN *` автоматически выполняется в конце каждого сеанса.

Транзакция, выполнившая `UNLISTEN`, не может быть подготовлена для двухфазной фиксации.

Примеры

Подписка на получение события:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Сразу после выполнения `UNLISTEN` последующие сообщения `NOTIFY` игнорируются:

```
UNLISTEN virtual;
NOTIFY virtual;
-- событие NOTIFY не поступает
```

Совместимость

Команда `UNLISTEN` отсутствует в стандарте SQL.

См. также

LISTEN, NOTIFY

UPDATE

UPDATE — изменить строки таблицы

Синтаксис

```
[ WITH [ RECURSIVE ] запрос_WITH [, ...] ]
UPDATE [ ONLY ] имя_таблицы [ * ] [ [ AS ] псевдоним ]
    SET { имя_столбца = { выражение | DEFAULT } |
          ( имя_столбца [, ...] ) = [ ROW ] ( { выражение | DEFAULT } [, ...] ) |
          ( имя_столбца [, ...] ) = ( вложенный_SELECT )
        } [, ...]
    [ FROM элемент_FROM [, ...] ]
    [ WHERE условие | WHERE CURRENT OF имя_курсора ]
    [ RETURNING { * | выражение_результата [ [ AS ] имя_результата ] } [, ...] ]
```

Описание

`UPDATE` изменяет значения указанных столбцов во всех строках, удовлетворяющих условию. В предложении `SET` должны указываться только те столбцы, которые будут изменены; столбцы, не изменяемые явно, сохраняют свои предыдущие значения.

Изменить строки в таблице, используя информацию из других таблиц в базе данных, можно двумя способами: применяя вложенные запросы или указав дополнительные таблицы в предложении `FROM`. Выбор предпочитаемого варианта зависит от конкретных обстоятельств.

Предложение `RETURNING` указывает, что команда `UPDATE` должна вычислить и возвратить значения для каждой фактически изменённой строки. Вычислить в нём можно любое выражение со столбцами целевой таблицы и/или столбцами других таблиц, упомянутых во `FROM`. При этом в выражении будут использоваться новые (изменённые) значения столбцов таблицы. Список `RETURNING` имеет тот же синтаксис, что и список результатов `SELECT`.

Для выполнения этой команды необходимо иметь право `UPDATE` для таблицы, или как минимум для столбцов, перечисленных в списке изменяемых. Также необходимо иметь право `SELECT` для всех столбцов, значения которых считываются в *`выражениях`* или *`условии`*.

Параметры

`запрос_WITH`

Предложение `WITH` позволяет задать один или несколько подзапросов, на которые затем можно ссылаться по имени в запросе `UPDATE`. Подробнее об этом см. Раздел 7.8 и SELECT.

`имя_таблицы`

Имя таблицы (возможно, дополненное схемой), строки которой будут изменены. Если перед именем таблицы добавлено `ONLY`, соответствующие строки изменяются только в указанной таблице. Без `ONLY` строки будут также изменены во всех таблицах, унаследованных от указанной. При желании, после имени таблицы можно указать `*`, чтобы явно обозначить, что операция затрагивает все дочерние таблицы.

`псевдоним`

Альтернативное имя целевой таблицы. Когда указывается это имя, оно полностью скрывает фактическое имя таблицы. Например, в запросе `UPDATE foo AS f` дополнительные компоненты оператора `UPDATE` должны обращаться к целевой таблице по имени `f`, а не `foo`.

`имя_столбца`

Имя столбца в таблице *`имя_таблицы`*. Имя столбца при необходимости может быть дополнено именем вложенного поля или индексом массива. Имя таблицы добавлять к имени целевого столбца не нужно — например, запись `UPDATE table_name SET table_name.col = 1` ошибочна.

выражение

Выражение, результат которого присваивается столбцу. В этом выражении можно использовать предыдущие значения этого и других столбцов таблицы.

`DEFAULT`

Присвоить столбцу значение по умолчанию (это может быть NULL, если для столбца не определено некоторое выражение по умолчанию). Столбец идентификации при этом получает значение, выданное соответствующей последовательностью. Для генерируемого столбца это указание допускается, но не меняет обычное поведение, то есть значение столбца вычисляется генерирующим выражением.

вложенный_SELECT

Подзапрос `SELECT`, выдающий столько выходных столбцов, сколько перечислено в предшествующем ему списке столбцов в скобках. При выполнении этого подзапроса должна быть получена максимум одна строка. Если он выдаёт одну строку, значения столбцов в нём присваиваются целевым столбцам; если же он не возвращает строку, целевым столбцам присваивается NULL. Этот подзапрос может обращаться к предыдущим значениям текущей изменяемой строки в таблице.

элемент_FROM

Табличное выражение, позволяющее обращаться в условии `WHERE` и выражениях новых данных к столбцам других таблиц. В нём используется тот же синтаксис, что и в предложении `FROM` оператора `SELECT`; например, вы можете определить псевдоним для таблицы. Имя целевой таблицы повторять в предложении `FROM` нужно, только если вы хотите определить замкнутое соединение (в этом случае для данного имени должен определяться псевдоним).

условие

Выражение, возвращающее значение типа `boolean`. Изменены будут только те стоки, для которых это выражение возвращает `true`.

имя_курсора

Имя курсора, который будет использоваться в условии `WHERE CURRENT OF`. С таким условием будет изменена строка, выбранная из этого курсора последней. Курсор должен образовываться запросом, не применяющим группировку, к целевой таблице команды `UPDATE`. Заметьте, что `WHERE CURRENT OF` нельзя задать вместе с логическим условием. За дополнительными сведениями об использовании курсоров с `WHERE CURRENT OF` обратитесь к DECLARE.

выражение_результата

Выражение, которое будет вычисляться и возвращаться командой `UPDATE` после изменения каждой строки. В этом выражении можно использовать имена любых столбцов таблицы *имя_таблицы* или таблиц, перечисленных в списке `FROM`. Чтобы получить все столбцы, достаточно написать `*`.

имя_результата

Имя, назначаемое возвращаемому столбцу.

Выводимая информация

В случае успешного завершения, `UPDATE` возвращает метку команды в виде

```
UPDATE число
```

Здесь *число* обозначает количество изменённых строк, включая те подлежащие изменению строки, значения в которых не были изменены. Заметьте, что это число может быть меньше количества строк, удовлетворяющих *условию*, когда изменения отменяются триггером `BEFORE UPDATE`. Если *число* равно 0, данный запрос не изменил ни одной строки (это не считается ошибкой).

Если команда `UPDATE` содержит предложение `RETURNING`, её результат будет похож на результат оператора `SELECT` (с теми же столбцами и значениями, что содержатся в списке `RETURNING`), полученный для строк, изменённых этой командой.

Примечания

Когда присутствует предложение `FROM`, целевая таблица по сути соединяется с таблицами, перечисленными в *`элементе_FROM`*, и каждая выходная строка соединения представляет операцию изменения для целевой таблицы. Применяя предложение `FROM`, необходимо обеспечить, чтобы соединение выдавало максимум одну выходную строку для каждой строки, которую нужно изменить. Другими словами, целевая строка не должна соединяться с более чем одной строкой из других таблиц. Если это условие нарушается, только одна из строк соединения будет использоваться для изменения целевой строки, но какая именно, предсказать нельзя.

Из-за этой неопределённости надёжнее ссылаться на другие таблицы только в подзапросах, хотя такие запросы часто хуже читаются и работают медленнее, чем соединение.

В секционированной таблице строка при изменении может перестать удовлетворять ограничению содержащей её секции. При этом если есть другая секция в дереве секционирования, ограничению которой эта строка удовлетворяет, то она переносится в данную секцию. Если такой секции нет, происходит ошибка. За кулисами перемещение строки выполняется посредством операций `DELETE` и `INSERT`.

Существует возможность того, что при выполнении другой параллельной операции `UPDATE` или `DELETE` с перемещаемой строкой возникнет ошибка сериализации. Например, предположим, что в сеансе 1 выполняется `UPDATE` для ключа секционирования, а тем временем в параллельном сеансе 2, в котором эта строка видима, выполняется операция `UPDATE` или `DELETE` с этой строкой. В этом случае `UPDATE`/`DELETE` в сеансе 2 заметит перемещение строки и выдаст ошибку сериализации (которая всегда представляется кодом SQLSTATE '40001'). Получив такую ошибку, приложения могут попытаться повторить транзакцию. В обычном случае, когда таблица не секционирована или строка не перемещается, в сеансе 2 видна изменённая строка, и операция `UPDATE`/`DELETE` выполняется с новой версией строки.

Заметьте, что строки могут перемещаться из локальных секций в секцию в сторонней таблице (если обёртка сторонних данных поддерживает перенаправление кортежей), но не из секции в сторонней таблице в другую секцию.

Попытка переместить строку из одной секции в другую завершится неудачей, если обнаружится, что внешний ключ напрямую ссылается на предка исходной секции, а в `UPDATE` фигурирует не этот предок.

Примеры

Изменение слова `Drama` на `Dramatic` в столбце `kind` таблицы `films`:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Изменение значений температуры и сброс уровня осадков к значению по умолчанию в одной строке таблицы `weather`:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Выполнение той же операции с получением изменённых записей:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp;
```

Такое же изменение с применением альтернативного синтаксиса со списком столбцов:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
```

```
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Увеличение счётчика продаж для менеджера, занимающегося компанией Acme Corporation, с применением предложения `FROM`:

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

Выполнение той же операции, с вложенным запросом в предложении `WHERE`:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

Изменение имени контакта в таблице счетов (это должно быть имя назначенного менеджера по продажам):

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
    (SELECT first_name, last_name FROM employees
     WHERE employees.id = accounts.sales_person);
```

Подобный результат можно получить, применив соединение:

```
UPDATE accounts SET contact_first_name = first_name,
                    contact_last_name = last_name
  FROM employees WHERE employees.id = accounts.sales_person;
```

Однако если `employees.id` — не уникальный ключ, второй запрос может давать непредсказуемые результаты, тогда как первый запрос гарантированно выдаст ошибку, если найдётся несколько записей с одним `id`. Кроме того, если соответствующая запись `accounts.sales_person` не найдётся, первый запрос запишет в поля имени NULL, а второй вовсе не изменит строку.

Обновление статистики в сводной таблице в соответствии с текущими данными:

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
     WHERE d.group_id = s.group_id);
```

Попытка добавить новый продукт вместе с количеством. Если такая запись уже существует, вместо этого увеличить количество данного продукта в существующей записи. Чтобы реализовать этот подход, не откатывая всю транзакцию, можно использовать точки сохранения:

```
BEGIN;
-- другие операции
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- Предполагая, что здесь возникает ошибка из-за нарушения уникальности ключа,
-- мы выполняем следующие команды:
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- Продолжение других операций и в завершение...
COMMIT;
```

Изменение столбца `kind` таблицы `films` в строке, на которой в данный момент находится курсор `c_films`:

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

Одновременное изменение большого количества строк может негативно влиять на производительность, вызывая раздувание таблиц, отставание реплики, повышенный уровень конфликтов блокировок. В таких случаях имеет смысл выполнять операции небольшими порциями, заодно применяя к таблице команду `VACUUM` между ними. Для команды `UPDATE` нет предложения `LIMIT`, но можно добиться похожего эффекта с помощью Общих табличных выражений и замкнутого соединения.

Замкнутое соединение на системном столбце ctid эффективно работает со стандартным табличным методом доступа Postgres Pro:

```
WITH exceeded_max_retries AS (
  SELECT w.ctid FROM work_item AS w
    WHERE w.status = 'active' AND w.num_retries > 10
    ORDER BY w.retry_timestamp
    FOR UPDATE
    LIMIT 5000
)
UPDATE work_item SET status = 'failed'
  FROM exceeded_max_retries AS emr
  WHERE work_item.ctid = emr.ctid;
```

Эту команду нужно будет повторять до тех пор, пока не останется ни одной строки, требующей изменения. Предложение `ORDER BY` позволяет указывать, какие строки обновить первыми. Оно также помогает избегать взаимоблокировок с другими операциями изменений, если они запрашивают блокировки в одинаковом порядке. Если существует опасность конфликта блокировок, можно добавить `SKIP LOCKED` к CTE, чтобы разные команды не могли изменять одну и ту же строку. Однако в таком случае потребуется финальная команда `UPDATE` без `SKIP LOCKED` или `LIMIT`, чтобы убедиться, что никакие строки не были пропущены.

Совместимость

Эта команда соответствует стандарту SQL, за исключением предложений `FROM` и `RETURNING`, которые являются расширениями Postgres Pro, как и возможность применять `WITH` с `UPDATE`.

В некоторых других СУБД также поддерживается дополнительное предложение `FROM`, но предполагается, что целевая таблица должна ещё раз упоминаться в этом предложении. Postgres Pro воспринимает предложение `FROM` не так, поэтому будьте внимательны, портируя приложения, которые используют это расширение языка.

Согласно стандарту, исходным значением для вложенного списка имён столбцов в скобках может быть любое выражение, выдающее строку с нужным количеством столбцов. Postgres Pro принимает в качестве этого значения только конструктор строки или вложенный `SELECT`. Изменяемое значение отдельного столбца можно обозначать словом `DEFAULT` в конструкторе строки, но не внутри вложенного `SELECT`.

VACUUM

VACUUM — провести сборку мусора и, возможно, проанализировать базу данных

Синтаксис

```
VACUUM [ ( параметр [, ...] ) ] [ таблица_и_столбцы [, ...] ]

Здесь допускается параметр:

    FULL [ boolean ]
    FREEZE [ boolean ]
    VERBOSE [ boolean ]
    ANALYZE [ boolean ]
    DISABLE_PAGE_SKIPPING [ boolean ]
    SKIP_LOCKED [ boolean ]
    INDEX_CLEANUP { AUTO | ON | OFF }
    PROCESS_MAIN [ boolean ]
    PROCESS_TOAST [ boolean ]
    TRUNCATE [ boolean ]
    PARALLEL integer
    SKIP_DATABASE_STATS [ boolean ]
    ONLY_DATABASE_STATS [ boolean ]
    BUFFER_USAGE_LIMIT размер

и таблица_и_столбцы:

    имя_таблицы [ ( имя_столбца [, ...] ) ]
```

Описание

`VACUUM` высвобождает пространство, занимаемое «мёртвыми» кортежами. При обычных операциях Postgres Pro кортежи, удалённые или устаревшие в результате обновления, физически не удаляются из таблицы; они сохраняются в ней, пока не будет выполнена команда `VACUUM`. Таким образом, периодически необходимо выполнять `VACUUM`, особенно для часто изменяемых таблиц.

Без списка *таблица_и_столбцы* команда `VACUUM` обрабатывает все таблицы и материализованные представления в текущей базе данных, на очистку которых текущий пользователь имеет право. С этим списком `VACUUM` обрабатывает только указанную таблицу(ы).

`VACUUM ANALYZE` выполняет очистку (`VACUUM`), а затем анализ (`ANALYZE`) всех указанных таблиц. Это удобная комбинация для регулярного обслуживания БД. За дополнительной информацией об анализе обратитесь к описанию ANALYZE.

Простая команда `VACUUM` (без `FULL`) только высвобождает пространство и делает его доступным для повторного использования. Эта форма команды может работать параллельно с обычными операциями чтения и записи таблицы, так она не требует исключительной блокировки. Однако освобождённое место не возвращается операционной системе (в большинстве случаев); оно просто остаётся доступным для размещения данных этой же таблицы. Она также позволяет задействовать для обработки несколько процессоров. Этот режим называется *параллельной очисткой*. Для отключения этого режима можно воспользоваться указанием `PARALLEL` и задать нулевое количество параллельных рабочих процессов. `VACUUM FULL` переписывает всё содержимое таблицы в новый файл на диске, не содержащий ничего лишнего, что позволяет возвратить неиспользованное пространство операционной системе. Эта форма работает намного медленнее и запрашивает блокировку в режиме `ACCESS EXCLUSIVE` для каждой обрабатываемой таблицы.

Параметры

`FULL`

Выбирает режим «полной» очистки, который может освободить больше пространства, но выполняется гораздо дольше и запрашивает исключительную блокировку таблицы. Этот режим также требует дополнительное место на диске, так как он записывает новую копию таблицы и не освобождает старую до завершения операции. Обычно это следует использовать, только когда требуется высвободить значительный объём пространства, выделенного таблице.

`FREEZE`

Выбирает агрессивную «заморозку» кортежей. Добавление указания `FREEZE` равносильно выполнению команды `VACUUM` с параметрами vacuum_freeze_min_age и vacuum_freeze_table_age, равными нулю. Агрессивная заморозка всегда выполняется при перезаписи таблицы, поэтому в режиме `FULL` это указание избыточно.

`VERBOSE`

Выводит подробный отчёт об очистке для каждой таблицы.

`ANALYZE`

Обновляет статистику, которую использует планировщик для выбора наиболее эффективного способа выполнения запроса.

`DISABLE_PAGE_SKIPPING`

Обычно `VACUUM` пропускает страницы, учитывая карту видимости. Страницы, на которых, судя по карте, все кортежи заморожены, можно пропускать всегда, а страницы, в которых все кортежи видны всем транзакциям, могут обрабатываться только при агрессивной очистке. Более того, за исключением агрессивной очистки, некоторые страницы можно пропускать, чтобы не ждать, пока другие сеансы закончат их использовать. Этот параметр отключает пропуск страниц и предназначен для использования только когда целостность карты видимости вызывает подозрения, что возможно при аппаратных или программных сбоях, приводящих к разрушению БД.

`SKIP_LOCKED`

Указывает, что команда `VACUUM` не должна ждать освобождения никаких конфликтующих блокировок, начиная обработку отношения: если отношение не удаётся заблокировать сразу, без ожидания, оно пропускается. Заметьте, что даже с этим указанием `VACUUM` может заблокироваться, открывая индексы отношения. Кроме того, `VACUUM ANALYZE` может заблокироваться и при получении выборки строк из секций, потомков в иерархии наследования или некоторых видов сторонних таблиц. Учтите также, что при наличии конфликтующей блокировки в секционированной таблице команда `VACUUM` с этим указанием пропускает все её секции, тогда как обычно все они обрабатываются.

`INDEX_CLEANUP`

Обычно `VACUUM` пропускает очистку индекса, когда в таблице имеется всего несколько мёртвых кортежей. В подобных ситуациях стоимость обработки всех индексов таблицы может значительно превышать выгоду от удаления из индекса мёртвых кортежей. Данное указание можно использовать, чтобы в ходе `VACUUM` индексы обрабатывались принудительно при любом ненулевом количестве мёртвых кортежей. Со значением `AUTO` (действующим по умолчанию) `VACUUM` может пропускать очистку индексов, когда она видится нецелесообразной. Если для `INDEX_CLEANUP` задано значение `ON`, `VACUUM` будет безусловно удалять все мёртвые кортежи из индексов. Это может быть полезно для обеспечения обратной совместимости с предыдущими версиями PostgreSQL, где такое поведение было стандартным.

Для указания `INDEX_CLEANUP` можно также задать значение `OFF`, чтобы в ходе `VACUUM` *всегда* пропускалась очистка индексов, даже при наличии в таблице множества мёртвых кортежей.

Это может быть полезно, когда нужно выполнить очистку как можно быстрее для предотвращения надвигающейся угрозы зацикливания идентификаторов транзакций (см. Подраздел 23.1.5). Однако предпочтительнее положиться на механизм защиты от такого зацикливания, управляемый параметром vacuum_failsafe_age, который обычно срабатывает автоматически. Если очистка индекса не производится регулярно, производительность может ухудшаться, так как по мере внесения изменений в таблицу индексы будут накапливать мёртвые кортежи, а сама таблица будет накапливать мёртвые указатели, которые могут быть удалены только после очистки индекса.

Это указание не действует на таблицы, в которых нет индексов, и игнорируется в случае использования указания `FULL`. Также оно не влияет на работу механизма защиты от зацикливания идентификаторов транзакций. Когда он включается, очистка индексов пропускается, даже если для `INDEX_CLEANUP` задано значение `ON`.

`PROCESS_MAIN`

Указывает, что команда `VACUUM` должна попытаться обработать основное отношение. Обычно это желательная операция и она выполняется по умолчанию. Отключать эту операцию может быть полезно, если требуется провести очистку только `TOAST`-таблицы отношения.

`PROCESS_TOAST`

Указывает, что команда `VACUUM` должна попытаться обработать соответствующую `TOAST`-таблицу для каждого отношения, если таковая существует. Обычно это желательная операция и она выполняется по умолчанию. Отключать эту операцию может быть полезно, если требуется провести очистку только основного отношения. Это указание игнорируется в случае использования указания `FULL`.

`TRUNCATE`

Указывает, что команда `VACUUM` должна попытаться обрезать пустые страницы в конце таблицы, чтобы освободившееся место было возвращено операционной системе. Обычно это желательная операция и она выполняется по умолчанию, если только для таблицы, подлежащей очистке, не сброшен параметр `vacuum_truncate`. Отключать эту операцию может иметь смысл, чтобы избежать блокировки `ACCESS EXCLUSIVE` для таблицы, подлежащей очистке. Это указание игнорируется в случае использования указания `FULL`.

`PARALLEL`

Управляет этапами очистки и уборки индексов в ходе параллельного выполнения `VACUUM`, определяя *`целое_число`* фоновых рабочих процессов (более подробно каждый этап очистки описан в Таблице 26.46). Число рабочих процессов, используемых для этой операции, равняется числу индексов в отношении, подходящих для параллельной очистки, и может ограничиваться сверху количеством, заданным указанием `PARALLEL`, а также дополнительно ограничивается параметром max_parallel_maintenance_workers. Индекс может обрабатываться в режиме параллельной очистки тогда и только тогда, когда его размер превышает min_parallel_index_scan_size. Заметьте, что при этом не гарантируется, что во время очистки будет задействоваться столько параллельных исполнителей, сколько задаёт параметр *`целое_число`*. В ходе очистки рабочие процессы могут использоваться в меньшем количестве или не использоваться вовсе. Для обработки одного индекса может быть использован только один рабочий процесс. Поэтому параллельные исполнители будут запускаться, только если в таблице есть минимум `2` индекса. Рабочие процессы очистки запускаются перед началом каждого этапа и завершаются после его окончания. В будущих выпусках это поведение может измениться. В режиме `FULL` это указание не поддерживается.

`SKIP_DATABASE_STATS`

Указывает, что команда `VACUUM` должна пропускать обновление статистики о самых старых незамороженных значениях XID на уровне базы данных. Как правило, команда `VACUUM` обновляет эту статистику по окончании выполнения. Однако в базе данных с очень большим количеством таблиц это может занять некоторое время и не привести ни к какому результату (если среди очищенных таблиц не было таблицы с самыми старыми незамороженными значениями XID). Кроме того, если запустить несколько команд `VACUUM` параллельно, они будут обновлять статистику по очереди. Таким образом, если приложение выполняет серию команд `VACUUM`, может быть полезно задать этот параметр для всех команд, кроме последней, или задать его для всех команд и после их выполнения отдельно запустить `VACUUM (ONLY_DATABASE_STATS)`.

`ONLY_DATABASE_STATS`

Указывает, что команда `VACUUM` должна только обновить статистику о самых старых незамороженных значениях XID на уровне базы данных. Если этот параметр задан, список *`таблица_и_столбцы`* должен быть пустым, а из других параметров должен быть задан только `VERBOSE`.

`BUFFER_USAGE_LIMIT`

Указывает размер кольцевого буфера *стратегии доступа к буферам* (Buffer Access Strategy) для `VACUUM`. Этот размер используется для расчёта количества общих буферов, переиспользуемых в рамках этой стратегии. С значением `0` использование стратегии доступа к буферам отключается. Если параметр `ANALYZE` также задан, значение `BUFFER_USAGE_LIMIT` используется как на этапе анализа, так и на этапе очистки. Этот параметр не может использоваться в режиме `FULL`, если не задан параметр `ANALYZE`. Если не задать этот параметр, `VACUUM` использует значение из vacuum_buffer_usage_limit. С большим значением параметра `VACUUM` может выполняться быстрее, но при слишком большом значении многие полезные страницы могут вытесняться из общих буферов. Минимальное значение — `128 kB`, максимальное — `16 GB`.

`boolean`

Включает или отключает заданный параметр. Для включения параметра можно написать `TRUE`, `ON` или `1`, а для отключения — `FALSE`, `OFF` или `0`. Значение *`boolean`* можно опустить, в этом случае подразумевается `TRUE`.

`integer`

Задаёт неотрицательное целое значение, передаваемое выбранному параметру.

`размер`

Задаёт объём памяти в килобайтах. Можно также указать в виде строки, содержащей числовой размер, за которым следует одна из следующих единиц информации: `B` (байты), `kB` (килобайты), `MB` (мегабайты), `GB` (гигабайты) или `TB` (терабайты).

`имя_таблицы`

Имя (возможно, дополненное схемой) определённой таблицы или материализованного представления, подлежащего очистке. Если указанная таблица является секционированной, очистке подвергаются все её конечные секции.

`имя_столбца`

Имя столбца, подлежащего анализу. По умолчанию анализируются все столбцы. Если указывается список столбцов, также должно присутствовать указание `ANALYZE`.

Выводимая информация

С указанием `VERBOSE` команда `VACUUM` выдаёт сообщения о процессе очистки, отмечая текущую обрабатываемую таблицу. Также она выводит различные статистические сведения о таблицах.

Примечания

Чтобы очистить таблицу, обычно нужно иметь право `MAINTAIN` для таблицы. Однако владельцам баз данных также разрешено очищать все таблицы в своих базах, за исключением общих каталогов. `VACUUM` при обработке пропускает все таблицы, на очистку которых текущий пользователь не имеет прав.

На время выполнения команды `VACUUM` search_path временно меняется на `pg_catalog, pg_temp`.

`VACUUM` нельзя выполнять внутри блока транзакции.

Для таблиц с индексами GIN, `VACUUM` (в любой форме) также завершает все ожидающие операции добавления в индекс, перемещая записи индекса из очереди в соответствующие места в основной структуре индекса GIN. За подробностями обратитесь к Подразделу 62.4.4.1.

Мы рекомендуем проводить очистку всех баз данных на постоянной основе, чтобы происходило удаление мёртвых строк. Postgres Pro включает средство «автоочистки», которое позволяет автоматизировать регулярную очистку. Чтобы узнать больше об автоматической и ручной очистке, обратитесь к Разделу 23.1.

Режим `FULL` не рекомендуется для обычного применения, но в некоторых случаях он бывает полезен. Например, когда были удалены или изменены почти все строки таблицы, может возникнуть желание физически сжать её, чтобы освободить место на диске и ускорить сканирование этой таблицы. Чаще всего `VACUUM FULL` сжимает таблицу более эффективно, чем обычный `VACUUM`.

Режим `PARALLEL` используется только для очистки. Если это указание задаётся вместе с `ANALYZE`, на работу `ANALYZE` оно не влияет.

`VACUUM` создаёт значительную нагрузку на подсистему ввода-вывода, что может отрицательно сказаться на производительности других активных сеансов. Поэтому иногда полезно использовать возможность задержки очистки с учётом её стоимости. При параллельной очистке каждый рабочий процесс приостанавливается на время, пропорциональное объёму произведённой им работы. За подробностями обратитесь к Подразделу 18.4.4.

Каждый процесс, выполняющий операцию `VACUUM` без указания `FULL` будет выдавать информацию о ходе её выполнения в представлении `pg_stat_progress_vacuum`. Процессы же, выполняющие `VACUUM FULL`, будут выдавать аналогичную информацию в представлении `pg_stat_progress_cluster`. За подробностями обратитесь к Подразделу 26.4.5 и Подразделу 26.4.2.

Примеры

Очистка одной таблицы `onek`, проведение её анализа для оптимизатора и печать подробного отчёта о действиях операции очистки:

```
VACUUM (VERBOSE, ANALYZE) onek;
```

Совместимость

Оператор `VACUUM` отсутствует в стандарте SQL.

До версии PostgreSQL 9.0 использовался и по-прежнему поддерживается следующий синтаксис:

```
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ] [ таблица_и_столбцы [, ...] ]
```

Заметьте, что в этом синтаксисе параметры должны идти строго в указанном порядке.

См. также

vacuumdb, Подраздел 18.4.4, Подраздел 23.1.6, Подраздел 26.4.5, Подраздел 26.4.2

VALUES

VALUES — вычислить набор строк

Синтаксис

```
VALUES ( выражение [, ...] ) [, ...]
    [ ORDER BY выражение_сортировки [ ASC | DESC | USING оператор ] [, ...] ]
    [ LIMIT { число | ALL } ]
    [ OFFSET начало [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ число ] { ROW | ROWS } ONLY ]
```

Описание

`VALUES` вычисляет значение строки или множество значений строк, заданное выражениями. Чаще всего эта команда используется для формирования «таблицы констант» в большой команде, но её можно использовать и отдельно.

Когда указывается больше, чем одна строка, все строки должны иметь одинаковое количество элементов. Типы данных результирующих столбцов таблицы определяются в результате совмещения явных и неявных типов выражений, заданных для этих столбцов, по тем же правилам, что и в `UNION` (см. Раздел 10.5).

В составе других команд синтаксис допускает использование `VALUES` везде, где допускается `SELECT`. Так как грамматически она воспринимается как `SELECT`, с командой `VALUES` можно использовать предложения `ORDER BY`, `LIMIT` (или равнозначное `FETCH FIRST`) и `OFFSET`.

Параметры

выражение

Константа или выражение, которое вычисляется и вставляется в указанное место результирующей таблицы (множества строк). В списке `VALUES`, находящемся на верхнем уровне оператора `INSERT`, *выражение* может быть заменено словом `DEFAULT`, указывающим, что в целевой столбец должно быть вставлено значение этого столбца по умолчанию. Когда `VALUES` употребляется в других контекстах, указание `DEFAULT` использовать нельзя.

выражение_сортировки

Выражение или целочисленная константа, указывающая, как должны сортироваться строки результата. Это выражение может обращаться к столбцам результата `VALUES` по именам `column1`, `column2` и т. д. За дополнительными подробностями обратитесь к разделу Предложение ORDER BY в описании SELECT.

оператор

Оператор сортировки. За подробностями обратитесь к разделу Предложение ORDER BY в описании SELECT.

число

Максимальное число строк, которое должно быть возвращено. За подробностями обратитесь к разделу Предложение LIMIT в описании SELECT.

начало

Число строк, которые должны быть пропущены, прежде чем начнётся выдача строк. За подробностями обратитесь к разделу Предложение LIMIT в описании SELECT.

Примечания

Следует избегать составления списков `VALUES` с очень большим количеством строк, так как при этом можно столкнуться с нехваткой памяти или снижением производительности. Применение

`VALUES` в команде `INSERT` — особый случай (так как ожидаемые типы столбцов становятся известны из целевой таблицы команды `INSERT` и их не надо вычислять, сканируя весь список `VALUES`), так что в этом контексте можно работать с гораздо более объёмными списками, чем в других.

Примеры

Простейшая команда `VALUES`:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

Эта команда выдаст таблицу из двух столбцов и трёх строк. По сути она равнозначна запросу:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

Более типично использование `VALUES` в составе большей команды SQL. Чаще всего она применяется в `INSERT`:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

В контексте `INSERT` список `VALUES` может содержать слово `DEFAULT`, указывающее, что в данном месте вместо некоторого значения должно использоваться значение столбца по умолчанию:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

`VALUES` также может применяться там, где можно написать вложенный `SELECT`, например в предложении `FROM`:

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

Заметьте, что когда `VALUES` используется в предложении `FROM`, предложение `AS` становится обязательным, так же, как и для `SELECT`. При этом не требуется указывать в `AS` имена всех столбцов, но это рекомендуется делать. (По умолчанию Postgres Pro даёт столбцам `VALUES` имена `column1`, `column2` и т. д., но в других СУБД имена могут быть другими.)

Когда `VALUES` используется в команде `INSERT`, значения автоматически приводятся к типу данных соответствующего целевого столбца. Когда оно используется в других контекстах, может потребоваться указать нужный тип данных. Если все записи представлены строковыми константами в кавычках, достаточно привести к нужному типу значения в первой строке, чтобы задать тип для всех:

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

Подсказка

Для простых проверок на включение `IN` лучше полагаться на форму `IN` со списком скаляров, чем записывать запрос `VALUES`, как показано выше. Список скаляров проще записать и обрабатывается он зачастую более эффективно.

Совместимость

`VALUES` соответствует стандарту SQL. Указания `LIMIT` и `OFFSET` являются расширениями Postgres Pro; см. также SELECT.

См. также

INSERT, SELECT

Клиентские приложения Postgres Pro

Раздел описывает клиентские приложения и утилиты Postgres Pro. Некоторые из описанных приложений требуют особые привилегии. Основной отличительной особенностью этих приложений является возможность исполнения на любом компьютере, независимо от расположения сервера баз данных.

Имя пользователя и базы данных передаются из командной строки на сервер без изменения регистра — все пробельные и специальные символы необходимо экранировать с помощью кавычек. Имена таблиц и другие идентификаторы передаются регистр-независимо, за исключением отдельно описанных ситуаций, где может требоваться экранирование.

clusterdb

clusterdb — кластеризовать базу данных Postgres Pro

Синтаксис

`clusterdb` [*`параметр-подключения`*...] [*`параметр`*...] [`--table` | `-t` *`таблица`*] ... [*`dbname`* | `-a` | `--all`]

Описание

clusterdb это приложение для повторной кластеризации таблиц базы данных Postgres Pro. Утилита находит ранее кластеризованные таблицы и проводит операцию на основании последнего использованного индекса. Затрагиваются лишь ранее кластеризованные таблицы.

clusterdb — это обёртка для SQL-команды CLUSTER. Кластеризация баз данных с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

clusterdb принимает следующие аргументы командной строки:

`-a`
`--all`

Кластеризовать все базы данных.

`[-d]` *`dbname`*
`[--dbname=]`*`dbname`*

Указывает имя базы данных для кластеризации, когда не используется параметр `-a`/`--all`. Если это указание отсутствует, имя базы определяется переменной окружения `PGDATABASE`. Если эта переменная не установлена, именем базы будет имя пользователя, указанное для подключения. В аргументе *`имя_бд`* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении clusterdb.

`-q`
`--quiet`

Подавлять вывод сообщений о прогрессе выполнения.

`-t` *`таблица`*
`--table=`*`таблица`*

Кластеризовать *`таблицу`*. Возможно множественное использование параметра `-t`.

`-v`
`--verbose`

Вывести подробную информацию во время процесса.

`-V`
`--version`

Вывести версию clusterdb и прервать дальнейшее выполнение.

`-?`
`--help`

Вывести справку по аргументам команды clusterdb.

clusterdb также принимает из командной строки параметры подключения:

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как clusterdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, clusterdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*`dbname`*

Когда используется ключ `-a`/`--all`, выполняется подключение к указанной базе данных для определения списка подлежащих кластеризации баз данных. Если ключ не указан, будет выбрана база `postgres`, а если она не существует — `template1`. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Диагностика

В случае возникновения трудностей, обратитесь к CLUSTER и psql. Переменные окружения и параметры подключения по умолчанию libpq будут применены при запуске утилиты, это следует учитывать при диагностике.

Примеры

Для кластеризации базы данных `test`:

```
$ clusterdb test
```

Для кластеризации отдельной таблицы `foo` базы данных `xyzzy`:

```
$ clusterdb --table=foo xyzzy
```

См. также

CLUSTER

createdb

createdb — создать базу данных Postgres Pro

Синтаксис

`createdb` [*параметр-подключения*...] [*параметр*...] [*dbname* [*описание*]]

Описание

createdb создаёт базу данных Postgres Pro.

Чаще всего пользователь, выполняющий эту команду, назначается владельцем создаваемой базы данных. Однако можно указать владельца явным образом с помощью флага `-O`, если у текущего пользователя достаточно привилегий.

createdb это обёртка для SQL-команды `CREATE DATABASE`. Создание баз данных с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

createdb принимает в качестве аргументов:

dbname

Указывает имя создаваемой базы. Имя должно быть уникальным в рамках кластера Postgres Pro. По умолчанию в качестве имени базы данных берётся имя текущего системного пользователя.

описание

Добавляет комментарий к создаваемой базе.

`-D` *табличное_пространство*
`--tablespace=`*табличное_пространство*

Указывает табличное пространство, используемое по умолчанию. Имя пространства обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении createdb.

`-E` *кодировка*
`--encoding=`*кодировка*

Указывает кодировку базы данных. Поддерживаемые сервером Postgres Pro кодировки описаны в Подразделе 22.3.1.

`-l` *локаль*
`--locale=`*локаль*

Указывает локаль базы данных. Установка того же значения для `--lc-collate`, `--lc-ctype` и `--icu-locale` равнозначна установке этого параметра. Некоторые локали можно указать только для ICU и только в параметре `--icu-locale`.

`--lc-collate=`*локаль*

Устанавливает параметр LC_COLLATE для базы данных.

`--lc-ctype=`*локаль*

Устанавливает параметр LC_CTYPE для базы данных.

`--builtin-locale=`*`локаль`*

Указывает имя локали при использовании провайдера `builtin`. Поддержка локалей описана в Разделе 22.1.

`--icu-locale=`*`локаль`*

Указывает идентификатор локали ICU, которая будет использоваться в этой БД, если выбран провайдер локалей ICU. По умолчанию используется значение `--locale`, если оно указано, в противном случае — значение `--locale` из базы-шаблона, если оно указано. Если же не указано и оно — значение lc_collate.

`--icu-rules=`*`правила`*

Указывает дополнительные правила для настройки поведения сортировки по умолчанию. Параметр поддерживается только для ICU.

`--locale-provider={builtin|libc|icu}`

Указывает провайдера локали для основного правила сортировки базы данных.

`-O` *`владелец`*
`--owner=`*`владелец`*

Указывает пользователя в качестве владельца создаваемой базы. Имя пользователя обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

`-S` *`стратегия`*
`--strategy=`*`стратегия`*

Указывает стратегию создания базы данных. За подробностями обратитесь к CREATE DATABASE STRATEGY.

`-T` *`шаблон`*
`--template=`*`шаблон`*

Указывает шаблон, на основе которого будет создана база данных. Имя шаблона обрабатывается аналогично идентификаторам, заключённым в двойные кавычки.

`-V`
`--version`

Вывести версию createdb и прервать дальнейшее исполнение.

`-?`
`--help`

Вывести помощь по команде createdb и прервать выполнение.

Флаги `-D`, `-l`, `-E`, `-O` и `-T` по назначению соответствуют флагам SQL-команды `CREATE DATABASE`.

createdb также принимает из командной строки параметры подключения:

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как createdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, createdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*`dbname`*

Указывает имя опорной базы данных, к которой будет произведено подключение для создания новой. Если имя не указано, будет выбрана база `postgres`, а если она не существует — `template1`. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

Переменные окружения

`PGDATABASE`

Если установлено и не переопределено в командной строке, задаёт имя создаваемой базы данных.

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию. `PGUSER` указывает имя пользователя при создании базы данных, если не указано явно в командной строке или в переменной окружения `PGDATABASE`.

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Диагностика

В случае возникновения трудностей обратитесь к CREATE DATABASE и psql. При диагностике нужно учитывать, что при запуске утилиты используются значения переменных окружения и параметров подключения по умолчанию libpq.

Примеры

Создать базу данных `demo` на сервере, используемом по умолчанию, можно так:

```
$ createdb demo
```

Создать базу `demo` на сервере `eden`, порт 5000, из шаблонной базы `template0` можно такой командой командной строки, за которой стоит следующая команда SQL:

```
$ createdb -p 5000 -h eden -T template0 -e demo
CREATE DATABASE demo TEMPLATE template0;
```

См. также

dropdb, CREATE DATABASE

createuser

createuser — создать новую учётную запись Postgres Pro

Синтаксис

`createuser` [*параметр-подключения*...] [*параметр*...] [*имя_пользователя*]

Описание

createuser создаёт нового пользователя Postgres Pro, а если точнее — роль. Лишь суперпользователь и пользователи с привилегией `CREATEROLE` могут создавать новые роли, таким образом, createuser должна запускаться от их лица.

Чтобы создать роль с правом `SUPERUSER`, `REPLICATION` или `BYPASSRLS`, необходимо подключиться от имени существующего суперпользователя, одного лишь права `CREATEROLE` недостаточно. Поскольку суперпользователи могут обходить все ограничения доступа в базе данных, к назначению этих полномочий не следует относиться легкомысленно.

createuser — это обёртка для SQL-команды `CREATE ROLE`. Создание пользователей с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

createuser принимает следующие аргументы:

имя_пользователя

Задаёт имя создаваемого пользователя Postgres Pro. Это имя должно отличаться от имён всех существующих ролей в данной инсталляции Postgres Pro.

`-a` *role*
`--with-admin=`*role*

Указывает существующую роль, которая автоматически добавляется в качестве члена новой роли с правом `ADMIN OPTION`, что позволяет ей выдавать членство в новой роли другим ролям. Можно указать несколько существующих ролей, добавив ключ `-a` несколько раз.

`-c` *номер*
`--connection-limit=`*номер*

Устанавливает максимальное допустимое количество соединений для создаваемого пользователя. По умолчанию ограничение в количестве соединений отсутствует.

`-d`
`--createdb`

Разрешает новому пользователю создавать базы данных.

`-D`
`--no-createdb`

Запрещает новому пользователю создавать базы данных. Это поведение по умолчанию.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении createuser.

`-E`
`--encrypted`

Параметр является устаревшим, но в целях совместимости ещё работает.

`-g` *`role`*
`--member-of=`*`role`*
`--role=`*`роль`* (устаревший аргумент)

Указывает, что новая роль должна быть автоматически добавлена в качестве члена указанной существующей группы. Можно указать несколько существующих ролей, добавив ключ `-g` несколько раз.

`-i`
`--inherit`

Создаваемая роль автоматически унаследует права ролей, в которые она включается. Это поведение по умолчанию.

`-I`
`--no-inherit`

Роль не будет наследовать права ролей, в которые она включается.

`--interactive`

Запросить имя для создаваемого пользователя, а также значения для флагов `-d`/`-D`, `-r`/`-R`, `-s`/`-S`, если они явно не указаны в командной строке. До версии PostgreSQL 9.1 включительно это было поведением по умолчанию.

`-l`
`--login`

Новый пользователь сможет подключаться к серверу (то есть его имя может быть идентификатором начального пользователя сеанса). Это свойство по умолчанию.

`-L`
`--no-login`

Новый пользователь не сможет подключаться к серверу. (Роль без права входа на сервер тем не менее полезна для управления разрешениями в базе данных.)

`-m` *`role`*
`--with-member=`*`role`*

Указывает существующую роль, которая автоматически добавляется в качестве члена новой роли. Можно указать несколько существующих ролей, добавив ключ `-m` несколько раз.

`-P`
`--pwprompt`

Если флаг указан, то createuser запросит пароль для создаваемого пользователя. Если не планируется аутентификация по паролю, то пароль можно не устанавливать.

`-r`
`--createrole`

Новый пользователь сможет создавать, изменять и удалять другие роли, добавлять для них комментарии и изменять метку безопасности (то есть этот пользователь будет иметь право `CREATEROLE`). Подробнее возможности, предоставляемые этим правом, описаны в Создание роли.

`-R`
`--no-createrole`

Запрещает пользователю создавать новые роли. Это поведение по умолчанию.

`-s`
`--superuser`

Создаваемая роль будет иметь права суперпользователя.

`-S`
`--no-superuser`

Новый пользователь не будет суперпользователем. Это поведение по умолчанию.

`-v` *`отметка_времени`*
`--valid-until=`*`отметка_времени`*

Задаёт дату и время, после которого пароль роли перестаёт действовать. По умолчанию срок действия пароля будет неограниченным.

`-V`
`--version`

Вывести версию createuser и завершить выполнение.

`--bypassrls`

Новый пользователь не подчиняется никаким политикам защиты на уровне строк (RLS).

`--no-bypassrls`

На нового пользователя распространяются политики защиты на уровне строк (RLS). Это поведение по умолчанию.

`--profile=`*`профиль`*

Новой роли будет назначен профиль, применяющий политику управления паролями. Все ограничения этого профиля будут применяться к роли.

`--replication`

Создаваемый пользователь будет наделён правом `REPLICATION`. Это рассмотрено подробнее в документации по CREATE ROLE.

`--no-replication`

Создаваемый пользователь не будет иметь привилегии `REPLICATION`, что рассмотрено подробнее в документации по CREATE ROLE. Это поведение по умолчанию.

`-?`
`--help`

Вывести помощь по команде createuser.

createuser также принимает из командной строки параметры подключения:

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя для подключения (не имя создаваемого пользователя).

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как createuser запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, createuser лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Переменные окружения

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Диагностика

В случае возникновения трудностей, обратитесь к CREATE ROLE и psql. Переменные окружения и параметры подключения по умолчанию libpq будут применены при запуске утилиты, это следует учитывать при диагностике.

Примеры

Чтобы создать роль `joe` на сервере, используемом по умолчанию:

```
$ createuser joe
```

Чтобы создать роль `joe` на сервере, используемом по умолчанию, с запросом дополнительных параметров:

```
$ createuser --interactive joe
Назначить роль суперпользователем? (y/n) n
Разрешить новой роли создавать базы данных? (y/n) n
Разрешить новой роли создавать другие роли? (y/n) n
```

Чтобы создать того же пользователя `joe` с явно заданными атрибутами, подключившись к компьютеру `eden`, порту 5000:

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

Чтобы создать роль `joe` с правами суперпользователя и предустановленным паролем:

```
$ createuser -P -s -e joe
Введите пароль для новой роли: xyzzy
Повторите его: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB
 CREATEROLE INHERIT LOGIN;
```

В приведённом примере введённый пароль отображается лишь для отражения сути, на деле же он не выводится на экран. Как можно видеть, он шифруется прежде чем передаётся в команде клиенту.

См. также

dropuser, CREATE ROLE, createrole_self_grant

dropdb

dropdb — удалить базу данных Postgres Pro

Синтаксис

`dropdb` [*`параметр-подключения`*...] [*`параметр`*...] *`dbname`*

Описание

dropdb удаляет ранее созданную базу данных Postgres Pro, и должна выполняться от имени суперпользователя или её владельца.

dropdb это обёртка для SQL-команды `DROP DATABASE`. Удаление баз данных с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

dropdb принимает в качестве аргументов:

`dbname`

Указывает имя удаляемой базы данных.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении dropdb.

`-f`
`--force`

Попытаться принудительно завершить все существующие подключения к целевой базе, прежде чем удалять её. Подробнее это описано в DROP DATABASE.

`-i`
`--interactive`

Выводит вопрос о подтверждении перед удалением.

`-V`
`--version`

Выводит версию dropdb.

`--if-exists`

Не считать ошибкой, если база данных не существует. В этом случае будет выдано замечание.

`-?`
`--help`

Вывести справку по команде dropdb.

dropdb также принимает из командной строки параметры подключения:

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как dropdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, dropdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*`dbname`*

Указывает имя опорной базы данных, к которой будет произведено подключение для удаления целевой. Если имя не указано, будет выбрана база `postgres`, а если она не существует (или именно она и удаляется) — `template1`. Здесь может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

Переменные окружения

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Диагностика

В случае возникновения трудностей, обратитесь к DROP DATABASE и psql. При диагностике следует учесть, что при запуске утилиты также применяются переменные окружения и параметры подключения по умолчанию libpq.

Примеры

Для удаления базы данных `demo` на сервере, используемом по умолчанию:

```
$ dropdb demo
```

Для удаления базы данных `demo` на сервере `eden`, слушающим подключения на порту 5000, в интерактивном режиме и выводом запросов к серверу:

```
$ dropdb -p 5000 -h eden -i -e demo
База данных "demo" будет удалена навсегда.
Продолжить? (y/n) y
DROP DATABASE demo;
```

См. также

createdb, DROP DATABASE

dropuser

dropuser — удалить учётную запись пользователя Postgres Pro

Синтаксис

`dropuser` [*параметр-подключения*...] [*параметр*...] [*имя_пользователя*]

Описание

dropuser удаляет ранее созданного пользователя PostgreSQL. Суперпользователи могут удалить любую роль. Удалять роли не суперпользователей могут пользователи с привилегией `CREATEROLE`, получившие право `ADMIN OPTION` для целевой роли.

dropuser это обёртка для SQL-команды `DROP ROLE`. Удаление пользователей с её помощью по сути не отличается от выполнения того же действия при обращении к серверу другими способами.

Параметры

dropuser принимает в качестве аргументов:

имя_пользователя

Указывает имя удаляемой роли Postgres Pro. Если передан флаг `-i`/`--interactive`, а имя не указано в параметрах команды, его необходимо будет ввести интерактивно.

`-e`
`--echo`

Вывести команды к серверу, генерируемые при выполнении dropuser.

`-i`
`--interactive`

Вывести подтверждение об удалении роли, и запросить её имя, если оно не указано в параметрах команды.

`-V`
`--version`

Вывести версию dropuser.

`--if-exists`

Перехватить ошибку, если пользователь не существует. В этом случае вместо ошибки будет выведено информационное сообщение.

`-?`
`--help`

Вывести справку по команде dropuser.

dropuser также принимает из командной строки параметры подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится текущее подключение к базе.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как dropuser запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, dropuser лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Переменные окружения

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Диагностика

В случае возникновения трудностей, обратитесь к DROP ROLE и psql. При диагностике следует учесть, что при запуске утилиты также применяются переменные окружения и параметры подключения по умолчанию libpq.

Примеры

Чтобы удалить роль `joe` на сервере, используемом по умолчанию:

```
$ dropuser joe
```

Чтобы удалить роль `joe` на сервере `eden`, слушающем подключения на порту 5000, в интерактивном режиме и с выводом выполняемых команд:

```
$ dropuser -p 5000 -h eden -i -e joe
Роль "joe" будет удалена навсегда.
Продолжить? (y/n) y
DROP ROLE joe;
```

См. также

createuser, DROP ROLE

ecpg

ecpg — встроенный C-препроцессор SQL

Синтаксис

`ecpg` [*параметр*...] *файл*...

Описание

`ecpg` — это встроенный SQL препроцессор для программ, написанных на языке C. Он преобразует программы на C, содержащие SQL-выражения, заменяя их вызовами встроенных функций. Получаемые на выходе файлы можно затем скомпилировать и скомпоновать.

`ecpg` преобразует каждый файл, переданный в параметрах, в соответствующий файл на C. Если имя входного файла указано без расширения, подразумевается `.pgc`. Указанное имя, но с расширением `.c`, становится по умолчанию именем выходного файла. Переопределить имя выходного файла можно, воспользовавшись параметром `-o`.

Если в качестве имени входного файла указано `-`, `ecpg` читает программу с устройства стандартного ввода (и выводит результат в стандартный вывод, если не задан параметр `-o`).

Данный раздел не содержит описания встроенного SQL-языка. Для более подробной информации см. Главу 34.

Параметры

`ecpg` принимает в качестве аргументов:

`-c`

Автоматически генерировать код, написанный на языке C, из кода SQL. Сейчас это справедливо для `EXEC SQL TYPE`.

`-C` *режим*

Установить режим совместимости; *режим* может принимать значения: `INFORMIX`, `INFORMIX_SE` и `ORACLE`.

`-D` *символ*[=*значение*]

Определить символ начала команд препроцессора, аналогично использованию директивы `EXEC SQL DEFINE`. Если *значение* не указано, символ определяется `1`.

`-h`

Обрабатывать заголовочные файлы. Когда добавляется этот параметр, расширением выходного файла становится не `.c`, а `.h`, и расширением входного файла по умолчанию считается не `.pgc`, а `.pgh`. Кроме этого с данным параметром подразумевается `-c`.

`-i`

Также разбирать и системные включения.

`-I` *каталог*

Указать дополнительный путь включаемых файлов, используемый при выполнении `EXEC SQL INCLUDE`. По умолчанию используются `.` (текущий каталог), `/usr/local/include`, каталог, задаваемый при компиляции Postgres Pro (обычно — `/usr/local/pgsql/include`), и `/usr/include`, в порядке, как это перечислено.

`-o` *`имя_файла`*

Задаёт *`имя_файла`*, в который программа `ecpg` должна выводить результат. Указание `-o -` направляет результат в устройство стандартного вывода.

`-r` *`параметр`*

Определяет поведение времени выполнения. *`Флаг`* может принимать следующие значения:

`no_indicator`

Использовать специальные символы для представления значений null. Исторически некоторые базы данных используют такой подход.

`prepare`

Сформировать подготовленные выражения. libecpg сформирует кеш подготовленных выражений и будет использовать их при необходимости повторно. В случае переполнения кеша, libecpg освободит память за счёт вытеснения наименее используемых выражений.

`questionmarks`

Разрешает использовать знак вопроса в качестве аргумента подстановки в целях совместимости. Ранее это было поведением по умолчанию.

`-t`

Включить автоматическую фиксацию транзакций. В этом режиме каждая SQL-команда будет автоматически фиксироваться, пока не будет явно включена в блок транзакции. В режиме по умолчанию команды фиксируются лишь при явном вызове `EXEC SQL COMMIT`.

`-v`

Вывести информацию о версии, а также путях поиска включаемых файлов.

`--version`

Вывести версию ecpg.

`-?`
`--help`

Вывести справку по команде ecpg.

Примечания

При компиляции полученных файлов, компилятор должен иметь возможность найти заголовочные файлы ECPG в каталоге включений Postgres Pro. Для этого можно использовать флаг `-I` во время компиляции, например, `-I/usr/local/pgsql/include`.

Программы на C со встроенным SQL необходимо скомпоновать с библиотекой `libecpg`, например, используя флаг компоновщика `-L/usr/local/pgsql/lib -lecpg`.

Имена каталогов, подходящих для установки, можно найти в разделе pg_config.

Примеры

Если имеется исходный файл на C `prog1.pgc` со встроенным SQL, можно создать исполняемую программу, используя следующую последовательность команд:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_amcheck

pg_amcheck — проверить одну или несколько БД Postgres Pro на предмет повреждения

Синтаксис

`pg_amcheck` [*параметр*...] [*dbname*]

Описание

Программа pg_amcheck поддерживает запуск реализованных в расширении amcheck функций, выявляющих повреждения, в одной или нескольких базах данных, с возможностью выбора проверяемых индексов, таблиц и схем. Она также позволяет выбрать, какие проверки будут выполняться, и запустить их в параллельном режиме (для которого задаётся число параллельных подключений).

В настоящее время поддерживаются только обычные табличные отношения, таблицы TOAST, материализованные представления, последовательности и индексы btree. Другие типы отношений просто пропускаются.

Если задаётся `имя_бд`, это должно быть имя одной проверяемой базы данных, а никакие другие параметры выбора базы данных задаваться не должны. Если же, напротив, присутствуют параметры выбора баз данных, будут проверяться все соответствующие базы. В отсутствие этих параметров проверяться будет только база по умолчанию. Параметрами, выбирающими базы, являются `--all`, `--database` и `--exclude-database`. Также в их число можно включить `--relation`, `--exclude-relation`, `--table`, `--exclude-table`, `--index` и `--exclude-index`, но только если их значения задаются тремя компонентами (например, `mydb*.myschema*.myrel*`). И наконец, к таким параметрам можно отнести `--schema` и `--exclude-schema`, если их значения задаются двумя компонентами (например, `mydb*.myschema*`).

В качестве параметра *имя_бд* может также передаваться строка подключения.

Параметры

Следующие параметры командной строки определяют, что будет проверяться:

`-a`
`--all`

Проверить все базы данных (кроме исключённых аргументом `--exclude-database`).

`-d` *шаблон*
`--database=`*шаблон*

Проверить базы данных, соответствующие заданному *шаблону*, кроме исключённых аргументом `--exclude-database`. Этот параметр можно добавлять неоднократно.

`-D` *шаблон*
`--exclude-database=`*шаблон*

Исключить базы данных, соответствующие заданному *шаблону*. Этот параметр можно добавлять неоднократно.

`-i` *шаблон*
`--index=`*шаблон*

Проверить индексы, соответствующие заданному *шаблону*, если они не исключены каким-либо образом. Этот параметр можно добавлять неоднократно.

Этот параметр подобен параметру `--relation`, но применяется только к индексам, а не к другим типам отношений.

`-I` *`шаблон`*
`--exclude-index=`*`шаблон`*

Исключить индексы, соответствующие заданному *`шаблону`*. Этот параметр можно добавлять неоднократно.

Этот параметр подобен параметру `--exclude-relation`, но применяется только к индексам, а не к другим типам отношений.

`-r` *`шаблон`*
`--relation=`*`шаблон`*

Проверить отношения, соответствующие заданному *`шаблону`*, если они не исключены каким-либо образом. Этот параметр можно добавлять неоднократно.

Шаблоны могут быть неполными, например `myrel*`, либо они могут задаваться с указанием схемы, например `myschema*.myrel*`, а также с указанием базы и схемы, например `mydb*.myschema*.myrel*`. В случае указания шаблона с базой данных, все соответствующие базы будут добавлены в список баз, подлежащих проверке.

`-R` *`шаблон`*
`--exclude-relation=`*`шаблон`*

Исключить отношения, соответствующие заданному *`шаблону`*. Этот параметр можно добавлять неоднократно.

Как и с `--relation`, *`шаблон`* может быть неполным или дополненным указанием схемы, а также указанием базы и схемы.

`-s` *`шаблон`*
`--schema=`*`шаблон`*

Проверить таблицы и индексы в схемах, соответствующих заданному *`шаблону`*, если соответствующие объекты не исключены каким-либо образом. Этот параметр можно добавлять неоднократно.

Чтобы проверить только таблицы в схемах, соответствующих определённому шаблону, можно указать в параметрах `--table=ШАБЛОН_СХЕМЫ.* --no-dependent-indexes`. Чтобы выбрать только индексы, можно воспользоваться указанием `--index=ШАБЛОН_СХЕМЫ.*`.

Шаблон схемы может содержать указание базы. Например, вы можете написать `--schema=mydb*.myschema*`, чтобы выбрать схемы `myschema*` в базах данных, соответствующих шаблону `mydb*`.

`-S` *`шаблон`*
`--exclude-schema=`*`шаблон`*

Исключить таблицы и индексы в схемах, соответствующих заданному *`шаблону`*. Этот параметр можно добавлять неоднократно.

Как и в параметре `--schema`, к шаблону можно добавить имя базы данных.

`-t` *`шаблон`*
`--table=`*`шаблон`*

Проверить таблицы в схемах, соответствующих заданному *`шаблону`*, если соответствующие объекты не исключены каким-либо образом. Этот параметр можно добавлять неоднократно.

Этот параметр подобен параметру `--relation`, но применяется только к таблицам, материализованным представлениям и последовательностям, а не к индексам.

`-T` *`шаблон`*
`--exclude-table=`*`шаблон`*

Исключить таблицы, соответствующие заданному *`шаблону`*. Этот параметр можно добавлять неоднократно.

Этот параметр подобен параметру `--exclude-relation`, но применяется только к таблицам, материализованным представлениям и последовательностям, а не к индексам.

`--no-dependent-indexes`

По умолчанию, если проверяется таблица, также будут проверяться все индексы btree этой таблицы, даже если они не были явно выбраны ключами типа `--index` или `--relation`. Данный параметр отключает это поведение.

`--no-dependent-toast`

По умолчанию, если проверяется таблица, также будет проверяться её TOAST-таблица (если таковая имеется), даже если она не была явно выбрана ключами типа `--table` или `--relation`. Данный параметр отключает это поведение.

`--no-strict-names`

По умолчанию, если аргументу `--database`, `--table`, `--index` или `--relation` не соответствуют никакие объекты, это считается критической ошибкой. С данным параметром уровень ошибки понижается до предупреждения.

Следующие параметры командной строки управляют методами проверки таблиц:

`--exclude-toast-pointers`

По умолчанию, когда в таблице встречается указатель на TOAST, осуществляется поиск записи в TOAST-таблице для проверки корректности указателя. Эти проверки могут быть довольно медленными, и данный параметр позволяет пропустить их.

`--on-error-stop`

Прекращать обработку табличного отношения сразу после сообщения обо всех повреждениях на первой повреждённой странице и переходить к следующей таблице или индексу.

Заметьте, что проверка индекса всегда прекращается после обнаружения первой повреждённой страницы. Данный параметр имеет смысл только для табличных отношений.

`--skip=`*`параметр`*

С указанием `all-frozen` проверки повреждений таблиц будут пропускать страницы, помеченные как полностью замороженные, во всех таблицах.

С указанием `all-visible` проверки повреждений таблиц будут пропускать страницы, помеченные как полностью видимые, во всех таблицах.

По умолчанию никакие страницы не пропускаются. Такое поведение задаётся значением `none`, но так как это значение подразумевается, явно задавать его не требуется.

`--startblock=`*`блок`*

Начать проверку с блока с заданным номером. Если проверяемое табличное отношение содержит меньше заданного числа блоков, будет выдана ошибка. Данный параметр представляется полезным только для проверки одной конкретной таблицы, и он не действует на индексы. Дополнительные замечания приведены в описании `--endblock`.

`--endblock=`*`блок`*

Завершить проверку на блоке с указанным номером. Если проверяемое табличное отношение содержит меньше заданного числа блоков, будет выдана ошибка. Данный параметр представляется полезным только для проверки одной конкретной таблицы, и он не действует на индексы. Если проверяется и обычная, и TOAST-таблица, этот параметр действует на обе, но при проверке TOAST-указателей тем не менее возможны обращения к блокам за этим пределом, если только эта проверка не была выключена параметром `--exclude-toast-pointers`.

Следующие параметры командной строки управляют методами проверки индексов-B-деревьев:

`--checkunique`

Проверить для каждого индекса с ограничением уникальности, что в нём нет одновременно видимых дубликатов, используя параметр `checkunique` в amcheck.

`--heapallindexed`

Проверять для каждого обрабатываемого индекса наличие всех кортежей кучи в виде индексных кортежей с использованием режима `heapallindexed` проверки amcheck.

`--parent-check`

Выполнять для каждого проверяемого индекса btree функцию amcheck `bt_index_parent_check`, проводящую дополнительные проверки связей родитель-потомок.

По умолчанию выполняется функция amcheck `bt_index_check`, но заметьте, что в случае использования параметра `--rootdescend` неявно выбирается функция `bt_index_parent_check`.

`--rootdescend`

При проверке каждого индекса для каждого кортежа заново находить индексные кортежи на уровне листьев, производя поиск с корневой страницы, то есть задействовать режим amcheck `rootdescend`.

При использовании данного параметра также неявно включается параметр `--parent-check`.

Этот режим проверки изначально реализовывался как средство, полезное при разработке функциональности индексов-B-деревьев. Он может обнаруживать не все или вовсе не обнаруживать те типы повреждений, которые встречаются на практике. Также в этом режиме проверка выполняется значительно дольше и для неё требуется больше серверных ресурсов.

Предупреждение

Дополнительные проверки, выполняемые для индексов-B-деревьев, когда указан параметр `--parent-check` или параметр `--rootdescend`, требуют относительно сильных блокировок на уровне отношений. Только эти проверки блокируют одновременное изменение данных командами `INSERT`, `UPDATE` и `DELETE`.

Следующие параметры командной строки управляют подключением к серверу:

`-h` *`компьютер`*
`--host=`*`компьютер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U`
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_amcheck запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_amcheck лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*`dbname`*

Задаёт базу данных или строку подключения для установления подключения, через которое будет определяться список баз данных для проверки. Если не используются ни ключ `--all`, ни параметры, задающие шаблон имён проверяемых баз данных, такое подключение не требуется и данный параметр игнорируется. Иначе все параметры из переданной строки, за исключением имени базы данных, будут также использоваться при подключении к проверяемым базам. Если этот параметр опущен, выполняется подключение к базе `postgres`, а если к ней подключиться не удаётся — к базе `template1`.

Другие параметры:

`-e`
`--echo`

Выводить в stdout все SQL-запросы, передаваемые серверу.

`-j` *`число`*
`--jobs=`*`число`*

Использовать заданное *`число`* одновременных подключений к серверу, а если оно превышает количество объектов, число подключений ограничивается этим количеством.

По умолчанию используется одно подключение.

`-P`
`--progress`

Выводить информацию о прогрессе операции. Выводимая информация включает количество отношений, для которых была проведена проверка, и общий размер этих отношений. В неё также включается общее количество отношений, которые должны быть проверены, и примерный размер этих отношений.

`-v`
`--verbose`

Выводить больше сообщений. В частности, будут выводиться сообщения о проверке каждого отношения, а также увеличится уровень детализации ошибок сервера.

`-V`
`--version`

Вывести версию pg_amcheck и завершиться.

`--install-missing`
`--install-missing=`*`схема`*

Установить все отсутствующие расширения, которые требуются для проверки баз(ы) данных. Если это расширение не было установлено, его объекты будут помещены в заданную *`схему`* или, если она не задана, в схему `pg_catalog`.

В настоящее время для работы pg_amcheck требуется только расширение amcheck.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_amcheck и завершиться.

Переменные окружения

`pg_amcheck`, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Программа pg_amcheck предназначена для работы с Postgres Pro 14.0 и выше.

См. также

amcheck

pg_basebackup

pg_basebackup — создать резервную копию кластера Postgres Pro

Синтаксис

`pg_basebackup` [*параметр*...]

Описание

Программа pg_basebackup предназначена для создания базовых копий работающего кластера баз данных Postgres Pro. Процедура создания копии не влияет на работу других клиентов базы. Полученные копии могут использоваться и для восстановления на момент времени (см. Раздел 24.3), и в качестве базового состояния ведомого сервера при реализации трансляции журнала или потоковой репликации (см. Раздел 25.2).

pg_basebackup может выполнять полное или инкрементальное базовое резервное копирование базы данных. При полном резервном копировании создаётся точная копия файлов кластера баз данных. При инкрементальном резервном копировании некоторые файлы, которые были бы включены в полную резервную копию, могут заменяться инкрементальными версиями тех же файлов, содержащими только те блоки, которые были изменены с момента исходной резервной копии. Инкрементальную копию нельзя восстановить напрямую: сначала необходимо использовать pg_combinebackup, чтобы объединить её с предыдущими резервными копиями, от которых она зависит. См. Подраздел 24.3.3 для получения дополнительной информации об инкрементальных резервных копиях и Подраздел 24.3.5 для пошагового восстановления из резервной копии.

В любом режиме pg_basebackup резервное копирование включается и завершается автоматически. Резервные копии всегда создаются для кластера целиком; создать копию отдельных баз данных или объектов базы данных нельзя. Для выборочного копирования нужно использовать другие средства, например pg_dump.

Копия создаётся через обычное подключение к Postgres Pro, и при этом используется протокол репликации. Соединение с сервером должен устанавливать пользователь с правом `REPLICATION` (см. Раздел 20.2) или суперпользователь, а в `pg_hba.conf` должно разрешаться подключение для репликации. Кроме того, на сервере должно быть установлено достаточно большое значение max_wal_senders, чтобы мог запуститься минимум один передатчик WAL для резервного копирования и ещё один для потоковой трансляции WAL (если она применяется).

Можно запустить одновременно несколько команд `pg_basebackup`, но с точки зрения производительности обычно эффективнее делать всего одну копию одновременно, а затем копировать получаемый результат.

С помощью pg_basebackup можно получить базовую копию не только с ведущего, но и с ведомого сервера. Чтобы сделать копию с ведомого, настройте его, чтобы он мог принимать подключения репликации (для этого нужно установить подходящие значения `max_wal_senders`, hot_standby и подкорректировать `pg_hba.conf`). При этом на ведущем необходимо также включить full_page_writes.

Заметьте, что при копировании с ведомого сервера есть некоторые ограничения:

- Файл истории резервного копирования в целевом кластере баз данных не создаётся.
- pg_basebackup не может принудительно переключить ведомый сервер на новый файл WAL в конце копирования. Поэтому, если используется режим `-X none` и активность записи на ведущем сервере низкая, pg_basebackup может довольно долго ждать переключения и архивирования последнего файла WAL, необходимого для полноты копии. В этом случае может иметь смысл выполнить на ведущем сервере `pg_switch_wal` для немедленного переключения файла WAL.
- Если ведомый сервер повышается и становится ведущим в процессе копирования, копирование прерывается.

- Все необходимые для резервной копии WAL-записи должны содержать полные страницы, для чего нужно включить режим `full_page_writes` на ведущем сервере.

Когда pg_basebackup создаёт базовую копию, в представлении `pg_stat_progress_basebackup` отображается состояние этого процесса. За подробностями обратитесь к Подразделу 26.4.6.

Параметры

Следующие аргументы командной строки задают расположение и формат вывода:

`-D` *`каталог`*
`--pgdata=`*`каталог`*

Целевой каталог, куда будет записана копия. Если он не существует, pg_basebackup создаст его, а также отсутствующие родительские каталоги, при необходимости. Если он существует, он должен быть пустым.

Если копия создаётся в формате tar, в качестве целевого каталога можно задать - (минус), и тогда файл tar будет записан в `stdout`.

Этот флаг является обязательным.

`-F` *`формат`*
`--format=`*`формат`*

Устанавливает формат вывода. *`формат`* может принимать следующие значения:

`p`
`plain`

Записывает выводимые данные в обычные файлы, сохраняя структуру каталогов данных и табличных пространств как на исходном сервере. Если в кластере нет дополнительных табличных пространств, вся база будет помещена в заданный каталог. Иначе основной каталог хранения данных будет помещён в целевой каталог, а все остальные табличные пространства — в те же абсолютные пути, в которых они располагаются на исходном сервере. (Чтобы изменить эти пути, воспользуйтесь параметром `--tablespace-mapping`).

Это формат по умолчанию.

`t`
`tar`

Записывает в целевой каталог файлы в формате tar. Содержимое основного каталога данных будет записано в файл `base.tar`, а каждое дополнительное табличное пространство — в отдельный файл, содержащий в имени OID этого пространства.

Если в качестве имени целевого каталога задано - (минус), содержимое tar будет записано в устройство стандартного вывода, что позволяет, например, передать его программе gzip. Это возможно, только если в кластере нет дополнительных табличных пространств и не используется трансляция WAL.

`-i` *`old_manifest_file`*
`--incremental=`*`old_manifest_file`*

Выполнять инкрементальное резервное копирование. Должен быть предоставлен манифест резервной копии, на основе которой будет выполняться инкрементальное копирование; этот манифест будет загружен на сервер, который затем отправит запрошенную инкрементальную резервную копию.

`-R`
`--write-recovery-conf`

Создать файл `standby.signal` и добавить параметры конфигурации в файл `postgresql.auto.conf` в целевом каталоге (или внутри архива, если используется формат tar). Это упрощает настройку ведомого сервера при восстановлении этой копии.

В файл `postgresql.auto.conf` будут записаны параметры соединения и слот репликации, если его использует pg_basebackup, так что впоследствии при потоковой репликации и синхронизации слотов логической репликации будут использоваться те же параметры. Имя базы данных будет записано, только если оно явно указано в строке подключения или в переменной среды.

`-t` *`получатель`*
`--target=`*`получатель`*

Указывает серверу, куда отправлять созданную копию. Получателем по умолчанию является `client`, то есть резервная копия будет отправлена на компьютер, где работает pg_basebackup. Если же в качестве получателя установлено `server:/some/path`, резервная копия будет сохранена на компьютере, где работает сервер, в каталоге `/some/path`. Для сохранения резервной копии на сервере требуются права суперпользователя или роли `pg_write_server_files`. Если в качестве получателя указать `blackhole`, содержимое копии отправляется в «чёрную дыру» и нигде не сохраняется. Этот вариант следует использовать только в целях тестирования, так как собственно резервной копии вы не получите.

Так как приём потока WAL реализован в pg_basebackup, а не на стороне сервера, этот параметр нельзя использовать вместе с `-Xstream`. Так как режим `-Xstream` используется по умолчанию, указывая данный параметр, необходимо также указать либо `-Xfetch`, либо `-Xnone`.

`-T` *`старый_каталог=новый_каталог`*
`--tablespace-mapping=`*`старый_каталог=новый_каталог`*

Переместить табличное пространство из *`старого_каталога`* в *`новый_каталог`* в процессе копирования. Чтобы перемещение произошло, в параметре *`старый_каталог`* путь табличного пространства должен задаваться в точности так, как он определён на исходном сервере. (Но не будет ошибкой, если на исходном сервере не окажется табличного пространства, на которое указывает *`старый_каталог`*.) В то же время, *`новый_каталог`* задаёт путь в файловой системе получающего сервера. Как и основной целевой каталог, *`новый_каталог`* может не существовать, но если он существует, он должен быть пустым. И *`старый_каталог`*, и *`новый_каталог`* должны задаваться абсолютными путями. Если в пути встречается символ `=`, его необходимо экранировать обратной косой чертой. Этот параметр можно добавить несколько раз для нескольких табличных пространств.

Если табличное пространство перемещается таким способом, символические ссылки внутри основного каталога хранения данных также приводятся в соответствие с новым местоположением. Таким образом, для экземпляра сервера подготавливается новый каталог данных, в котором все табличные пространства оказываются в новом расположении.

В настоящее время этот параметр работает только с обычным форматом вывода; если выбран формат tar, параметр игнорируется.

`--waldir=`*`каталог_wal`*

Задать каталог, в который будут записаны файлы WAL (журнал предзаписи). По умолчанию файлы WAL будут записываться в подкаталог `pg_wal` целевого каталога, но с помощью этого параметра их можно поместить в любое место. Путь *`каталог_wal`* должен быть абсолютным. Как и основной целевой каталог, *`каталог_wal`* может не существовать, но если он существует, он должен быть пустым. Этот параметр можно задать, только если копия создаётся в простом формате.

`-X` *`метод`*
`--wal-method=`*`метод`*

Включает в резервную копию все необходимые файлы журнала предзаписи (файлы WAL). В том числе включаются все файлы журнала, сгенерированные в процессе создания резервной копии. Любой метод, кроме `none`, позволяет запустить сервер с восстановленным каталогом, не используя архив WAL; таким образом будет получена полностью самодостаточная резервная копия.

Поддерживаются следующие *`методы`* получения журналов предзаписи:

`n`
`none`

Не включать журналы предзаписи в резервную копию.

`f`
`fetch`

Файлы журнала предзаписи собираются в конце процесса копирования. Таким образом необходимо установить достаточно большое значение параметра wal_keep_size, чтобы избежать преждевременного удаления нужных данных журнала. В случае переработки этих данных до завершения процесса копирования возникнет ошибка, а копия будет непригодной к использованию.

Когда используется формат tar, файлы журнала предзаписи включаются в архив `base.tar`.

`s`
`stream`

Передавать журнал предзаписи в процессе создания резервной копии. При выборе этого метода открывается второе соединение к серверу, через которое будет передаваться журнал предзаписи параллельно с созданием копии. Таким образом, этот метод требует использования не одного, а двух соединений репликации, но если клиент будет успевать получать данные журнала предзаписи, на исходном сервере не потребуется сохранять дополнительные журналы.

Когда используется формат tar, файлы журнала предзаписи сохраняются в отдельном архиве с именем `pg_wal.tar` (если версия сервера ниже 10, файл будет называться `pg_xlog.tar`).

Это значение по умолчанию.

`-z`
`--gzip`

Включает gzip-сжатие выводимого tar-файла с уровнем компрессии по умолчанию. Сжатие поддерживается только для формата tar, при этом ко всем именам файлов tar добавляется суффикс `.gz`.

`-Z` *`уровень`*
`-Z [{client|server}-]`*`метод`*`[:`*`дополнительная_информация`*`]`
`--compress=`*`уровень`*
`--compress=[{client|server}-]`*`метод`*`[:`*`дополнительная_информация`*`]`

Запрашивает сжатие резервной копии. Если добавляется префикс `client` или `server`, он указывает, где должно выполняться сжатие. Когда сжатие выполняет сервер, объём передаваемых данных уменьшается, но увеличивается нагрузка на процессор сервера. По умолчанию используется `client`, за исключением случаев, когда используется ключ `--target`. В этих случаях резервная копия не передаётся клиенту, поэтому смысл имеет только сжатие на стороне сервера. В режиме `-Xstream` (выбираемом по умолчанию) сжатие на стороне сервера не распространяется на WAL. Чтобы WAL сжимался, примените сжатие на стороне клиента или используйте режим `-Xfetch`.

В качестве метода сжатия можно выбрать `gzip`, `lz4`, `zstd`, `none` без сжатия или целое число (если задать 0 — без сжатия, если больше нуля — `gzip`). В качестве дополнительной информации можно передать параметры сжатия. Если в строке информации передаётся целое число, оно задаёт уровень сжатия. В противном случае она должна содержать список элементов, разделённых запятыми, в форме *`ключевое_слово`* или *`ключевое_слово=значение`*. На данный момент поддерживаются ключевые слова `level`, `long` и `workers`. Если метод сжатия задан целым числом, строку информации использовать нельзя.

Если уровень сжатия не указан, будет выбран уровень сжатия по умолчанию. Если указан только уровень сжатия, но не указан метод, будет применяться метод сжатия `gzip`, когда уровень больше 0, а когда уровень равен 0, сжатие не будет выполняться.

Когда формат tar сжимается методом `gzip`, `lz4` или `zstd`, ко всем именам файлов tar автоматически добавляется суффикс `.gz`, `.lz4` или `.zst` соответственно. Когда используется простой формат, сжатие на стороне клиента выполнить нельзя, но при этом можно запросить сжатие на стороне сервера. В таком случае сервер будет сжимать копию для передачи, а клиент будет распаковывать и извлекать из неё данные.

Когда этот параметр используется в сочетании с `-Xstream`, `pg_wal.tar` будет сжат методом `gzip`, если выбрано сжатие gzip на стороне клиента, и не будет сжат, если выбран любой другой алгоритм сжатия или выбрано сжатие на стороне сервера.

Описанные далее аргументы командной строки управляют формированием резервной копии и вызовом программы:

`-c {fast|spread}`
`--checkpoint={fast|spread}`

Устанавливает режим контрольных точек: fast (быстрый) или spread (протяжённый, по умолчанию). Подробнее см. Подраздел 24.3.4.

`-C`
`--create-slot`

Указывает, что до начала копирования должен быть создан слот репликации с именем, заданным в `--slot`. Если такой слот уже существует, выдаётся ошибка.

`-l` *`метка`*
`--label=`*`метка`*

Устанавливает метку для созданной резервной копии. Если не указана, то по умолчанию будет использовано значение «`pg_basebackup base backup`».

`-n`
`--no-clean`

По умолчанию, когда программа `pg_basebackup` прерывается с ошибкой, она удаляет все каталоги, которые она могла создать, прежде чем обнаружила, что не может завершить задание (например, целевой каталог и каталог журнала предзаписи). Данный ключ отключает эту очистку и тем самым полезен для отладки.

Заметьте, что каталоги табличных пространств не очищаются в любом случае.

`-N`
`--no-sync`

По умолчанию `pg_basebackup` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_basebackup` завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя операционной системы резервная копия может оказаться испорченной. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

`-P`
`--progress`

Включает отчёт о прогрессе. Если этот режим включён, то во время создания копии будет передаваться примерный процент выполнения. Так как данные в базе могут меняться во время копирования, это значение будет лишь приближённым и может достигать не точно `100%`. В частности, когда в копию включается журнал WAL, конечный размер невозможно предсказать заранее, и в этом случае ожидаемый конечный размер будет увеличиваться, превысив ориентировочный полный размер без WAL.

`-r` *`скорость_передачи`*
`--max-rate=`*`скорость_передачи`*

Задаёт максимальную скорость, с которой данные будут загружаться с исходного сервера. Это может быть полезно для ограничения влияния pg_basebackup на сервер. Значение параметра задаётся в килобайтах в секунду. Для указания мегабайт в секунду нужно добавить к числу суффикс `M`. Так же принимается суффикс `k`, но он ничего не меняет. Допустимые значения лежат в диапазоне от 32 КБ/с до 1024 МБ/с.

Этот параметр всегда оказывает влияние на передачу каталога данных, а на передачу файлов WAL он влияет, только если выбран метод передачи `fetch`.

`-S` *`имя_слота`*
`--slot=`*`имя_слота`*

Этот параметр может применяться только вместе с `-X stream`. Он выбирает слот репликации, который будет использоваться для передачи WAL. Если базовая копия предназначена для использования на ведомом сервере с потоковой репликацией через слот, это же имя слота должно задаваться на ведомом в качестве primary_slot_name. Тем самым гарантируется, что ведущий сервер не удалит никакие необходимые данные WAL после того, как базовая копия будет получена, и до того, как начнётся потоковая репликация на новом ведомом.

В случае отсутствия ключа `-C` требуется, чтобы указанный слот репликации уже существовал.

Если этот ключ не указан и сервер поддерживает временные слоты репликации (они появились в версии 10), для трансляции WAL автоматически используется временный слот репликации.

`--sync-method=`*`метод`*

Со значением `fsync` (по умолчанию) `pg_basebackup` будет рекурсивно открывать и синхронизировать все файлы в каталоге резервных копий. При использовании простого формата поиск файлов будет осуществляться по символическим ссылкам для каталога WAL и каждого настроенного табличного пространства.

В Linux возможен вариант `syncfs`, когда от ОС требуется синхронизировать целиком файловую систему, содержащую каталог резервных копий. При использовании простого формата `pg_basebackup` также синхронизирует файловые системы, содержащие файлы WAL и табличные пространства. Более подробно особенности использования `syncfs` описаны в recovery_init_sync_method.

В режиме `--no-sync` этот параметр не оказывает никакого влияния.

`-v`
`--verbose`

Включает режим подробного вывода. Будет выводится некоторая дополнительная информация при начале и завершении, а также имена обрабатываемых файлов, если включён отчёт о прогрессе.

`--manifest-checksums=`*`алгоритм`*

Задаёт алгоритм контрольных сумм, которые будут рассчитываться для всех файлов, описанных в манифесте копии. В настоящее время поддерживаются алгоритмы `NONE` (отсутствует), `CRC32C`, `SHA224`, `SHA256`, `SHA384` и `SHA512`. По умолчанию применяется `CRC32C`.

Если выбран вариант `NONE`, манифест копии не будет содержать контрольные суммы. С любым другим вариантом в манифест будет записана контрольная сумма каждого файла в копии, рассчитанная по выбранному алгоритму. Помимо этого, в манифесте всегда сохраняется контрольная сумма его содержимого, рассчитанная по алгоритму `SHA256`. Алгоритмы семейства `SHA` требуют значительно больше процессорных ресурсов, чем `CRC32C`, поэтому выбор такого алгоритма может повлечь увеличение времени создания копии.

Функции хеширования SHA обеспечивают криптографическую стойкость хеша каждого файла, и таким образом полезны, когда нужны гарантии, что резервная копия не была модифицирована. С другой стороны, алгоритм CRC32C вычисляет контрольную сумму гораздо быстрее и вполне подходит для выявления изменений, внесённых не злонамеренно, а случайно. Заметьте, что такая защита от злоумышленника, имеющего доступ к сохранённой копии, будет эффективна, только если манифест копии хранится отдельно в безопасном месте или ещё каким-то образом проверяется, что манифест не был изменён с момента создания копии.

Для проверки целостности копии по созданному манифесту можно воспользоваться программой pg_verifybackup.

`--manifest-force-encode`

Принудительно включает шестнадцатеричное кодирование всех имён файлов в манифесте. Если этот параметр не задаётся, в шестнадцатеричном виде кодируются только имена файлов не в кодировке UTF-8. Этот параметр в первую очередь предназначен для проверки того, что средства чтения манифеста могут правильно разобрать такие имена.

`--no-estimate-size`

Отключает расчёт примерного объёма данных, которые будут передаваться в процессе копирования, в результате чего столбец `backup_total` в представлении `pg_stat_progress_basebackup` будет содержать `NULL`.

Без этого указания процесс копирования начнётся с перечисления файлов для подсчёта размера всей базы данных, а затем продолжится отправкой непосредственно данных. Это может немного увеличить общую длительность процесса, в частности, пройдёт больше времени до начала передачи данных. Данный параметр полезен, когда время расчёта объёма оказывается слишком большим.

Этот параметр нельзя использовать вместе с параметром `--progress`.

`--no-manifest`

Отключает создание манифеста копии. Если этот флаг не указан, сервер будет формировать и передавать в составе копии манифест, который может быть проверен с использованием pg_verifybackup. Манифест представляет собой список всех файлов, включённых в копию, за исключением файлов WAL, которые могут быть в неё добавлены. Также в нём сохраняется размер, дата последнего изменения и, возможно, контрольная сумма каждого файла.

`--no-slot`

Предотвращает создание временного слота репликации для резервного копирования.

По умолчанию, если выбрана передача журнала, но имя слота в `-S` не задано, создаётся временный слот репликации (при условии, что это поддерживает исходный сервер).

Основное предназначение этого ключа в том, чтобы можно было сделать базовую резервную копию, когда на сервере нет свободных слотов репликации. Использование слота репликации почти всегда предпочтительнее, так как при этом предотвращается удаление сервером необходимых файлов WAL во время резервного копирования.

`--no-verify-checksums`

Отключает проверку контрольных сумм, если они включены на сервере, с которого делается резервная копия.

По умолчанию контрольные суммы проверяются, и при выявлении их несоответствия выдаётся ненулевой код завершения. Однако базовая резервная копия в этом случае не удаляется, как и с ключом `--no-clean`. Ошибки контрольных сумм также можно просмотреть в представлении `pg_stat_database`.

Далее описаны параметры, управляющие подключением к исходному серверу:

`-d` *`строка_подключения`*
`--dbname=`*`строка_подключения`*

Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Этот параметр называется `--dbname` для согласованности с другими клиентскими приложениями, но так как pg_receivewal не подключается к какой-либо конкретной базе данных, любое имя базы данных в строке подключения игнорируется сервером. Тем не менее переданное таким образом имя базы данных переопределяет имя по умолчанию (`replication`) для поиска пароля соединения репликации в `~/.pgpass`. Подобным образом это имя может использоваться при подключении через средний слой или прокси-сервер для, например, маршрутизации соединения. Имя базы данных также может использоваться при синхронизации слотов логической репликации.

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-s` *`interval`*
`--status-interval=`*`interval`*

Указывает интервал в секундах между сообщениями о состоянии, передаваемыми исходному серверу. Чем меньше указанное значение, тем точнее будет информация о процессе резервного копирования на сервере. Нулевое значение полностью отключает периодическое обновление состояния, хотя эти сообщения будут всё равно посылаться по запросу сервера во избежание отключения по тайм-ауту. Значение по умолчанию — 10 секунд.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Задаёт имя пользователя для подключения.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль нельзя получить другими средствами, например из файла `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к исходному серверу.

Это несущественный параметр, так как pg_basebackup запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_basebackup лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Другие флаги:

`-V`
`--version`

Вывести версию pg_basebackup и завершиться.

`-?`
`--help`

Вывести справку по аргументам командной строки pg_basebackup и завершиться.

Переменные окружения

Как и большинство других утилит Postgres Pro, приложение также использует переменные окружения, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

До начала копирования на исходном сервере необходимо выполнить контрольную точку. И если копирование запускается без ключа `--checkpoint=fast`, это может занять некоторое время, в течение которого pg_basebackup не будет проявлять никакой активности.

Резервная копия будет включать в себя все файлы каталога хранения данных и табличных пространств, а также конфигурационные файлы и прочие файлы, размещённые в каталоге данных, за исключением определённых временных файлов, принадлежащих Postgres Pro, и файлов операционной системы. Однако копируются лишь простые файлы и каталоги, кроме них, сохраняются только символические ссылки на табличные пространства. Символические ссылки, указывающие на определённые каталоги, известные Postgres Pro, копируются как пустые каталоги. Другие символические ссылки и файлы спецустройств игнорируются. За дополнительными подробностями обратитесь к Разделу 53.4.

Если не указан параметр `--tablespace-mapping`, в простом формате табличные пространства будут копироваться в тот же путь, который они имеют на исходном сервере. Поэтому при наличии табличных пространств создать базовую копию в простом формате на том же сервере не удастся, так как копия будет направлена в те же каталоги, где располагаются исходные табличные пространства.

Когда применяется формат tar, пользователь должен позаботиться о том, чтобы все архивы tar были распакованы до запуска сервера Postgres Pro, который будет работать с этими данными. Если имеются дополнительные табличные пространства, архивы tar для них должны быть распакованы в правильные каталоги. В таком случае для этих табличных пространств сервером будут созданы символические ссылки согласно содержимому файла `tablespace_map`, включённого в архив `base.tar`.

pg_basebackup совместим с серверами той же или более младших версий, но не ниже версии 9.1. Однако режим трансляции WAL (`-X stream`) работает только с серверами версии 9.3 и новее, формат tar (`--format=tar`) поддерживается только с версиями не ниже 9.5, а инкрементальное резервное копирование (`--incremental`) поддерживается только с версиями 17 и новее.

pg_basebackup сохранит разрешения для группы, установленные для файлов данных, если такие разрешения были включены в исходном кластере.

Примеры

Создание резервной копии сервера `mydbserver` и сохранение её в локальном каталоге `/usr/local/pgsql/data`:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

Создание резервной копии локального сервера в отдельных сжатых файлах tar для каждого табличного пространства и сохранение их в каталоге `backup` с индикатором прогресса в процессе выполнения:

```
$ pg_basebackup -D backup -Ft -z -P
```

Создание резервной копии локальной базы данных с одним табличным пространством и сжатие её с помощью bzip2:

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

(Эта команда прервётся с ошибкой, если в базе данных будет несколько табличных пространств.)

Создание резервной копии локальной базы данных с перемещением табличного пространства `/opt/ts` в `./backup/ts`:

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

Создание резервной копии локального сервера в отдельных сжатых файлах tar для каждого табличного пространства (при этом используется метод сжатия gzip и уровень сжатия 9) и сохранение их в каталоге `backup`:

```
$ pg_basebackup -D backup -Ft --compress=gzip:9
```

См. также

pg_dump, Подраздел 26.4.6

pgbench

pgbench — запустить тест производительности Postgres Pro

Синтаксис

`pgbench -i` [*параметр*...] [*dbname*]

`pgbench` [*параметр*...] [*dbname*]

Описание

pgbench — это простая программа для запуска тестов производительности Postgres Pro. Она многократно выполняет одну последовательность команд, возможно в параллельных сеансах базы данных, а затем вычисляет среднюю скорость транзакций (число транзакций в секунду). По умолчанию pgbench тестирует сценарий, примерно соответствующий TPC-B, который состоит из пяти команд `SELECT`, `UPDATE` и `INSERT` в одной транзакции. Однако вы можете легко протестировать и другие сценарии, написав собственные скрипты транзакций.

Типичный вывод pgbench выглядит так:

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
number of failed transactions: 0 (0.000%)
latency average = 11.013 ms
latency stddev = 7.351 ms
initial connection time = 45.758 ms
tps = 896.967014 (without initial connection time)
```

В первых семи строках выводятся значения некоторых самых важных параметров. В шестой строке выводится максимальное число повторов транзакций с ошибками сериализации или взаимоблокировки (за подробностями обратитесь к Повторы и отказы из-за ошибок сериализации/взаимоблокировки). В восьмой строке показывается количество выполненных и запланированных транзакций (произведение числа клиентов и числа транзакций для одного клиента); эти количества будут различаться, только если выполнение завершится досрочно или какие-либо команды SQL завершатся ошибкой. (В режиме `-T` выводится только число фактически выполненных транзакций.) В следующей строке выводится количество транзакций, не выполненных из-за ошибок сериализации или взаимоблокировок (за подробностями обратитесь к Повторы и отказы из-за ошибок сериализации/взаимоблокировки). В последней строке показывается число транзакций в секунду.

Для запускаемого по умолчанию теста типа TPC-B требуется предварительно подготовить определённые таблицы. Чтобы создать и наполнить эти таблицы, следует запустить pgbench с ключом `-i` (инициализировать). (Если вы применяете нестандартный скрипт, это не требуется, но тем не менее нужно подготовить конфигурацию, нужную вашему тесту.) Запуск инициализации выглядит так:

```
pgbench -i [ другие-параметры ] имя_базы
```

где *имя_базы* — имя уже существующей базы, в которой будет проводиться тест. (Чтобы указать, как подключиться к серверу баз данных, вы также можете добавить параметры `-h`, `-p` и/или `-U`.)

Внимание

`pgbench -i` создаёт четыре таблицы `pgbench_accounts`, `pgbench_branches`, `pgbench_history` и `pgbench_tellers`, предварительно уничтожая существующие таблицы с этими именами.

> Если вы вдруг используете эти имена в своей базе данных, обязательно переключитесь на другую базу!

С «коэффициентом масштаба», по умолчанию равным 1, эти таблицы изначально содержат такое количество строк:

```
table                   # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

Эти числа можно (и в большинстве случаев даже нужно) увеличить, воспользовавшись параметром `-s` (коэффициент масштаба). При этом также может быть полезен ключ `-F` (фактор заполнения).

Подготовив требуемую конфигурацию, можно запустить тест производительности командой без `-i`, то есть:

```
pgbench [ параметры ] имя_базы
```

Практически во всех случаях, чтобы получить полезные результаты, необходимо передать какие-либо дополнительные параметры. Наиболее важные параметры: `-c` (число клиентов), `-t` (число транзакций), `-T` (длительность) и `-f` (файл со скриптом). Полный список параметров приведён ниже.

Параметры

Следующий список разделён на три подраздела: одни параметры используются при инициализации базы данных, другие при проведении тестирования, а третьи в обоих случаях.

Параметры инициализации

pgbench принимает следующие аргументы командной строки для инициализации:

`[-d]` *`dbname`*
`[--dbname=]`*`dbname`*

Указывает имя базы, в которой будет проводиться тест. Если имя не задано, то используется значение переменной окружения `PGDATABASE`. Если и переменная не задана, то в качестве имени базы будет взято имя пользователя, под которым осуществляется подключение.

`-i`
`--initialize`

Требуется для вызова режима инициализации.

`-I` *`этапы_инициализации`*
`--init-steps=`*`этапы_инициализации`*

Выполнять только выбранные из всех обычных подготовительных этапов. В параметре *`этапы_инициализации`* отдельные символы для каждого этапа выбирают, какие этапы должны выполняться. Все этапы выполняются в определённом порядке. Список этапов по умолчанию: `dtgvp`. Полный перечень подготовительных этапов:

`d` (Drop, удалить)

Удалить все существующие таблицы pgbench.

`t` (create Tables, создать таблицы)

Создать таблицы, используемые стандартным сценарием pgbench, а именно: `pgbench_accounts`, `pgbench_branches`, `pgbench_history` и `pgbench_tellers`.

`g` или `G` (Generate data, сгенерировать данные на стороне клиента или на стороне сервера)

Сгенерировать данные и загрузить их в стандартные таблицы, заменив все уже существующие данные.

С ключом `g` (выбирающим генерирование данных на стороне клиента), данные формируются в клиентском коде `pgbench`, а затем передаются на сервер. При этом соединение клиент/сервер нагружается командой `COPY`. С Postgres Pro версии 14 или выше `pgbench` использует параметр `FREEZE` для ускорения последующей операции `VACUUM`, за исключением таблицы `pgbench_accounts`, если применяется секционирование. С использованием ключа `g` после каждых 100 000 строк выдаётся сообщение о прогрессе генерации данных для всех таблиц.

С ключом `G` (выбирающим генерирование данных на стороне сервера), клиентский код `pgbench` передаёт на сервер только небольшие запросы, а собственно формированием данных занимается сервер. В этом случае сетевое соединение не нагружается, но возрастает нагрузка на сервер. При генерировании данных с ключом `G` никакие сообщения о ходе операции не выдаются.

По умолчанию при инициализации базы данные генерируются на стороне клиента (то есть подразумевается ключ `g`).

`v` (Vacuum, очистка)

Вызывать `VACUUM` для стандартных таблиц.

`p` (create Primary keys, создать первичные ключи)

Создать первичные ключи в стандартных таблицах.

`f` (create Foreign keys, создать внешние ключи)

Создать ограничения внешних ключей между стандартными таблицами. (Заметьте, что это действие по умолчанию не выполняется.)

`-F` *`фактор_заполнения`*
`--fillfactor=`*`фактор_заполнения`*

Создать таблицы `pgbench_accounts`, `pgbench_tellers` и `pgbench_branches` с заданным фактором заполнения. Значение по умолчанию — 100.

`-n`
`--no-vacuum`

Не выполнять очистку во время инициализации. (Этот параметр выключает этап инициализации `v`, даже если он был указан в `-I`.)

`-q`
`--quiet`

Переключить вывод в немногословный режим, когда выводится только одно сообщение о прогрессе в 5 секунд. В режиме по умолчанию одно сообщение выводится на каждые 100000 строк, при этом за секунду обычно выводится довольно много строк (особенно на хорошем оборудовании).

Этот параметр не оказывает влияния, если в `-I` выбран вариант `G`.

`-s` *`коэффициент_масштаба`*
`--scale=`*`коэффициент_масштаба`*

Умножить число генерируемых строк на заданный коэффициент. Например, с ключом `-s 100` в таблицу `pgbench_accounts` будут записаны 10 000 000 строк. Значение по умолчанию — 1. При коэффициенте, равном 20 000 или больше, столбцы, содержащие идентификаторы счетов (столбцы `aid`), перейдут к большим целым числам (типу `bigint`), чтобы в них могли уместиться все возможные значения идентификаторов.

`--foreign-keys`

Создать ограничения внешних ключей между стандартными таблицами. (Этот ключ добавляет этап `f` к последовательности подготовительных этапов, если он отсутствует.)

`--index-tablespace=`*`табл_пространство_индексов`*

Создать индексы в указанном табличном пространстве, а не в пространстве по умолчанию.

`--partition-method=`*`ИМЯ`*

Создать секционированную таблицу `pgbench_accounts`, применив метод *`ИМЯ`* (это может быть `range` или `hash`). Для использования этого параметра необходимо, чтобы было задано ненулевое значение `--partitions`. Если этот параметр не указывается, подразумевается метод `range`.

`--partitions=`*`ЧИСЛО`*

Создать секционированную таблицу `pgbench_accounts` с заданным *`ЧИСЛОМ`* секций примерно равного размера в соответствии с масштабированным количеством счетов. По умолчанию подразумевается число `0`, то есть таблица не секционируется.

`--tablespace=`*`табличное_пространство`*

Создать таблицы в указанном табличном пространстве, а не в пространстве по умолчанию.

`--unlogged-tables`

Создать все таблицы как нежурналируемые, а не как постоянные таблицы.

Параметры тестирования производительности

pgbench принимает следующие аргументы командной строки для тестирования производительности:

`-b` *`имя_скрипта[@вес]`*
`--builtin=`*`имя_скрипта[@вес]`*

Добавляет в список скриптов, которые будут выполняться, указанный встроенный скрипт. В число встроенных скриптов входят `tpcb-like`, `simple-update` и `select-only`. Также принимаются однозначные начала их имён. Со специальным именем `list` программа выводит список встроенных скриптов и немедленно завершается.

Дополнительно можно задать целочисленный вес после `@`, меняющий вероятность выбора этого скрипта относительно других. По умолчанию вес считается равным 1. Подробности следуют ниже.

`-c` *`клиенты`*
`--client=`*`клиенты`*

Число имитируемых клиентов, то есть число одновременных сеансов базы данных. Значение по умолчанию — 1.

`-C`
`--connect`

Устанавливать новое подключение для каждой транзакции вместо одного для каждого клиента. Это полезно для оценивания издержек подключений.

`-D` *`имя_переменной=значение`*
`--define=`*`имя_переменной=значение`*

Определить переменную для пользовательского скрипта (см. ниже). Параметр `-D` может добавляться неоднократно.

`-f` *`имя_файла[@вес]`*
`--file=`*`имя_файла[@вес]`*

Добавить в список выполняемых скриптов скрипт транзакции из файла *`имя_файла`*.

Дополнительно можно задать целочисленный вес после `@`, меняющий вероятность выбора этого скрипта относительно других. По умолчанию вес считается равным 1. (Если вам нужно передать имя скрипта, содержащее символ `@`, добавьте к такому имени вес, чтобы исключить неоднозначность прочтения, например `filen@me@1`.) Подробности следуют ниже.

`-j` *`потоки`*
`--jobs=`*`потоки`*

Число рабочих потоков в pgbench. Использовать нескольких потоков может быть полезно на многопроцессорных компьютерах. Клиенты распределяются по доступным потокам равномерно, насколько это возможно. Значение по умолчанию — 1.

`-l`
`--log`

Записать информацию о каждой транзакции в файл протокола. Подробности описаны ниже.

`-L` *`предел`*
`--latency-limit=`*`предел`*

Транзакции, продолжающиеся дольше указанного *`предела`* (в миллисекундах), подсчитываются и отмечаются отдельно, как *опаздывающие*.

В режиме ограничения скорости (`--rate=...`) транзакции, которые отстают от графика более чем на заданный *`предел`* (в мс) и поэтому никак не могут уложиться в отведённый интервал, не передаются серверу вовсе. Они подсчитываются и отмечаются отдельно как *пропущенные*.

Когда используется параметр `--max-tries`, транзакция, прерванная из-за аномалии сериализации или взаимоблокировки, не будет повторяться, если общее время всех её повторений превышает *`предел`* в миллисекундах. Чтобы ограничить только общее время повторений, а не их количество, установите значение `--max-tries=0`. По умолчанию параметр `--max-tries` имеет значение 1, и транзакции с ошибками сериализации/взаимоблокировки не повторяются. Подробнее о повторных попытках выполнения таких транзакций рассказывается в Повторы и отказы из-за ошибок сериализации/взаимоблокировки.

`-M` *`режим_запросов`*
`--protocol=`*`режим_запросов`*

Протокол, выбираемый для передачи запросов на сервер:

- `simple`: использовать простой протокол запросов.
- `extended`: использовать расширенный протокол запросов.
- `prepared`: использовать расширенный протокол запросов с подготовленными операторами.

В режиме `prepared` pgbench повторно использует результат разбора запроса, начиная со второй итерации, и поэтому работает быстрее, чем в других режимах.

По умолчанию выбирается простой протокол запросов. (За подробностями обратитесь к Главе 53.)

`-n`
`--no-vacuum`

Не производить очистку таблиц перед запуском теста. Этот параметр *необходим*, если вы применяете собственный сценарий, не затрагивающий стандартные таблицы `pgbench_accounts`, `pgbench_branches`, `pgbench_history` и `pgbench_tellers`.

`-N`
`--skip-some-updates`

Запустить встроенный упрощённый скрипт simple-update. Краткий вариант записи `-b simple-update`.

`-P` *`сек`*
`--progress=`*`сек`*

Выводить отчёт о прогрессе через заданное число секунд (*`сек`*). Выдаваемый отчёт включает время, прошедшее с момента запуска, скорость (в TPS) с момента предыдущего отчёта, а также среднее время ожидания транзакций, стандартное отклонение и количество неуспешных транзакций с момента последнего отчёта. В режиме ограничения скорости (`-R`) время ожидания вычисляется относительно назначенного времени запуска транзакции, а не фактического времени её начала, так что оно включает и среднее время отставания от графика. Когда параметр `--max-tries` включает повторение транзакций после ошибок сериализации/взаимоблокировок, в отчёт добавляется количество повторявшихся транзакций и общее число повторов.

`-r`
`--report-per-command`

Выдать следующую статистику по каждой команде после завершения теста: среднюю длительность выполнения операторов (время выполнения с точки зрения клиента), число отказов и повторений вследствие ошибок сериализации и взаимоблокировки в этой команде. Статистика повторений отображается в отчёте, только если параметр `--max-tries` не равен 1.

`-R` *`скорость передачи`*
`--rate=`*`скорость передачи`*

Выполнять транзакции, ориентируясь на заданную скорость, а не максимально быстро (по умолчанию). Скорость задаётся в транзакциях в секунду. Если заданная скорость превышает максимально возможную, это ограничение скорости не повлияет на результаты.

Для получения нужной скорости транзакции запускаются со случайными задержками, имеющими распределение Пуассона. При этом запланированное время запуска отсчитывается от начального времени, а не от завершения предыдущей транзакции. Это означает, что если какие-то транзакции отстанут от изначально рассчитанного времени завершения, всё же возможно, что последующие нагонят график.

В режиме ограничения скорости время ожидания транзакций, выводимое по итогам тестирования, вычисляется, исходя из запланированного времени запуска, так что в него входит время, которое очередная транзакция должна была ждать завершения предыдущей транзакции. Это время называется временем отклонения от графика, и его среднее и максимальное значения выводятся отдельно. Время ожидания транзакций с момента их фактического запуска, то есть время, потраченное на выполнение транзакций в базе данных, можно получить, если вычесть время отклонения от графика из времени ожидания транзакций.

Если ограничение `--latency-limit` задаётся вместе с `--rate`, транзакция может заведомо не вписываться в отведённое ей время, если предыдущая транзакция завершится слишком поздно, так как ожидаемое время окончания транзакции отсчитывается от времени запуска по графику. Такие транзакции не передаются серверу, а пропускаются и подсчитываются отдельно.

Большое значение отклонения от графика свидетельствует о том, что система не успевает выполнять транзакции с заданной скоростью и выбранным числом клиентов и потоков. Когда среднее время ожидания транзакции превышает запланированный интервал между транзакциями, каждая последующая транзакция будет отставать от графика, и чем дольше будет выполняться тестирование, тем больше будет отставание. Когда это наблюдается, нужно уменьшить скорость транзакций.

`-s` *`коэффициент_масштаба`*
`--scale=`*`коэффициент_масштаба`*

Показать заданный коэффициент масштаба в выводе pgbench. Для встроенных тестов это не требуется; корректный коэффициент масштаба будет получен в результате подсчёта строк в таблице `pgbench_branches`. Однако при использовании только нестандартных тестов (запускаемых с ключом `-f`) без этого параметра в качестве коэффициента масштаба будет выводиться 1.

`-S`
`--select-only`

Запустить встроенный скрипт select-only (только выборка). Краткий вариант записи `-b select-only`.

`-t` *`транзакции`*
`--transactions=`*`транзакции`*

Число транзакций, которые будут выполняться каждым клиентом (по умолчанию 10).

`-T` *`секунды`*
`--time=`*`секунды`*

Выполнять тест с ограничением по времени (в секундах), а не по числу транзакций для каждого клиента. Параметры `-t` и `-T` являются взаимоисключающими.

`-v`
`--vacuum-all`

Очищать все четыре стандартные таблицы перед запуском теста. Без параметров `-n` и `-v` pgbench будет очищать от старых записей таблицы `pgbench_tellers` и `pgbench_branches`, а также опустошать `pgbench_history`.

`--aggregate-interval=`*`секунды`*

Длительность интервала агрегации (в секундах). Может использоваться только с ключом `-l`. С данным параметром в протокол выводится сводка по интервалам, как описано ниже.

`--exit-on-abort`

Немедленно завершить работу, если клиент отключается из-за ошибки. Без этого параметра, даже если один клиент отключается, другие клиенты продолжат работать в соответствии с параметром `-t` или `-T`, а pgbench выведет неполные результаты.

Обратите внимание, что ошибки сериализации или взаимоблокировки не прерывают работу клиента. За дополнительными сведениями обратитесь к Повторы и отказы из-за ошибок сериализации/взаимоблокировки.

`--failures-detailed`

Выдавать информацию об ошибках в протоколе по транзакциям и в протоколе с агрегированием, а также в основном отчёте и в отчётах по скриптам, группируя её по типам:

- ошибки сериализации;
- ошибки взаимоблокировки;

За подробностями обратитесь к Повторы и отказы из-за ошибок сериализации/взаимоблокировки.

`--log-prefix=`*`префикс`*

Задать префикс имён файлов для файлов протоколов, создаваемых с ключом `--log`. Префикс по умолчанию — `pgbench_log`.

`--max-tries=`*`число_попыток`*

Разрешить повторение транзакций с ошибками сериализации/взаимоблокировки и установить максимальное число попыток выполнения транзакций. Этот параметр можно комбинировать с параметром `--latency-limit`, который ограничивает общее время всех попыток для транзакции; также заметьте, что нельзя выбрать неограниченное количество попыток (`--max-tries=0`), не определив `--latency-limit` или `--time`. Значение по умолчанию — 1, то есть транзакции с ошибками сериализации/взаимоблокировки повторяться не будут. Подробнее о повторении таких транзакций рассказывается в Повторы и отказы из-за ошибок сериализации/взаимоблокировки.

`--progress-timestamp`

При отображении прогресса (с параметром `-P`) выводить текущее время (в формате Unix), а не количество секунд от начала запуска. Время задаётся в секундах с точностью до миллисекунд. Это помогает сравнивать журналы, записываемые разными средствами.

`--random-seed=`*`затравка`*

Установить затравку для генератора случайных чисел. Инициализирует генератор случайных чисел, который затем выдаёт последовательность начальных состояний отдельных генераторов для каждого потока. *`затравка`* может принимать следующие значения: `time` (по умолчанию, затравка базируется на текущем времени), `rand` (задействовать надёжный генератор случайных чисел или выдать ошибку, если он отсутствует) или беззнаковое десятичное число. Генератор случайных чисел может вызываться явно из скрипта pgbench (функциями `random...`) или неявно (например, для планирования выполнения транзакций с ключом `--rate`). В случае установки значения явным образом оно выводится в терминале. Любое значение, допустимое в качестве *`затравки`*, можно также задать в переменной окружения `PGBENCH_RANDOM_SEED`. Чтобы заданная затравка применялась во всех возможных случаях использования, задайте этот параметр первым или установите переменную окружения.

Явное указание определённой затравки позволяет точно воспроизвести выполнение `pgbench` в части использования случайных чисел. Так как случайное состояние поддерживается внутри потока, это означает, что выполнение `pgbench` при одинаковых запусках повторится в точности, если один поток используется одним клиентом и отсутствуют внешние зависимости или зависимости от данных. Со статистической точки зрения точное воспроизведение выполнения нежелательно, так как это может скрыть вариативность производительности или показать завышенную скорость, например из-за попадания в одни и те же страницы данных. Однако это может быть очень полезно для отладки, например, для повторения редкого сценария, приводящего к ошибке. Используйте данную возможность обдуманно.

`--sampling-rate=`*`скорость передачи`*

Частота выборки для записи данных в протокол, изменяя которую можно уменьшить объём протокола. При указании этого параметра в протокол выводится информация только о заданном проценте транзакций. Со значением 1.0 в нём будут отмечаться все транзакции, а с 0.05 только 5%.

Обрабатывая протокол, не забудьте учесть частоту выборки. Например, вычисляя скорость (TPS), вам нужно будет соответственно умножить содержащиеся в нём числа (например, с частотой выборки 0.01 вы получите только 1/100 фактической скорости).

`--show-script=`*`имя_скрипта`*

Вывести код встроенного скрипта *`имя_скрипта`* в stderr и сразу завершиться.

`--verbose-errors`

Выводить сообщения обо всех ошибках сериализации/взаимоблокировки и отказах (ошибках, после которых транзакция не повторяется) с информацией о том, как ограничиваются повторения и насколько достигается ограничение. (Учтите, что в этом случае объём вывода может значительно увеличиться.) За подробностями обратитесь к Повторы и отказы из-за ошибок сериализации/взаимоблокировки.

Общие параметры

Программа pgbench также принимает следующие общие аргументы командной строки, определяющие параметры подключения и прочие общие параметры:

`--debug`

Выводить отладочные сообщения.

`-h` *`компьютер`*
`--host=`*`компьютер`*

Адрес сервера баз данных

`-p` *`порт`*
`--port=`*`порт`*

Номер порта сервера баз данных

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя для подключения

`-V`
`--version`

Вывести версию pgbench и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки pgbench и завершиться.

Код завершения

В случае успешного выполнения возвращается код 0. Код завершения 1 указывает на статичные проблемы, например ошибки в параметрах командной строки или непредвиденные внутренние ошибки. В случае ошибок, возникающих на ранних этапах при запуске теста, например при сбое начального подключения, кодом завершения также будет 1. При возникновении ошибок во время выполнения, например при обращении к базе данных или выполнении скрипта, выдаётся код завершения 2. В последнем случае pgbench выведет частичные результаты, если не указан параметр `--exit-on-abort`.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию.

Как и большинство других утилит Postgres Pro, приложение также использует переменные окружения, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Каково содержание «транзакции», которую выполняет pgbench?

Программа pgbench выполняет тестовые скрипты, выбирая их случайным образом из заданного списка. Это могут быть как встроенные скрипты, задаваемые аргументами `-b`, так и пользовательские, задаваемые аргументами `-f`. Для каждого скрипта можно задать относительный вес после `@`, чтобы скорректировать вероятность его выбора. По умолчанию вес считается равным `1`. Скрипты с весом `0` игнорируются.

Стандартный встроенный скрипт (также вызываемый с ключом `-b tpcb-like`) выдаёт семь команд в транзакции со случайно выбранными `aid`, `tid`, `bid` и `delta`. Его сценарий написан по мотивам теста производительности TPC-B, но это не собственно TPC-B, потому он называется так.

1. `BEGIN;`
2. `UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;`
3. `SELECT abalance FROM pgbench_accounts WHERE aid = :aid;`
4. `UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;`
5. `UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;`
6. `INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);`
7. `END;`

При выборе встроенного скрипта `simple-update` (или указании `-N`) шаги 4 и 5 исключаются из транзакции. Это позволяет избежать конкуренции при обращении к этим таблицам, но тест становится ещё менее похожим на TPC-B.

При выборе встроенного теста `select-only` (или указании `-S`) выполняется только `SELECT`.

Пользовательские скрипты

Программа pgbench поддерживает запуск пользовательских сценариев оценки производительности, позволяя заменять стандартный скрипт транзакции (описанный выше) скриптом, считываемым из файла (с параметром `-f`). В этом случае «транзакцией» считается одно выполнение данного скрипта.

Файл скрипта содержит одну или несколько команд SQL, разделённых точкой с запятой. Пустые строки и строки, начинающиеся с `--`, игнорируются. В файлах скриптов также могут содержаться «метакоманды», которые обрабатывает сама программа pgbench, как описано ниже.

> ### Примечание
>
> До версии Postgres Pro 9.6, SQL-команды в файлах скриптов завершались символами перевода строки, и поэтому они не могли занимать несколько строк. Теперь для разделения последовательных команд SQL *требуется* добавлять точку с запятой (хотя без неё можно обойтись в конце SQL-команды, за которой идёт метакоманда). Если вам нужно создать файл скрипта, работающий и со старыми версиями pgbench, записывайте каждую команду SQL в отдельной строке и завершайте её точкой с запятой.
>
> Предполагается, что скрипты pgbench не содержат незавершённых блоков SQL-транзакций. Если по достижении клиентом конца скрипта окажется, что блок последней транзакции не завершён, работа клиента будет прервана.

Для файлов скриптов реализован простой механизм подстановки переменных. Имя переменных должно состоять из букв (буквы могут быть не латинскими), подчёркиваний и цифр (но цифра не может быть первым символом). Переменные можно задать в командной строке параметрами `-D`, описанными выше, или метакомандами, рассматриваемыми ниже. Помимо переменных, которые можно установить параметрами командной строки `-D`, есть несколько автоматически устанавливаемых переменных; они перечислены в Таблице 302. Если значение этих переменных задаётся в параметре `-D`, оно переопределяет автоматическое значение. Когда значение переменной определено, его можно вставить в команду SQL, написав `:`*`имя_переменной`*. Каждый клиентский сеанс, если их несколько, получает собственный набор переменных. В одном операторе pgbench поддерживает до 255 ссылок на переменные.

Таблица 302. Автоматические переменные pgbench

Переменная	Описание
`client_id`	уникальное число, идентифицирующее клиентский сеанс (начиная с нуля)

Переменная	**Описание**
`default_seed`	затравка, по умолчанию используемая в функциях, вычисляющих хеш и псевдослучайные перестановки
`random_seed`	затравка генератора случайных чисел (в отсутствие переопределения с ключом `-D`)
`scale`	текущий коэффициент масштаба

Метакоманды в скрипте начинаются с обратной косой черты (`\`) и обычно продолжаются до конца строки, хотя их можно переносить на следующую строку последовательностью символов: обратная косая, возврат каретки. Аргументы метакоманд разделяются пробелами. Поддерживаемые метакоманды представлены ниже:

`\gset [`*`префикс`*`] \aset [`*`префикс`*`]`

Эти команды могут применяться для завершения SQL-запросов вместо завершающей точки с запятой (`;`).

Когда используется команда `\gset`, ожидается, что предыдущий SQL-запрос возвратит одну строку; значения столбцов будут сохранены в переменные с именами столбцов, а если указан *`префикс`*, он будет добавлен в эти имена.

Когда используется команда `\aset`, значения столбцов всех совмещённых SQL-запросов (разделённых `\;`) будут сохранены в переменные, названные по именам столбцов с добавлением *`префикса`*, если он задан. Если запрос не возвращает в результате строки, присваивание не выполняется и в этом можно убедиться, проверив, существует ли переменная. Если запрос возвращает несколько строк, в переменных сохраняется последнее значение.

Команды `\gset` и `\aset` нельзя использовать в конвейерном режиме, так как ко времени, когда результаты запросов могут понадобиться команде, они ещё не будут готовы.

В следующем примере итоговый баланс счёта из первого запроса попадает в переменную *`abalance`*, а целочисленные значения из третьего запроса попадают в переменные *`p_two`* и *`p_three`*. Результат второго запроса отбрасывается. Результаты двух последних запросов объединяются и сохраняются в переменных *`four`* и *`five`*.

```
UPDATE pgbench_accounts
  SET abalance = abalance + :delta
  WHERE aid = :aid
  RETURNING abalance \gset
-- объединяет два запроса
SELECT 1 \;
SELECT 2 AS two, 3 AS three \gset p_
SELECT 4 AS four \; SELECT 5 AS five \aset
```

`\if` *`выражение`*
`\elif` *`выражение`*
`\else`
`\endif`

Эта группа команд реализует вкладываемые условные блоки, подобные `\if` *`выражение`* в `psql`. В качестве условных задаются те же выражения, что и в `\set`, при этом истинным считается любое ненулевое значение.

`\set` *`имя_переменной выражение`*

Устанавливает для переменной *`имя_переменной`* значение, вычисленное из *`выражения`*. Выражение может содержать константу `NULL`, логические константы `TRUE` и `FALSE`, целочисленные константы (например, `5432`), константы с плавающей точкой (например, `3.14159`), ссылки на переменные `:`*`имя_переменной`*, операторы с обычными для SQL приоритетами и ассоциативностью, вызовы функций, общие условные SQL-выражения `CASE`, а также скобки.

Функции и большинство операторов возвращают `NULL` для аргументов `NULL`.

При проверке условия отличные от нуля числовые значения воспринимаются как `TRUE`, а числовые нулевые значения и `NULL` — как `FALSE`.

При переполнениях, вызванных слишком большими числами с плавающей точкой или целыми, а также целочисленными операциями (`+`, `-`, `*` и `/`), выдаются ошибки.

Если в конструкции `CASE` отсутствует заключительное `ELSE`, значением по умолчанию считается `NULL`.

Примеры:

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
           (100000 * :scale) + 1
\set divx CASE WHEN :x <> 0 THEN :y/:x ELSE NULL END
```

`\sleep` *`номер`* `[ us | ms | s ]`

Приостанавливает выполнение скрипта на заданное число микросекунд (`us`), миллисекунд (`ms`) или секунд (`s`). Когда единицы не указываются, подразумеваются секунды. Здесь *`число`* может быть целочисленной константой или ссылкой `:`*`имя_переменной`* на переменную с целочисленным значением.

Пример:

```
\sleep 10 ms
```

`\setshell` *`имя_переменной команда`* `[` *`аргумент`* `... ]`

Присваивает переменной *`имя_переменной`* результат команды оболочки *`команда`* с указанными *`аргументами`*. Эта команда должна просто выдать целочисленное значение в стандартный вывод.

Здесь *`команда`* и каждый *`аргумент`* может быть либо текстовой константой, либо ссылкой на переменную `:`*`имя_переменной`*. Если вы хотите записать *`аргумент`*, начинающийся с двоеточия, добавьте перед *`аргументом`* дополнительное двоеточие.

Пример:

```
\setshell назначаемая_переменная команда
 строковый_аргумент :переменная ::строка_начинающаяся_двоеточием
```

`\shell` *`команда`* `[` *`аргумент`* `... ]`

Действует так же, как и `\setshell`, но не учитывает результат команды.

Пример:

```
\shell команда строковый_аргумент :переменная ::строка_начинающаяся_двоеточием
```

`\startpipeline`
`\syncpipeline`
`\endpipeline`

Эта группа команд осуществляет конвейеризацию SQL-операторов. Конвейер должен начинаться с команды `\startpipeline` и заканчиваться командой `\endpipeline`. В промежутке между ними может быть любое количество команд `\syncpipeline`, которые посылают сообщение синхронизации, не прерывая текущий конвейер и не очищая буфер отправки. В конвейерном режиме команды передаются серверу, не дожидаясь результатов предыдущих команд. За подробностями обратитесь к Разделу 32.5. Для конвейерного режима должен использоваться расширенный протокол запросов.

Встроенные операторы

Перечисленные в Таблице 303 арифметические, битовые и логические операторы, а также операторы сравнения встроены в pgbench и могут применяться в выражениях в `\set`. Эти операторы приведены в порядке возрастания их приоритета. Не считая явно отмеченных исключений, операторы с двумя числовыми аргументами будут выдавать результат в типе с плавающей точкой, если какой-либо аргумент имеет такой тип; в противном случае результат будет целочисленным.

Таблица 303. Операторы pgbench

Оператор Описание Примеры
boolean `OR` *boolean* → *boolean* Логическое ИЛИ `5 or 0` → `TRUE`
boolean `AND` *boolean* → *boolean* Логическое И `3 and 0` → `FALSE`
`NOT` *boolean* → *boolean* Логическое НЕ `not false` → `TRUE`
boolean `IS [NOT] (NULL|TRUE|FALSE)` → *boolean* Логические проверки значений `1 is null` → `FALSE`
значение `ISNULL|NOTNULL` → *boolean* Проверки на NULL `1 notnull` → `TRUE`
номер `=` *номер* → *boolean* Равно `5 = 4` → `FALSE`
номер `<>` *номер* → *boolean* Не равно `5 <> 4` → `TRUE`
номер `!=` *номер* → *boolean* Не равно `5 != 5` → `FALSE`
номер `<` *номер* → *boolean* Меньше `5 < 4` → `FALSE`
номер `<=` *номер* → *boolean* Меньше или равно `5 <= 4` → `FALSE`
номер `>` *номер* → *boolean* Больше `5 > 4` → `TRUE`

Оператор **Описание** **Примеры**
номер `>=` *номер* → *boolean* Больше или равно `5 >= 4` → `TRUE`
integer `|` *integer* → *integer* Битовое ИЛИ `1 | 2` → `3`
integer `#` *integer* → *integer* Битовое исключающее ИЛИ `1 # 3` → `2`
integer `&` *integer* → *integer* Битовое И `1 & 3` → `1`
`~` *integer* → *integer* Битовое НЕ `~ 1` → `-2`
integer `<<` *integer* → *integer* Битовый сдвиг влево `1 << 2` → `4`
integer `>>` *integer* → *integer* Битовый сдвиг вправо `8 >> 2` → `2`
номер `+` *номер* → *номер* Сложение `5 + 4` → `9`
номер `-` *номер* → *номер* Вычитание `3 - 2.0` → `1.0`
номер `*` *номер* → *номер* Умножение `5 * 4` → `20`
номер `/` *номер* → *номер* Деление (если оба аргумента целочисленные, результат округляется в сторону нуля) `5 / 3` → `1`
integer `%` *integer* → *integer* Остаток от деления `3 % 2` → `1`
`-` *номер* → *номер* Смена знака `- 2.0` → `-2.0`

Встроенные функции

Функции, перечисленные в Таблице 304, встроены в pgbench и могут применяться в выражениях в метакоманде `\set`.

Таблица 304. Функции pgbench

Функция **Описание** **Примеры**
`abs` (*число*) → тип аргумента Модуль числа (абсолютное значение) `abs(-17)` → `17`
`debug` (*число*) → тип аргумента Выводит аргумент в stderr и выдаёт его. `debug(5432.1)` → `5432.1`

`double` (*число*) → `double`
Приводит аргумент к типу с плавающей точкой.
`double(5432)` → `5432.0`

`exp` (*число*) → `double`
Экспонента (`e` возводится в заданную степень)
`exp(1.0)` → `2.718281828459045`

`greatest` (*число* [, `...`]) → `double` при наличии аргумента(ов) double, иначе — `integer`
Выбирает наибольшее значение среди аргументов.
`greatest(5, 4, 3, 2)` → `5`

`hash` (*значение* [, *затравка*]) → `integer`
Псевдоним `hash_murmur2` .
`hash(10, 5432)` → `-5817877081768721676`

`hash_fnv1a` (*значение* [, *затравка*]) → `integer`
Вычисляет хеш по алгоритму *FNV-1a*.
`hash_fnv1a(10, 5432)` → `-7793829335365542153`

`hash_murmur2` (*значение* [, *затравка*]) → `integer`
Вычисляет хеш по алгоритму *MurmurHash2*.
`hash_murmur2(10, 5432)` → `-5817877081768721676`

`int` (*число*) → `integer`
Приводит аргумент к целочисленному типу.
`int(5.4 + 3.8)` → `9`

`least` (*число* [, `...`]) → `double` при наличии аргумента(ов) double, иначе — `integer`
Выбирает наименьшее значение среди аргументов.
`least(5, 4, 3, 2.1)` → `2.1`

`ln` (*число*) → `double`
Натуральный логарифм
`ln(2.718281828459045)` → `1.0`

`mod` (*целое*, *целое*) → `integer`
Остаток от деления
`mod(54, 32)` → `22`

`permute` (*i*, *size* [, *seed*]) → `integer`
Переставленное значение *i* в диапазоне `[0, size)` , то есть позиция *i* (по модулю *size*) в псевдослучайной перестановке целых чисел `0...size-1`, полученной с затравкой *seed*, см. ниже.
`permute(0, 4)` → `целое между 0 и 3`

Функция **Описание** **Примеры**

`pi` () → `double`
Приближённое значение π
`pi()` → `3.14159265358979323846`

`pow` (*x*, *y*) → `double`
`power` (*x*, *y*) → `double`
Возводит *x* в степень *y*
`pow(2.0, 10)` → `1024.0`

`random` (*lb*, *ub*) → `integer`
Выдаёт случайное целое число с равномерным распределением в интервале `[lb, ub]` .
`random(1, 10)` → `целое между 1 и 10`

`random_exponential` (*lb*, *ub*, *parameter*) → `integer`
Выдаёт случайное целое число с экспоненциальным распределением в интервале `[lb, ub]`, см. ниже.
`random_exponential(1, 10, 3.0)` → `целое между 1 и 10`

`random_gaussian` (*lb*, *ub*, *parameter*) → `integer`
Выдаёт целое число с распределением Гаусса в интервале `[lb, ub]` , см. ниже.
`random_gaussian(1, 10, 2.5)` → `целое между 1 и 10`

`random_zipfian` (*lb*, *ub*, *parameter*) → `integer`
Выдаёт целое число с распределением Ципфа в интервале `[lb, ub]` , см. ниже.
`random_zipfian(1, 10, 1.5)` → `целое между 1 и 10`

`sqrt` (*число*) → `double`
Квадратный корень
`sqrt(2.0)` → `1.414213562`

Функция `random` выдаёт значения с равномерным распределением, то есть вероятности получения всех чисел в интервале равны. Функции `random_exponential`, `random_gaussian` и `random_zipfian` требуют указания дополнительного параметра типа double, определяющего точную форму распределения.

- Для экспоненциального распределения *`parameter`* управляет распределением, обрезая быстро спадающее экспоненциальное распределение в точке *`parameter`*, а затем это распределение проецируется на целые числа между границами. Точнее говоря, с

  f(x) = exp(-parameter * (x - min) / (max - min + 1)) / (1 - exp(-parameter))

  значение *`i`* между *`min`* и *`max`* выдаётся с вероятностью: `f(i) - f(i + 1)`.

  Интуиция подсказывает, что чем больше *`parameter`*, тем чаще будут выдаваться значения, близкие к *`min`*, и тем реже значения, близкие к *`max`*. Чем *`parameter`* ближе к 0, тем более плоским (более равномерным) будет распределение. В грубом приближении при таком распределении наиболее частый 1% значений в диапазоне рядом с *`min`* выдаётся *`parameter`*% времени. Значение *`parameter`* должно быть строго положительным.

- Для распределения Гаусса по интервалу строится обычное нормальное распределение (классическая кривая Гаусса в форме колокола) и этот интервал обрезается в точке `-parameter` слева и `+parameter` справа. Вероятнее всего при таком распределении выдаются значения из середины интервала. Точнее говоря, если `PHI(x)` — функция распределения нормальной случайной величины со средним значением `mu`, равным `(max + min) / 2.0`, и

  f(x) = PHI(2.0 * parameter * (x - mu) / (max - min + 1)) /

(2.0 * PHI(parameter) - 1)

тогда значение *`i`* между *`min`* и *`max`* включительно выдаётся с вероятностью: `f(i + 0.5) - f(i - 0.5)`. Интуиция подсказывает, что чем больше *`parameter`*, тем чаще будут выдаваться значения в середине интервала, и тем реже значения у границ *`min`* и *`max`*. Около 67% значений будут выдаваться из среднего интервала `1.0 / parameter`, то есть плюс/минус `0.5 / parameter` от среднего значения, и 95% из среднего интервала `2.0 / parameter`, то есть плюс/минус `1.0 / parameter` от среднего значения; например, если *`parameter`* равен 4.0, 67% значений выдаются из средней четверти (1.0 / 4.0) интервала (то есть от `3.0 / 8.0` до `5.0 / 8.0`) и 95% из средней половины (`2.0 / 4.0`) интервала (из второй и третьей четвертей). Значение *`parameter`* не может быть меньше 2.0.

- Функция `random_zipfian` генерирует ограниченное распределение по закону Ципфа. *`parameter`* определяет, насколько неравномерно распределение. Чем больше *`parameter`*, тем чаще выдаются значения, близкие к началу интервала. Это распределение таково, что при диапазоне, начинающемся с 1, отношение вероятности получить *k* к вероятности получения *k+1* равняется `((k+1)/k)**parameter`. Например, `random_zipfian(1, ..., 2.5)` будет выдавать число `1` примерно в `(2/1)**2.5 = 5.66` раза чаще, чем число `2`, а оно, в свою очередь, будет выдаваться примерно в `(3/2)**2.5 = 2.76` раза чаще, чем `3`, и так далее.

  Это распределение реализовано в pgbench по материалу книги «Non-Uniform Random Variate Generation» («Генерация неравномерно распределённых случайных чисел» Люк Деврой, стр. 550-551, Springer 1986. Вследствие ограничений алгоритма *`parameter`* может принимать значения только в интервале [1.001, 1000].

Примечание

Разрабатывая тест производительности, выбирающий строки неравномерно, учтите, что выбираемые строки могут коррелировать с другими данными, например, идентификаторами, выдаваемыми последовательностями, или физическим расположением строк, а это может искажать оценки производительности.

Чтобы избежать этого, можно использовать функцию `permute` для перемешивания выбранных строк и устранения таких корреляций, либо выполнить другие дополнительные операции, дающие подобный эффект.

Функции хеширования `hash`, `hash_murmur2` и `hash_fnv1a` принимают на вход хешируемое значение и необязательный параметр с затравкой. Если значение затравки не задаётся, используется значение переменной `:default_seed`, которая инициализируется случайным числом (если не задаётся явно ключом командной строки `-D`).

Функция `permute` принимает входное значение, размер и необязательный параметр с затравкой. Она получает псевдослучайную перестановку целых чисел в диапазоне `[0, размер)` и возвращает индекс входного значения в этой перестановке. Выбираемая перестановка зависит от затравки; по умолчанию затравкой будет значение `:default_seed`. В отличие от функций вычисления хеша, функция `permute` гарантирует отсутствие пропусков и наложений в выходных значениях. Входные значения, лежащие за границами интервала, пересчитываются по модулю размера. Если значение размера не положительное, эта функция выдаёт ошибку. Функцию `permute` можно использовать для выравнивания результатов функций, выдающих неравномерно распределённые случайные числа, таких как `random_zipfian` или `random_exponential`, чтобы чаще выдаваемые значения не создавали корреляции. Например, следующий скрипт pgbench эмулирует возможную реальную нагрузку, типичную для социальных медиа- и блог-платформ, где несколько пользователей генерируют львиную долю нагрузки:

```
\set size 1000000
\set r random_zipfian(1, :size, 1.07)
\set k 1 + permute(:r, :size)
```

В некоторых случаях требуются другие разнообразные распределения, не коррелирующие друг с другом, и тогда может быть полезно дополнительно указать затравку:

```
\set k1 1 + permute(:r, :size, :default_seed + 123)
\set k2 1 + permute(:r, :size, :default_seed + 321)
```

Примерно такого же эффекта можно добиться, используя функцию `hash`:

```
\set size 1000000
\set r random_zipfian(1, 100 * :size, 1.07)
\set k 1 + abs(hash(:r)) % :size
```

Но так как `hash` может порождать коллизии, некоторые значения окажутся пропущенными, а другие будут встречаться чаще других, при том что в исходном распределении этого не наблюдалось.

В качестве примера взгляните на встроенное определение транзакции типа TPC-B:

```
\set aid random(1, 100000 * :scale)
\set bid random(1, 1 * :scale)
\set tid random(1, 10 * :scale)
\set delta random(-5000, 5000)
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

С таким скриптом транзакция на каждой итерации будет обращаться к разным, случайно выбираемым строкам. (Этот пример показывает, почему важно, чтобы в каждом клиентском сеансе были собственные переменные — в противном случае они не будут независимо обращаться к разным строкам.)

Протоколирование транзакций

С параметром `-l` (но без `--aggregate-interval`), pgbench записывает информацию о каждой транзакции в протокол. Этот файл протокола будет называться *`префикс`*`.`*`nnn`*, где *`префикс`* по умолчанию — `pgbench_log`, а *`nnn`* — PID процесса pgbench. Префикс можно сменить, воспользовавшись ключом `--log-prefix`. Если параметр `-j` равен 2 или выше, будет создано несколько рабочих потоков, и каждый будет записывать отдельный протокол. Первый рабочий процесс будет использовать файл с тем же именем, что и в стандартном случае с одним потоком, а файлы остальных потоков будут называться *`префикс`*`.`*`nnn`*`.`*`mmm`*, где *`mmm`* — порядковый номер рабочего процесса, начиная с 1.

Каждая строка в файле протокола описывает одну транзакцию. Она содержит следующие поля, разделённые пробелами:

`код_клиента`

идентифицирует клиентский сеанс, в котором выполнялась транзакция

`число_транзакций`

показывает, сколько SQL-транзакций было выполнено в этом сеансе

`длительность`

общее время транзакции, в микросекундах

`номер_скрипта`

показывает, какой файл скрипта использовался (это полезно при указании нескольких скриптов ключами `-f` и `-b`)

`время_эпохи`

отметка времени в формате Unix, показывающая, когда транзакция была завершена *время_мкс*

микросекунды в отметке времени, показывающей, когда транзакция была завершена

отставание_от_графика

задержка начала транзакции, которая представляет собой разницу между запланированным временем запуска транзакции и фактическим временем запуска, в микросекундах (выводится, только когда указан параметр `--rate`)

повторные_попытки

число повторений транзакции после ошибок сериализации или взаимоблокировки (выводится, только когда `--max-tries` не равен одному)

Когда одновременно применяются параметры `--rate` и `--latency-limit`, в поле *длительность* для пропущенных транзакций выводится `skipped`. Если транзакция завершилась неудачей, в столбце *длительность* выводится `failed`. Когда применяется параметр `--failures-detailed`, вместо *длительности* для неуспешной транзакции будет указано `serialization` или `deadlock` в зависимости от типа ошибки (за дополнительными сведениями обратитесь к Повторы и отказы из-за ошибок сериализации/взаимоблокировки).

Фрагмент протокола, полученного при выполнении с одним клиентом:

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

Ещё один пример с `--rate=100` и `--latency-limit=5` (обратите внимание на дополнительный столбец *отставание_от_графика*):

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740
```

В этом примере транзакция 82 опоздала, так как её длительность (6.173 мс) превысила ограничение в 5 мс. Следующие две транзакции были пропущены, так как было слишком поздно их начинать.

Ниже показан фрагмент протокола с отказами и повторными попытками при максимальном числе попыток, равном 10 (обратите внимание на дополнительный столбец *повторные_попытки*):

```
3 0 47423 0 1499414498 34501 3
3 1 8333 0 1499414498 42848 0
3 2 8358 0 1499414498 51219 0
4 0 72345 0 1499414498 59433 6
1 3 41718 0 1499414498 67879 4
1 4 8416 0 1499414498 76311 0
3 3 33235 0 1499414498 84469 3
0 0 failed 0 1499414498 84905 9
2 0 failed 0 1499414498 86248 9
3 4 8307 0 1499414498 92788 0
```

Если применяется параметр `--failures-detailed`, в графе *длительность* выводится тип ошибки:

```
3 0 47423 0 1499414498 34501 3
3 1 8333 0 1499414498 42848 0
3 2 8358 0 1499414498 51219 0
4 0 72345 0 1499414498 59433 6
```

```
1 3 41718 0 1499414498 67879 4
1 4 8416 0 1499414498 76311 0
3 3 33235 0 1499414498 84469 3
0 0 serialization 0 1499414498 84905 9
2 0 serialization 0 1499414498 86248 9
3 4 8307 0 1499414498 92788 0
```

Когда проводится длительное тестирование с большим количеством транзакций, файлы протоколов могут быть очень объёмными. Чтобы в них записывалась только случайная выборка транзакций, можно запустить команду с параметром `--sampling-rate`.

Протоколирование с агрегированием

С параметром `--aggregate-interval` для файлов протоколов используется другой формат. Каждая строка протокола описывает один интервал агрегации. Она содержит следующие поля, разделённые пробелами:

начало_интервала

начальное время интервала в формате времени Unix

число_транзакций

количество транзакций в данном интервале

сумма_длительности

суммарная длительность транзакций

сумма_длительности_2

сумма квадратов длительностей транзакций

мин_длительность

минимальная длительность транзакции

макс_длительность

максимальная длительность транзакции

сумма_задержки

сумма задержек начала транзакций (ноль, если не указан параметр `--rate`)

сумма_задержки_2

сумма квадратов задержек начала транзакций (ноль, если не указан параметр `--rate`)

мин_задержка

минимальная задержка начала транзакции (ноль, если не указан параметр `--rate`)

макс_задержка

максимальная задержка начала транзакции (ноль, если не указан параметр `--rate`)

пропущено_транзакций

число транзакций, пропущенных из-за того, что было слишком поздно их начинать (ноль, если не заданы параметры `--rate` и `--latency-limit`)

повторено_транзакций

число транзакций, которые были повторены (ноль, если не указан параметр `--max-tries`)

повторные_попытки

число повторов после ошибок сериализации или взаимоблокировок (ноль, если параметр `--max-tries` не единица) *сбои_сериализации*

число транзакций, которые были прерваны из-за ошибок сериализации и не были повторены впоследствии (ноль, если не указан параметр `--failures-detailed`)

сбои_взаимоблокировки

число транзакций, которые были прерваны из-за ошибок взаимоблокировки и не были повторены впоследствии (ноль, если не указан параметр `--failures-detailed`)

Ниже представлен пример вывода с этим параметром:

```
pgbench --aggregate-interval=10 --time=20 --client=10 --log --rate=1000 --latency-limit=10 --failures-detailed --max-tries=10 test

1650260552 5178 26171317 177284491527 1136 44462 2647617 7321113867 0 9866 64 7564
 28340 4148 0
1650260562 4808 25573984 220121792172 1171 62083 3037380 9666800914 0 9998 598 7392
 26621 4527 0
```

Заметьте, что простой формат протокола (без агрегирования) показывает, какой скрипт использовался для каждой транзакции, в отличие от формата с агрегированием. Таким образом, если вам нужны подобные сведения, но в разрезе скриптов, вам придётся агрегировать данные самостоятельно.

Отчёт по операторам

С параметром `-r` программа pgbench собирает следующую статистику по каждому оператору:

- `latency` — время выполнения для каждого оператора. pgbench выводит среднее по всем успешным попыткам выполнения оператора.
- Число отказов для данного оператора. За дополнительной информацией обратитесь к Повторы и отказы из-за ошибок сериализации/взаимоблокировки.
- Число повторов после ошибок сериализации или взаимоблокировки в этом операторе. За дополнительными сведениями обратитесь к Повторы и отказы из-за ошибок сериализации/взаимоблокировки.

В отчёте отображается статистика повторов, только если параметр `--max-tries` не равен 1.

Все эти значения вычисляются для каждого оператора, выполняемого каждым клиентом, и выдаются после завершения теста.

Для скрипта по умолчанию вывод будет выглядеть примерно так:

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
number of failed transactions: 0 (0.000%)
number of transactions above the 50.0 ms latency limit: 1311/10000 (13.110 %)
latency average = 28.488 ms
latency stddev = 21.009 ms
initial connection time = 69.068 ms
tps = 346.224794 (without initial connection time)
statement latencies in milliseconds and failures:
   0.012  0  \set aid random(1, 100000 * :scale)
   0.002  0  \set bid random(1, 1 * :scale)
```

```
   0.002  0  \set tid random(1, 10 * :scale)
   0.002  0  \set delta random(-5000, 5000)
   0.319  0  BEGIN;
   0.834  0  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
   0.641  0  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
  11.126  0  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
  12.961  0  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
   0.634  0  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
   1.957  0  END;
```

Ещё один пример вывода для скрипта по умолчанию с выбором сериализуемого уровня изоляции (`PGOPTIONS='-c default_transaction_isolation=serializable' pgbench ...`):

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 10
number of transactions per client: 1000
number of transactions actually processed: 6317/10000
number of failed transactions: 3683 (36.830%)
number of transactions retried: 7667 (76.670%)
total number of retries: 45339
number of transactions above the 50.0 ms latency limit: 106/6317 (1.678 %)
latency average = 17.016 ms
latency stddev = 13.283 ms
initial connection time = 45.017 ms
tps = 186.792667 (without initial connection time)
statement latencies in milliseconds, failures and retries:
  0.006     0      0  \set aid random(1, 100000 * :scale)
  0.001     0      0  \set bid random(1, 1 * :scale)
  0.001     0      0  \set tid random(1, 10 * :scale)
  0.001     0      0  \set delta random(-5000, 5000)
  0.385     0      0  BEGIN;
  0.773     0      1  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE
 aid = :aid;
  0.624     0      0  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
  1.098   320   3762  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid
 = :tid;
  0.582  3363  41576  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE
 bid = :bid;
  0.465     0      0  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
  1.933     0      0  END;
```

Если задействуется несколько файлов скриптов, все статистические данные выводятся отдельно для каждого файла.

Учтите, что сбор дополнительных временных показателей влечёт некоторые издержки и приводит к снижению средней скорости и, как результат, падению TPS. На сколько именно снизится скорость, во многом зависит от платформы и оборудования. Хороший способ оценить, каковы эти издержки — сравнить средние значения TPS, получаемые с подсчётом времени операторов и без такого подсчёта.

Повторы и отказы из-за ошибок сериализации/взаимоблокировки

При выполнении pgbench могут возникать ошибки трёх основных типов:

- Ошибки основной программы. Они являются наиболее серьёзными и всегда приводят к немедленному завершению pgbench с соответствующим сообщением об ошибке. К ним относятся:
  - ошибки, возникшие при запуске pgbench (например, из-за недопустимого значения параметра);
  - ошибки в режиме инициализации (например, запрос из встроенного скрипта не смог создать таблицу);
  - ошибки перед запуском потоков (например, не удалось подключиться к серверу базы данных, синтаксическая ошибка в метакоманде, сбой при создании потока);
  - непредвиденные внутренние ошибки pgbench.
- Ошибки при обслуживании клиентов в потоке выполнения (например, клиент не смог установить соединение с сервером, сокет для подключения клиента к серверу стал недействительным). В таких случаях все клиенты этого потока останавливаются, а другие потоки продолжают работать. Однако если указать параметр `--exit-on-abort`, остановятся все потоки.
- Непосредственно клиентские ошибки. Они вызывают немедленное завершение pgbench с соответствующим сообщением только в случае непредвиденной внутренней ошибки pgbench или если указан параметр `--exit-on-abort`. В других случаях они могут вызвать лишь прерывание работы проблемного клиента, в то время как другие клиенты продолжат работу (однако некоторые клиентские ошибки обрабатываются без прерывания работы клиента и информация о них выдаётся отдельно, см. ниже). Далее в этом разделе ошибками считаются только непосредственно клиентские ошибки, а не внутренние ошибки pgbench.

Работа клиента прерывается в случае серьёзной ошибки; например, если было потеряно соединение с сервером баз данных или достигнут конец скрипта, а последняя транзакция не завершена. Кроме того, работа клиента прерывается, если выполнение команды SQL или метакоманды завершается сбоем по причинам, отличным от ошибок сериализации или взаимоблокировки. Если же команда SQL завершается ошибкой сериализации или взаимоблокировки, работа клиента продолжается. В таких случаях текущая транзакция откатывается, при этом клиентские переменные получают значения, которые они имели до начала этой транзакции (предполагается, что один скрипт транзакции содержит только одну транзакцию; за подробностями обратитесь к Каково содержание «транзакции», которую выполняет pgbench?). Транзакции с ошибками сериализации или взаимоблокировки повторяются после отката до успешного завершения либо до максимального числа повторений (которое задаётся параметром `--max-tries`), максимальной длительности повторений (которая задаётся параметром `--latency-limit`) или до истечения времени теста (это время задаётся параметром `--time`). Если последняя попытка завершается неудачей, транзакция считается неудачной, но работа клиента не прерывается.

Примечание

Если параметр `--max-tries` не указан, транзакции никогда не повторяются после ошибок сериализации или взаимоблокировки, так как его значение по умолчанию равно 1. Чтобы ограничить только общую длительность попыток, задайте неограниченное количество попыток (`--max-tries=0`) и установите параметр `--latency-limit`. Вы также можете ограничить общую продолжительность теста при неограниченном количестве попыток, используя параметр `--time`.

Учтите особенность повторения скриптов, содержащих несколько транзакций: скрипт всегда будет повторяться целиком, так что успешные транзакции в нём могут выполняться многократно.

Также учтите особенность повторения транзакций, содержащих команды оболочки. В отличие от результатов команд SQL, результаты команд оболочки не отменяются, за исключением значения переменной команды `\setshell`.

Длительность успешной транзакции включает общее время выполнения транзакции с учётом откатов и повторов. Измеряется длительность только для успешных транзакций и команд.

Количество неуспешных транзакций выводится в основном отчёте. Если параметр `--max-tries` не равен 1, основной отчёт также содержит статистику повторов: общее число транзакций, которые повторялись, и общее число повторов всех транзакций. Отчёт по скриптам наследует все эти поля от основного отчёта. Если параметр `--max-tries` не равен 1, статистика повторных попыток также отображается в отчёте по операторам.

Если вы хотите, чтобы в протоколе с агрегированием, а также в основном отчёте, отчёте по транзакциям и отчёте по скриптам сбои группировались по базовым типам, используйте параметр `--failures-detailed`. Если же вы хотите видеть все отдельные ошибки и отказы (ошибки, после которых транзакция не повторяется), а также видеть, как ограничиваются повторения и насколько достигается ограничение, используйте параметр `--verbose-errors`.

Табличные методы доступа

Можно определить табличный метод доступа к таблицам pgbench. В переменной окружения `PGOPTIONS` задаются конфигурационные параметры БД, которые передаются в Postgres Pro через командную строку (см. Подраздел 18.1.4). Например, табличный метод доступа по умолчанию к таблицам, создаваемым pgbench, под названием `wuzza` можно определить так:

```
PGOPTIONS='-c default_table_access_method=wuzza'
```

Полезные советы

Используя pgbench, можно без особого труда получить абсолютно бессмысленные числа. Последуйте приведённым советам, чтобы получить полезные результаты.

Во-первых, *никогда* не доверяйте тестам, которые выполняются всего несколько секунд. Воспользуйтесь параметром `-t` и `-T` и установите время выполнения не меньше нескольких минут, чтобы избавиться от шума в средних значениях. В некоторых случаях для получения воспроизводимых результатов тестирование должно продолжаться несколько часов. Чтобы понять, были ли получены воспроизводимые значения, имеет смысл запустить тестирование несколько раз.

Для стандартного сценария по типу TPC-B начальный коэффициент масштаба (`-s`) должен быть не меньше числа клиентов, с каким вы намерены проводить тестирование (`-c`); в противном случае вы, по большому счёту, будете замерять время конкурентных изменений. Таблица `pgbench_branches` содержит всего `-s` строк, а каждая транзакция хочет изменить одну из них, так что если значение `-c` превышает `-s`, это несомненно приведёт к тому, что многие транзакции будут блокироваться другими.

Стандартный сценарий тестирования также довольно сильно зависит от того, сколько времени прошло с момента инициализации таблиц: накопление неактуальных строк и «мёртвого» пространства в таблицах влияет на результаты. Чтобы правильно оценить результаты, необходимо учитывать, сколько всего изменений было произведено и когда выполнялась очистка. Если же включена автоочистка, это может быть чревато непредсказуемыми изменениями оценок производительности.

Полезность результатов pgbench также может ограничиваться тем, что тестирование с большим числом клиентских сеансов само по себе нагружает систему. Этого можно избежать, запуская pgbench на другом компьютере, не на сервере баз данных, хотя при этом большое значение имеет скорость сети. Иногда, оценивая производительность одного сервера, полезно запускать даже несколько экземпляров pgbench параллельно, на отдельных клиентских компьютерах.

Безопасность

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, не запускайте pgbench в этой базе. Программа pgbench использует неполные имена и не настраивает для себя путь поиска.

pg_combinebackup

pg_combinebackup — восстановить полную резервную копию из инкрементальной и зависимых резервных копий

Синтаксис

`pg_combinebackup` [*параметр*...] [*каталог_резервных_копий*...]

Описание

pg_combinebackup используется для восстановления синтетической полной резервной копии из инкрементальной резервной копии и более ранних копий, от которых она зависит.

Укажите в командной строке все необходимые резервные копии — от самой старой к самой новой. То есть первый каталог резервных копий должен быть путём к полной копии, а последний — путём к последней инкрементальной копии, которую нужно восстановить. Восстановленная копия будет записана в выходной каталог, указанный в параметре `-o`.

Приложение pg_combinebackup попытается проверить, что указанные резервные копии образуют допустимую цепочку, из которой может быть восстановлена корректная полная резервная копия. Однако в нём не предусмотрена функциональность, помогающая отслеживать зависимости между копиями. Если будет удалена одна или несколько предыдущих резервных копий, от которых зависит инкрементальная копия, вы не сможете её восстановить. Более того, pg_combinebackup пытается проверить только корректность связей между копиями, а не целостность каждой из них. Чтобы проверить целостность, используйте pg_verifybackup.

Поскольку вывод pg_combinebackup представляет собой синтетическую полную резервную копию, её можно использовать в качестве входных данных для следующего вызова pg_combinebackup. Синтетическая полная резервная копия будет указана в командной строке вместо цепочки резервных копий, из которых она была восстановлена.

Параметры

`-d`
`--debug`

Выводить подробные отладочные сообщения в `stderr`.

`-n`
`--dry-run`

Параметр `-n`/`--dry-run` даёт указание `pg_combinebackup` выяснить, что будет сделано без фактического создания целевого каталога или каких-либо выходных файлов. Он особенно полезен в сочетании с `--debug`.

`-N`
`--no-sync`

По умолчанию `pg_combinebackup` ожидает безопасной записи файлов на диск. Этот параметр заставляет `pg_combinebackup` возвращать результат без ожидания, что быстрее, но означает, что последующий сбой операционной системы может повредить получившуюся резервную копию. Как правило, этот параметр полезен для тестирования, но его не следует использовать в производственной среде.

`-o` *`выходной_каталог`*
`--output=`*`выходной_каталог`*

Указывает выходной каталог, в который должна быть записана синтетическая полная резервная копия. В настоящее время этот аргумент является обязательным.

`-T` *`старый_каталог`*`=`*`новый_каталог`*
`--tablespace-mapping=`*`старый_каталог`*`=`*`новый_каталог`*

Перемещает табличное пространство из *`старого_каталога`* в *`новый_каталог`* во время резервного копирования. *`старый_каталог`* — это абсолютный путь табличного пространства в том виде, в котором оно существует в последней резервной копии, указанной в командной строке, а *`новый_каталог`* — это абсолютный путь для использования табличного пространства в восстановленной резервной копии. Если какой-либо из путей должен содержать знак равенства (=), поставьте перед ним обратную косую черту. Этот параметр можно указывать несколько раз для нескольких табличных пространств.

`--clone`

Использовать эффективное клонирование файлов (в ряде систем это называется «reflink») вместо копирования файлов в новый каталог. В результате файлы данных могут копироваться практически мгновенно.

Если манифест копии недоступен или не содержит контрольную сумму нужного типа, для копирования файла будет использовано клонирование, но файл также будет прочитан по блокам для расчёта контрольной суммы.

Клонирование файлов поддерживается только в некоторых операционных и файловых системах. Если оно выбрано, но не поддерживается, запуск pg_combinebackup приведёт к ошибке. В настоящее время клонирование поддерживается в Linux (ядро 4.5 или более поздней версии) с Btrfs и XFS (в файловых системах, созданных с поддержкой reflink), а также в macOS с APFS.

`--copy`

Выполнять обычное копирование файлов. Это значение по умолчанию. (См. также `--copy-file-range` и `--clone`.)

`--copy-file-range`

Использовать системный вызов `copy_file_range` для эффективного копирования. В некоторых файловых системах результаты будут схожи с `--clone`, позволяя совместное использование физических дисковых блоков, в то время как на других могут копироваться блоки, но по оптимизированному пути. В настоящее время параметр поддерживается в Linux и FreeBSD.

Если манифест копии недоступен или не содержит контрольную сумму нужного типа, для копирования файла будет использована функция `copy_file_range`, но файл также будет прочитан по блокам для расчёта контрольной суммы.

`--manifest-checksums=`*`алгоритм`*

Как и pg_basebackup, pg_combinebackup записывает манифест копии в выходной каталог. В этом параметре указывается алгоритм подсчёта контрольных сумм, который следует применять к каждому файлу, включённому в манифест. В настоящее время доступны следующие алгоритмы: `NONE`, `CRC32C`, `SHA224`, `SHA256`, `SHA384` и `SHA512`. Значение по умолчанию — `CRC32C`.

`--no-manifest`

Отключить генерацию манифеста копии. Если этот параметр не указан, манифест для восстановленной резервной копии будет записан в выходной каталог.

`--sync-method=`*`метод`*

При установке `fsync`, являющимся значением по умолчанию, `pg_combinebackup` рекурсивно открывает и синхронизирует все файлы в каталоге резервных копий. При использовании простого формата поиск по файлам будет переходить по символическим ссылкам для каталогов WAL и каждого настроенного табличного пространства.

В Linux вместо этого можно использовать `syncfs`, чтобы дать указание ОС синхронизировать всю файловую систему, содержащую каталог резервных копий. При использовании простого формата `pg_combinebackup` также синхронизирует файловые системы, содержащие файлы WAL и каждое табличное пространство. Для получения дополнительной информации об особенностях, которые необходимо учитывать при использовании `syncfs`, обратитесь к recovery_init_sync_method.

В режиме `--no-sync` этот параметр не оказывает никакого влияния.

`-V`
`--version`

Вывести версию pg_combinebackup и завершиться.

`-?`
`--help`

Вывести справку по аргументам командной строки pg_combinebackup и завершиться.

Ограничения

`pg_combinebackup` не пересчитывает контрольные суммы страниц при записи выходного каталога. Поэтому, если какие-либо резервные копии, используемые для восстановления, были сделаны с отключёнными контрольными суммами, а последняя резервная копия — с включёнными, результирующий каталог может содержать страницы с недействительными контрольными суммами.

Чтобы избежать этой проблемы, рекомендуется сделать новую полную резервную копию после изменения состояния контрольной суммы кластера с помощью pg_checksums. В противном случае, чтобы исправить эту проблему, можно отключить и затем снова при необходимости включить контрольные суммы в каталоге, созданном `pg_combinebackup`.

Переменные окружения

Как и большинство других утилит PostgreSQL, приложение также использует переменные окружения, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

См. также

pg_basebackup

pg_config

pg_config — вывести информацию об установленной версии Postgres Pro

Синтаксис

`pg_config` [*параметр*...]

Описание

Утилита pg_config выводит параметры конфигурации текущей установленной версии Postgres Pro. Это помогает, например, найти заголовочные файлы и библиотеки, требующиеся программным средствам, которые хотят взаимодействовать с Postgres Pro.

Параметры

При использовании pg_config можно передать следующие параметры:

`--bindir`

Вывести расположение исполняемых файлов. Можно использовать, например, для поиска утилиты `psql`. Обычно там же находится и сама утилита `pg_config`.

`--docdir`

Вывести расположение файлов документации.

`--htmldir`

Вывести расположение файлов документации в формате HTML.

`--includedir`

Вывести расположение заголовочных C-файлов клиентских интерфейсов.

`--pkgincludedir`

Вывести расположение других заголовочных C-файлов.

`--includedir-server`

Вывести расположение заголовочных C-файлов для программирования серверной части.

`--libdir`

Вывести расположение библиотек объектного кода.

`--pkglibdir`

Вывести расположение динамически подгружаемых модулей, либо путь, где сервер должен их искать. По этому пути также могут размещаться и другие архитектурно-зависимые файлы.

`--localedir`

Вывести расположение файлов поддержки локалей. Если поддержка локалей не была сконфигурирована на этапе сборки Postgres Pro, будет выведена пустая строка.

`--mandir`

Вывести расположение страниц руководства man.

`--sharedir`

Вывести расположение архитектурно-независимых вспомогательных файлов.

`--sysconfdir`

Вывести расположение системных конфигурационных файлов.

`--pgxs`

Вывести расположение файлов сборки расширений.

`--configure`

Вывести список параметров `configure`, использованных при сборке Postgres Pro. Это может пригодиться, чтобы при последующей сборке сделать идентичную конфигурацию. Или для того, чтобы найти с какими параметрами был собран используемый бинарный пакет. (Стоит отметить, что бинарные пакеты нередко содержат патчи, специфичные для дистрибутивов.) См. примеры ниже.

`--cc`

Вывести использованное при сборке Postgres Pro значение переменной `CC`. Оно отражает, какой C-компилятор применялся.

`--cppflags`

Вывести использованное при сборке Postgres Pro значение переменной `CPPFLAGS`. Оно отражает флаги C-компилятора, применённые для препроцессора. Обычно это флаги `-I`.

`--cflags`

Вывести использованное при сборке Postgres Pro значение переменной `CFLAGS`. Оно отражает флаги C-компилятора, применённые при сборке.

`--cflags_sl`

Вывести использованное при сборке Postgres Pro значение переменной `CFLAGS_SL`. Оно отражает дополнительные флаги C-компилятора для сборки разделяемых библиотек.

`--ldflags`

Вывести использованное при сборке Postgres Pro значение переменной `LDFLAGS`. Оно отражает флаги компоновщика.

`--ldflags_ex`

Вывести использованное при сборке Postgres Pro значение переменной `LDFLAGS_EX`. Оно отражает флаги компоновщика, использованные при сборке лишь исполняемых файлов.

`--ldflags_sl`

Вывести использованное при сборке Postgres Pro значение переменной `LDFLAGS_SL`. Оно отражает флаги компоновщика, использованные при сборке лишь разделяемых библиотек.

`--libs`

Вывести использованное при сборке Postgres Pro значение переменной `LIBS`. Обычно оно отражает флаги подключения внешних библиотек к Postgres Pro, переданные с ключом `-l`.

`--version`

Вывести версию PostgreSQL, на которой основан Postgres Pro.

`--pgpro-version`

Вывести версию Postgres Pro.

`--pgpro-edition`

Вывести редакцию Postgres Pro.

`-?`
`--help`

Вывести справку по команде pg_config.

Если одновременно передано несколько параметров, то выводимая информация будет следовать согласно их порядку. Если параметры не переданы, то будет выведена вся информация с подписями, к чему она относится.

Примечания

Параметры `--docdir`, `--pkgincludedir`, `--localedir`, `--mandir`, `--sharedir`, `--sysconfdir`, `--cc`, `--cppflags`, `--cflags`, `--cflags_sl`, `--ldflags`, `--ldflags_sl` и `--libs` доступны, начиная с версии PostgreSQL 8.1. Параметр `--htmldir` добавлен в PostgreSQL 8.4. Параметр `--ldflags_ex` добавлен в PostgreSQL 9.0.

Пример

Чтобы воспроизвести конфигурацию сборки текущей инсталляции Postgres Pro, можно выполнить команду:

```
eval ./configure `pg_config --configure`
```

Вывод `pg_config --configure` содержит символы экранирования, поэтому значения аргументов, содержащие пробелы, представлены корректно. Таким образом, для получения корректного результата необходимо применить `eval`.

pg_dump

pg_dump — выгрузить базу данных Postgres Pro в виде скрипта или в архивном формате

Синтаксис

`pg_dump` [*параметр-подключения*...] [*параметр*...] [*имя_бд*]

Описание

pg_dump — это утилита для экспорта базы данных Postgres Pro. Она выполняет целостный экспорт, даже если база данных параллельно используется. pg_dump не препятствует доступу других пользователей к базе данных (ни для чтения, ни для записи). Однако обратите внимание, что за исключением простых случаев, не следует выбирать pg_dump для создания регулярных резервных копий производственных баз данных. За подробностями обратитесь к Главе 24.

pg_dump выгружает только одну базу данных. Чтобы экспортировать весь кластер или глобальные объекты, относящиеся ко всем базам в кластере, например, роли и табличные пространства, используйте pg_dumpall.

Выгружаемые данные могут быть сохранены в виде скрипта, либо в одном из архивных форматов. Скрипты представляют собой текстовые файлы, содержащие SQL-команды, необходимые для воссоздания базы данных до состояния на момент создания скрипта. Для восстановления из скрипта его содержимое можно передать psql. Скрипты можно использовать для восстановления базы на других машинах, в том числе с иной архитектурой, а с некоторыми коррективами даже в других СУБД.

Для восстановления из архивных форматов файлов используется утилита pg_restore. Эти форматы позволяют указывать pg_restore какие объекты базы данных восстановить, а также позволяют изменить порядок следования восстанавливаемых объектов. Архивные форматы файлов спроектированы так, чтобы их можно были переносить на другие платформы с другой архитектурой.

Применение архивных форматов в сочетании утилит pg_restore и pg_dump позволяет организовывать эффективный механизм архивации и переноса данных. pg_dump можно использовать для экспорта всей базы данных, а затем при применении pg_restore выбрать нужные объекты для восстановления. Наиболее гибкие форматы выходных файлов это «custom» (`-Fc`) и «directory» (`-Fd`). Они позволяют выбрать и изменить порядок объектов, поддерживают восстановление в несколько потоков, а также сжимаются по умолчанию. При этом только формат «directory» поддерживает выгрузку данных в несколько потоков.

Во время работы pg_dump следует обращать внимание на предупреждения, которые печатаются в стандартный поток ошибок, особенно ввиду рассмотренных далее ограничений.

Предупреждение

При восстановлении выгрузки в целевой точке выполняется произвольный код по выбору суперпользователей исходной точки. Для частичных выгрузок и частичных восстановлений такого ограничения нет. Если суперпользователи исходной точки не являются доверенными, выгружаемые команды SQL нужно проверять перед восстановлением. Выгрузки в нетекстовом формате можно проверить через параметр `--file` pg_restore. Обратите внимание, что клиенту, выполняющий выгрузку и восстановление, не нужно доверять суперпользователям исходной или целевой точки.

Параметры

Параметры командной строки для управления содержимым и форматом вывода.

имя_бд

Указывает имя базы данных, из которой будут выгружаться данные. Если имя не задано, то используется значение переменной окружения `PGDATABASE`. Если и переменная не задана, то в качестве имени базы будет взято имя пользователя, под которым осуществляется подключение.

`-a`
`--data-only`

Выводить только данные, но не схемы объектов (DDL). Будут копироваться данные таблиц, большие объекты, значения последовательностей.

Флаг похож на `--section=data`, но по историческим причинам не равнозначен ему.

`-b`
`--large-objects`
`--blobs` (устаревший параметр)

Включить большие объекты в выгрузку. Это поведение по умолчанию при отсутствии ключей `--schema`, `--table` или `--schema-only`. Таким образом, ключ `-b` полезен, лишь когда нужно добавить большие объекты при выгрузке только избранной схемы или таблицы. Заметьте, что большие объекты относятся к данным, и поэтому будут выгружаться, когда используется ключ `--data-only`, но не ключ `--schema-only`.

`-B`
`--no-large-objects`
`--no-blobs` (устаревший параметр)

Исключить из выгрузки большие объекты.

Когда задаётся и `-b`, и `-B`, большие объекты при выгрузке данных будут выводиться (см. описание ключа `-b`).

`-c`
`--clean`

Включить в выходной файл команды удаления `DROP` всех выгружаемых объектов базы данных перед командами создания этих объектов. Этот параметр полезен, если при восстановлении необходимо перезаписать существующую базу данных. Если дополнительно не указать флаг `--if-exists`, то при восстановлении в базу данных, где некоторые объекты отсутствуют, попытка удаления несуществующего объекта будет приводить к ошибке, которую можно игнорировать.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`-C`
`--create`

Сформировать в начале вывода команду для создания базы данных и затем подключения к ней. В этом случае не важно, какая база указана в параметрах подключения перед выполнением скрипта. Также, если указан ключ `--clean`, то скрипт сначала удалит, а затем пересоздаст базу данных перед подключением к ней.

С ключом `--create` в выходной файл также включается комментарий к базе данных (если он задан) и все назначения переменных конфигурации, связанные с базой данных, то есть все команды `ALTER DATABASE ... SET ...` и `ALTER ROLE ... IN DATABASE ... SET ...`, ссылающиеся на эту базу данных. Также выгружаются права доступа к самой базе данных, если не добавлен ключ `--no-acl`.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`-e` *`шаблон`*
`--extension=`*`шаблон`*

Выгрузить только расширения, соответствующие *`шаблону`*. Когда этот параметр отсутствует, выгружаются все несистемные расширения в целевой базе данных. Чтобы выгрузить несколько расширений, ключ `-e` можно задать несколько раз. Параметр *`шаблон`* интерпретируется по тем же правилам, что и шаблон в командах psql `\d` (см. Шаблоны поиска ниже), так что несколько расширений можно выбрать и шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости заключайте шаблон в кавычки, чтобы эти знаки не разворачивала оболочка системы.

Для расширения, указанного в `--extension`, в выгрузку включаются все его конфигурационные данные, зарегистрированные функцией `pg_extension_config_dump`.

> ### Примечание
>
> При использовании `-e`, pg_dump не выгружает прочие объекты, от которых выгружаемые расширения могут зависеть. Таким образом не гарантируется, что выгруженные расширения будут успешно восстановлены в чистой базе данных.

`-E` *`кодировка`*
`--encoding=`*`кодировка`*

Создать копию в заданной кодировке. По умолчанию копия создаётся в кодировке, используемой базой данных. Другой способ достичь того же результата — задать желаемую кодировку в переменной окружения `PGCLIENTENCODING`. Поддерживаемые кодировки описаны в Подразделе 22.3.1.

`-f` *`файл`*
`--file=`*`файл`*

Отправить вывод в указанный файл. Параметр можно не указывать, если используется формат с выводом в файл. В этом случае будет использован стандартный вывод. Однако для формата с выводом в каталог параметр является обязательным и должен задавать путь к каталогу. В этом случае целевой каталог будет создан командой `pg_dump` и не должен существовать заранее.

`-F` *`format`*
`--format=`*`format`*

Указывает формат вывода копии. *`format`* может принимать следующие значения:

`p`
`plain`

Сформировать текстовый SQL-скрипт. Это поведение по умолчанию.

`c`
`custom`

Выгрузить данные в специальном архивном формате, пригодном для дальнейшего использования утилитой pg_restore. Наряду с форматом `directory` является наиболее гибким форматом, позволяющим вручную выбирать и сортировать восстанавливаемые объекты. Вывод в этом формате по умолчанию сжимается.

`d`
`directory`

Выгрузить данные в формате каталога. Этот формат пригоден для дальнейшего использования утилитой pg_restore. При этом будет создан каталог, в котором для каждой таблицы и большого объекта будут созданы отдельные файлы, а также файл оглавления в машинно-читаемом формате, понятном для pg_restore. С полученной резервной копией можно работать штатными средствами Unix, например, несжатую копию можно сжать посредством gzip, lz4 или zstd. Этот формат по умолчанию сжимается с использованием `gzip`, а также поддерживает работу в несколько потоков.

`t`
`tar`

Выгрузить данные в формате `tar`, для дальнейшего использования с утилитой pg_restore. Этот формат совместим с форматом вывода в каталог: если архив распаковать, получится корректная копия в формате каталога. Однако формат tar не поддерживает сжатие. Также, применяя формат tar, при восстановлении нельзя изменить относительный порядок элементов данных.

`-j` *`число_заданий`*
`--jobs=`*`число_заданий`*

Осуществить выгрузку в параллельном режиме, обрабатывая одновременно несколько таблиц (в количестве *`число_заданий`*). Это может сократить время, необходимое для выгрузки, но увеличивает нагрузку на сервер. Этот параметр можно использовать только с форматом вывода в каталог, так как это единственный формат, позволяющий нескольким процессам записывать данные одновременно.

pg_dump откроет *`число_заданий`* + 1 соединений с базой данных. Таким образом необходимо обеспечить достаточное значение параметра max_connections.

Если во время выгрузки в несколько потоков параллельно работающие сеансы будут запрашивать эксклюзивные блокировки на объекты базы данных, то pg_dump может завершиться аварийно. Дело в том, что головной процесс pg_dump вначале запрашивает разделяемые блокировки (ACCESS SHARE) на объекты, которые позже будут выгружать рабочие процессы. Это делается для того, чтобы никто не смог удалить объекты на время работы pg_dump. Если же другой сеанс запросит эксклюзивную блокировку на объект, то запрос на блокировку будет поставлен в очередь, до тех пор пока разделяемая блокировка головного процесса pg_dump не будет снята. В последующем, любая попытка доступа к этому объекту будет вставать в очередь, вслед за эксклюзивной блокировкой. В том числе в очередь попадет и рабочий процесс pg_dump. Если не принять меры предосторожности, то получим классическую взаимоблокировку. Для предупреждения подобных конфликтов, рабочий процесс pg_dump ещё раз запрашивает разделяемую блокировку на объект с указанием `NOWAIT`. И если он не смог получить блокировку, значит кто-то ещё запросил эксклюзивную блокировку объекта. А это значит, что нет возможности продолжить выгрузку, поэтому pg_dump прерывает дальнейшую работу.

Для выполнения параллельной выгрузки сервер баз данных должен поддерживать функциональность синхронизированных снимков, которая была введена в версии PostgreSQL 9.2 для ведущих серверов и в 10 для ведомых. Это позволяет разным клиентам работать с одной и той же версией данных, несмотря на использование разных подключений. `pg_dump -j` использует множественные подключения. Первое подключение осуществляется головным процессом, а последующие — рабочими процессами. Без функциональности синхронизируемых снимков нет гарантии того, что каждое подключение увидит одни и те же данные, что может привести к несогласованности данных резервной копии.

`-n` *`шаблон`*
`--schema=`*`шаблон`*

Выгрузить только схемы, соответствующие *`шаблону`*; вместе с этими схемами будут выгружены и все содержащиеся в них объекты. Когда этот параметр отсутствует, выгружаются все несистемные схемы в целевой базе данных. Чтобы выгрузить несколько схем, ключ `-n` можно указать несколько раз. Параметр *`шаблон`* интерпретируется по тем же правилам, что и шаблон в командах psql `\d` (см. Шаблоны поиска), так что несколько схем можно выбрать и шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости заключайте шаблон в кавычки, чтобы эти знаки не разворачивала оболочка системы (см. Примеры ниже).

Примечание

При использовании `-n`, pg_dump не выгружает объекты других схем, от которых выгружаемая схема может зависеть. Таким образом не гарантируется, что выгруженная схема будет успешно восстановлена в чистой базе данных.

Примечание

Не принадлежащие схемам объекты (например, большие объекты), не выгружаются с параметром `-n`. Однако можно указать `--large-objects`, чтобы они попали в выгрузку.

`-N` *`шаблон`*
`--exclude-schema=`*`шаблон`*

Не выгружать схемы, соответствующие *`шаблону`*. Шаблон интерпретируется по тем же правилам, что и для параметра `-n`. Параметр `-N` можно использовать в команде несколько раз для исключения схем, соответствующих нескольким шаблонам.

При одновременном использовании параметров `-n` и `-N` будут выгружаться схемы, соответствующие шаблону параметра `-n` и не противоречащие шаблону параметра `-N`.

`-O`
`--no-owner`

Не формировать команды, устанавливающие владельца объектов базы данных. По умолчанию pg_dump генерирует команды `ALTER OWNER` или `SET SESSION AUTHORIZATION` для назначения владельцев объектов базы. Эти команды завершатся неудачно, если скрипт будет запущен не суперпользователем или не владельцем объектов. Чтобы создать скрипт, который можно выполнить при восстановлении от лица произвольного пользователя и назначить его в качестве владельца объектов восстанавливаемой базы, необходимо указать флаг `-O`.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`-R`
`--no-reconnect`

Параметр является устаревшим, но в целях совместимости ещё работает.

`-s`
`--schema-only`

Выгружать только определения объектов (схемы), без данных.

Действие параметра противоположно действию `--data-only`. Это похоже на указание `--section=pre-data --section=post-data`, но по историческим причинам не равнозначно ему.

(Не путайте этот параметр с `--schema`, где слово «схема» используется в другом значении.)

Чтобы не выгружать данные отдельных таблиц, используйте параметр `--exclude-table-data`.

`-S` *`имя_пользователя`*
`--superuser=`*`имя_пользователя`*

Указать суперпользователя, который будет использоваться для отключения триггеров. Параметр имеет значение только вместе с `--disable-triggers`. Обычно его лучше не использовать, а запускать полученный скрипт от имени суперпользователя.

`-t` *`шаблон`*
`--table=`*`шаблон`*

Выгрузить только таблицы, соответствующие *`шаблону`*. Чтобы выбрать несколько таблиц, ключ `-t` можно указать несколько раз. Параметр *`шаблон`* интерпретируется по тем же правилам, что и шаблон в командах psql `\d` (см. Шаблоны поиска), так что несколько таблиц можно выбрать и шаблоном со знаками подстановки. Используя знаки подстановки, при необходимости заключайте шаблон в кавычки, чтобы эти знаки не разворачивала оболочка системы (см. Примеры ниже).

Помимо обычных таблиц, с этим указанием можно выгрузить определения соответствующих шаблону последовательностей, представлений, материализованных представлений и сторонних таблиц. При этом содержимое представлений и матпредставлений выгружаться не будет, а содержимое сторонних таблиц будет выгружено, только если в аргументе `--include-foreign-data` указан соответствующий сторонний сервер.

Параметры `-n` и `-N` не действуют в присутствии параметра `-t`, так как отобранные им таблицы всё равно будут выгружены, а не табличные объекты выгружаться не будут.

> ### Примечание
>
> При использовании `-t`, pg_dump не выгружает прочие объекты, от которых выгружаемые таблицы могут зависеть. Таким образом не гарантируется, что выгруженные таблицы будут успешно восстановлены в чистой базе данных.

`-T` *`шаблон`*
`--exclude-table=`*`шаблон`*

Не выгружать таблицы, соответствующие *`шаблону`*. Шаблон интерпретируется по тем же правилам, что и для параметра `-t`. Параметр `-T` можно использовать в команде несколько раз для исключения таблиц, соответствующих нескольким шаблонам.

При одновременном использовании параметров `-t` и `-T` будут выгружаться таблицы, соответствующие шаблону параметра `-t` и не противоречащие шаблону параметра `-T`.

`-v`
`--verbose`

Включить подробный режим. pg_dump будет выводить в стандартный поток ошибок подробные комментарии к объектам, включая время начала и окончания выгрузки, а также сообщения о прогрессе выполнения. Если повторить этот ключ, в стандартный поток ошибок будут выдаваться дополнительные отладочные сообщения.

`-V`
`--version`

Вывести версию pg_dump.

`-x`
`--no-privileges`
`--no-acl`

Не выгружать права доступа (команды GRANT/REVOKE).

`-Z` *`уровень`*
`-Z` *`метод`*[:*`строка_информации`*]
`--compress=`*`уровень`*
`--compress=`*`метод`*[:*`строка_информации`*]

Указывает метод и/или уровень сжатия. В качестве метода сжатия можно выбрать `gzip`, `lz4`, `zstd` или `none` (без сжатия). В качестве дополнительной информации можно передать параметры сжатия. Если в строке информации передаётся целое число, оно задаёт уровень сжатия. В противном случае она должна содержать список элементов, разделённых запятыми, в форме *`ключевое_слово`* или *`ключевое_слово=значение`*. На данный момент поддерживаются ключевые слова `level` и `long`.

Если уровень сжатия не указан, будет выбран уровень сжатия по умолчанию. Если указан только уровень сжатия, но не указан метод, будет применяться метод сжатия `gzip`, когда уровень больше `0`, а когда уровень равен `0`, сжатие не будет выполняться.

Для специального формата и формата каталога будут сжиматься файлы отдельных таблиц. По умолчанию применяется умеренный уровень сжатия с использованием `gzip`. Если указать отличный от нулевого уровень сжатия для простого формата, то сжиматься будет весь выходной файл, как это было бы при передаче файла команде gzip, lz4 или zstd. Однако по умолчанию для простого формата сжатие не производится. Сжатие с использованием zstd можно проводить с указанием `long`, что может улучшить коэффициент сжатия за счёт повышения нагрузки на память.

Формат `tar` в настоящий момент не поддерживает сжатие.

`--binary-upgrade`

Этот параметр предназначен для утилит обновления сервера. Использование для иных целей не рекомендуется и не поддерживается. Поведение параметра может быть изменено в последующих версиях без предварительного уведомления.

`--column-inserts`
`--attribute-inserts`

Выгружать данные таблиц в виде команд `INSERT` с явным указанием столбцов (`INSERT INTO` *`таблица`* `(`*`столбец`*`, ...) VALUES ...`). Скорость восстановления при этом значительно снизится, но данный вариант оправдан, когда загружать данные нужно не в Postgres Pro. При этом в случае каких-либо ошибок при загрузке данных будут потеряны только строки `INSERT`, где возникли ошибки, но не всё содержимое таблицы.

`--disable-dollar-quoting`

Этот параметр запрещает заключать в доллары тело функций, что оставляет возможность только заключать их в кавычки, применяя стандартный синтаксис SQL.

`--disable-triggers`

Используется при выгрузке одних данных. Указывает pg_dump включать в вывод команды для временного выключения триггеров при восстановлении в целевой базе данных. Применяется в ситуациях, когда существуют проверки ссылочной целостности или другие триггеры, которые необходимо выключить на время восстановления.

В настоящее время команды, генерируемые с параметром `--disable-triggers`, должны исполняться от имени суперпользователя. Таким образом, необходимо также передавать флаг `-S`, либо при восстановлении выполнять скрипт от имени суперпользователя.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`--enable-row-security`

Этот параметр имеет смысл только при выгрузке содержимого таблицы, для которой включена защита строк. По умолчанию pg_dump устанавливает для row_security значение off, чтобы убедиться, что выгружаются все данные из таблицы. Если пользователь не имеет достаточных прав для обхода защиты строк, выдаётся ошибка. Этот параметр указывает pg_dump включить row_security, что позволит пользователю выгрузить часть содержимого таблицы, к которой он имеет доступ.

Заметьте, что в настоящее время для использования этого параметра обычно желательно, чтобы данные были выгружены в формате `INSERT`, так как команда `COPY FROM` в процессе восстановления не поддерживает защиту строк.

`--exclude-extension=`*`шаблон`*

Не выгружать расширения, соответствующие *`шаблону`*. Шаблон интерпретируется по тем же правилам, что и для параметра `-e`. Параметр `--exclude-extension` можно использовать в команде несколько раз для исключения расширений, соответствующих нескольким шаблонам.

При одновременном использовании параметров `-e` и `-exclude-extension` будут выгружаться расширения, соответствующие шаблону параметра `-e`, но не шаблону параметра `--exclude-extension`. Использование `--exclude-extension` без `-e` исключит из выгрузки расширения, соответствующие `--exclude-extension`.

`--exclude-table-and-children=`*`шаблон`*

Этот параметр делает то же, что `-T`/`--exclude-table`, кроме того, что с ним также исключаются любые наследуемые дочерние таблицы или секции таблиц, соответствующих *`шаблону`*.

`--exclude-table-data=`*`шаблон`*

Не выгружать содержимое таблиц, соответствующих *`шаблону`*. Шаблон таблицы интерпретируется по тем же правилам, что и для параметра `-t`. Параметр `--exclude-table-data` можно использовать в команде несколько раз для исключения таблиц, соответствующих нескольким шаблонам. Полезно, когда нужно получить определение таблицы, без содержимого.

Чтобы не выгружать содержимое всех таблиц базы, используйте параметр `--schema-only`.

`--exclude-table-data-and-children=`*`шаблон`*

Этот параметр делает то же, что `--exclude-table-data`, кроме того, что с ним также исключаются данные любых наследуемых дочерних таблиц или секций таблиц, соответствующих *`шаблону`*.

`--extra-float-digits=`*`число_цифр`*

Выводить числа с плавающей точкой не с максимальной точностью, а с заданным значением `extra_float_digits`. При выгрузке данных в целях резервного копирования данный параметр использовать не следует.

`--filter=`*`имя_файла`*

Задаёт имя файла, из которого будут считываться шаблоны включения/исключения объектов из выгрузки. Шаблоны интерпретируются по тем же правилам, что и соответствующие параметры: `-t`/`--table`, `--table-and-children`, `-T`/`--exclude-table` и `--exclude-table-and-children` для таблиц; `-n`/`--schema` и `-N`/`--exclude-schema` для схем; `--include-foreign-data` для данных сторонних серверов; `--exclude-table-data` и `--exclude-table-data-and-children` для табличных данных; `-e`/`--extension` и `--exclude-extension` для расширений. Для чтения из `STDIN` в качестве имени файла используйте `-`. Параметр `--filter` можно указывать вместе с вышеупомянутыми параметрами для включения/исключения объектов, а также использовать несколько раз, если нужно задать несколько файлов с фильтрами.

Файл содержит по одной строке на каждый шаблон объекта в следующем формате:

```
{ include | exclude } { extension | foreign_data | table | table_and_children |
 table_data | table_data_and_children | schema } ШАБЛОН
```

Первое ключевое слово указывает, будут ли объекты, соответствующие шаблону, включены или исключены. Второе ключевое слово указывает тип объектов, фильтруемых с помощью шаблона:

- `extension`: расширения; работает как параметр `-e`/`--extension` или `--exclude-extension`.
- `foreign_data`: данные на сторонних серверах; работает как параметр `--include-foreign-data`. Это ключевое слово может использоваться только с ключевым словом `include`.

- `table`: таблицы; работает как параметр `-t`/`--table` или `-T`/`--exclude-table`.
- `table_and_children`: таблицы, включающие любые секции или дочерние таблицы в иерархии наследования; работает как параметр `--table-and-children` или `--exclude-table-and-children`.
- `table_data`: табличные данные любых таблиц, соответствующих *шаблону*; работает как параметр `--exclude-table-data`. Это ключевое слово может использоваться только с ключевым словом `exclude`.
- `table_data_and_children`: табличные данные любых таблиц, соответствующих *шаблону*, а также любые секции или дочерние таблицы в иерархии наследования; работает как параметр `--exclude-table-data-and-children`. Это ключевое слово может использоваться только с ключевым словом `exclude`.
- `schema`: схемы; работает как параметр `-n`/`--schema` или `-N`/`--exclude-schema`.

Строки, начинающиеся с `#`, считаются комментариями и игнорируются. Комментарии также могут размещаться после строки с шаблоном объекта. Пустые строки также игнорируются. См. раздел Шаблоны поиска для информации о том, как применять экранирование в шаблонах.

Примеры файлов приведены ниже в Примеры.

`--if-exists`

Использовать команды `DROP ... IF EXISTS` для удаления объектов в режиме `--clean`. При этом возможные ошибки «does not exist» (не существует) не выводятся. Этот параметр недействителен без указания `--clean`.

`--include-foreign-data=`*`сторонний_сервер`*

Выгрузить данные всех сторонних таблиц со стороннего сервера, имя которого соответствует шаблону *сторонний_сервер*. Для выгрузки данных с нескольких сторонних серверов параметр `--include-foreign-data` можно использовать несколько раз. К тому же, значение *сторонний_сервер* интерпретируется как шаблон, согласно правилам, используемым командами `\d` утилиты psql (см. Шаблоны поиска). Поэтому несколько сторонних серверов можно также выбрать, используя в шаблоне символы подстановки. Когда используются символы подстановки, шаблон лучше экранировать кавычками, чтобы командная оболочка ОС не интерпретировала их по-своему (см. Примеры ниже). Единственное отличие от вышеупомянутых правил — шаблон не может быть пустым.

> ### Примечание
>
> Использование знаков подстановки в `--include-foreign-data` может привести к доступу к неподходящим сторонним серверам. Кроме того, для безопасного использования этого параметра необходимо убедиться, что у указанного сервера есть доверенный владелец.

> ### Примечание
>
> Когда используется параметр `--include-foreign-data`, pg_dump не проверяет возможность записи в стороннюю таблицу. Таким образом, не гарантируется, что выгруженная сторонняя таблица может быть успешно восстановлена.

`--inserts`

Выгружать данные таблиц в виде команд `INSERT` вместо `COPY`. Скорость восстановления при этом значительно снизится, но данный вариант оправдан, когда загружать данные нужно не в Postgres Pro. При этом в случае каких-либо ошибок при загрузке данных будут потеряны только строки `INSERT`, где возникли ошибки, но не всё содержимое таблицы. Заметьте, что восстановление может провалиться полностью, если у таблицы изменён порядок столбцов. В такой ситуации можно использовать параметр `--column-inserts`, для которого порядок столбцов не важен, но он работает ещё медленнее.

`--load-via-partition-root`

При выгрузке данных для секции таблицы выводить команды `COPY` или `INSERT`, ссылающиеся на корневую таблицу в иерархии секционирования, а не на эту секцию. В результате при загрузке данных подходящая секция будет выбираться заново для каждой строки. Это может быть полезно при восстановлении данных, когда на целевом сервере строки не всегда попадают в те же секции, в которых они находились на исходном. Это возможно, например, когда столбец секционирования имеет текстовый тип и в двух системах по-разному определено правило сортировки, по которому упорядочивается этот столбец.

`--lock-wait-timeout=`*`время_ожидания`*

Не ждать бесконечно получения разделяемых блокировок таблиц в начале процедуры выгрузки. Вместо этого выдать ошибку, если не удастся заблокировать таблицы за указанное *`время_ожидания`*. Это время можно задать в любом из форматов, принимаемых командой `SET statement_timeout`. (Допустимые форматы зависят от версии сервера, выгружающего данные, но количество миллисекунд в виде целого числа принимают все версии.)

`--no-comments`

Не выгружать комментарии.

`--no-publications`

Не выгружать публикации.

`--no-security-labels`

Не выгружать метки безопасности.

`--no-subscriptions`

Не выгружать подписки.

`--no-sync`

По умолчанию `pg_dump` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_dump` завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя операционной системы выгруженные данные могут оказаться испорченными. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

`--no-table-access-method`

Не выводить команды для выбора табличных методов доступа. При восстановлении все объекты будут создаваться с табличным методом доступа, выбираемым по умолчанию.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`--no-tablespaces`

Не формировать команды для указания табличных пространств. При восстановлении все объекты будут создаваться в табличном пространстве по умолчанию.

Этот параметр игнорируется, когда данные выгружаются в архивных форматах (не в текстовом). Для таких форматов данный параметр можно указать при вызове `pg_restore`.

`--no-toast-compression`

Не выдавать команды, задающие методы сжатия TOAST. С этим указанием все столбцы будут восстановлены с методом сжатия, выбранным по умолчанию.

`--no-unlogged-table-data`

Не выгружать данные нежурналируемых таблиц и последовательностей. Параметр не влияет на выгрузку определений таблиц и последовательностей, он только подавляет вывод их содержимого. С резервного сервера содержимое нежурналируемых таблиц и последовательностей не выгружается никогда.

`--on-conflict-do-nothing`

Добавить предложения `ON CONFLICT DO NOTHING` в команды `INSERT`. Это указание допускается только при выборе режима `--inserts`, `--column-inserts` или `--rows-per-insert`.

`--quote-all-identifiers`

Принудительно экранировать все идентификаторы. Этот параметр рекомендуется при выгрузке базы, когда основная версия сервера PostgreSQL, с которого выгружается база, отличается от версии pg_dump, или когда выгруженная копия предназначена для загрузки на сервере с другой основной версией. По умолчанию pg_dump экранирует только те идентификаторы, которые являются зарезервированными словами в собственной основной версии. Иногда это приводит к проблемам совместимости с серверами других версий, в которых множество зарезервированных слов может быть несколько другим. Применение параметра `--quote-all-identifiers` предотвращает подобные проблемы, ценой ухудшения читаемости скрипта с выгруженными данными.

`--restrict-key=`*`restrict_key`*

Использовать предоставленную строку в качестве ключа `\restrict` psql в выводе выгрузки. Задать это можно только для выгрузок в нетекстовом формате, например, если для параметра `--format` задано значение `plain` или параметра нет вовсе. Если ключ ограничения не указан, pg_dumpall сгенерирует случайный ключ. Ключи могут содержать только буквенные и цифровые символы.

Этот параметр в основном предназначен для целей тестирования и других сценариев, требующих повторяющегося создания вывода (например, для сравнения файлов выгрузки). Параметр не рекомендуется использовать в повседневной работе, поскольку, зная принципы функционирования ключа, злоумышленник может внедрить произвольный код, который будет выполнен на машине при помощи psql и вывода выгрузки.

`--rows-per-insert=`*`число_строк`*

Выгружать данные таблиц в виде команд `INSERT` вместо `COPY`. В данном параметре задаётся максимальное число строк для одной команды `INSERT`. Указанное в нём значение должно быть больше 0. При этом в случае каких-либо ошибок при загрузке данных будут потеряны только строки `INSERT`, где возникли ошибки, но не всё содержимое таблицы.

`--section=`*`имя_секции`*

Выгружать лишь указанную секцию. Имя секции может принимать значения `pre-data`, `data` или `post-data`. Для выгрузки нескольких секций, параметр можно использовать несколько раз в одной команде. По умолчанию резервируются все секции.

Секция `data` содержит непосредственно данные таблиц, больших объектов и значения последовательностей. Секция `post-data` содержит определения индексов, триггеров, правил и ограничений (кроме ограничений проверки, созданных без `NOT VALID`). Секция `pre-data` включает определения остальных элементов.

`--serializable-deferrable`

Использовать при выгрузке транзакцию с уровнем изоляции `serializable` для получения снимка, согласованного с последующими состояниями базы. Правда для этого нужно выждать момент, когда в потоке транзакций нет аномалий, и поэтому нет риска, что выгрузка завершится неудачно, и риска отката других транзакций с ошибкой `serialization_failure`. Более подробно изоляция транзакций и управление одновременным доступом описывается в Главе 13.

Параметр не особо полезен в случаях, когда требуется восстановление после сбоя. Он полезен для создания копии базы данных, в которой формируются отчёты и выполняются другие операции чтения, в то время как в основной базе продолжается обычная работа. Без этого параметра выгрузка может содержать не целостное состояние базы данных. Например, если используется пакетная обработка, статус пакета может отражаться как завершённый, в то время как в выгрузке будут не все элементы пакета.

Параметр не будет влиять на результат, если во время запуска pg_dump нет активных транзакций на чтение-запись. Если же активные транзакции чтения-записи есть, то начало выгрузки может быть отложено на неопределённый период времени. После того как выгрузка началась, производительность с этим ключом или без него будет одинаковой.

`--snapshot=`*`имя_снимка`*

Использовать заданный синхронный снимок при выгрузке данных из базы (за подробностями обратитесь к Таблице 9.98).

Этот параметр полезен, когда требуется синхронизировать выгружаемые данные со слотом логической репликации (см. Главу 47) или с другим одновременным сеансом.

В случае с параллельной выгрузкой будет использоваться имя снимка, определённое этим параметром; новый снимок не будет сделан.

`--strict-names`

Требует, чтобы каждому шаблону расширения (`-e`/`--extension`), схемы (`-n`/`--schema`) и таблицы (`-t`/`--table`) соответствовал минимум один объект (расширение/схема/таблица) в выгружаемой базе данных. Это также относится и к фильтрам, используемым в параметре `--filter`. Заметьте, что если не находится вообще ни одного такого объекта для заданных шаблонов, pg_dump выдаёт ошибку и без ключа `--strict-names`.

Этот параметр не действует на ключи `--exclude-extension`, `-N`/`--exclude-schema`, `-T`/`--exclude-table` или `--exclude-table-data`. Если не находятся объекты, соответствующие шаблонам исключения, это не считается ошибкой.

`--sync-method=`*`метод`*

Со значением `fsync` (по умолчанию) `pg_dump --format=directory` будет рекурсивно открывать и синхронизировать все файлы в каталоге архива.

В Linux возможен вариант `syncfs`, когда от ОС требуется синхронизировать целиком каждую из файловых систем, содержащих каталог архива. Более подробно особенности использования `syncfs` описаны в recovery_init_sync_method.

Этот параметр не действует в режиме `--no-sync` или если значение параметра `--format` не `directory`.

`--table-and-children=`*`шаблон`*

Этот параметр делает то же, что `-t`/`--table`, кроме того, что с ним также включаются любые наследуемые дочерние таблицы или секции таблиц, соответствующих *`шаблону`*.

`--use-set-session-authorization`

Выводить команды `SET SESSION AUTHORIZATION`, соответствующие стандарту, вместо `ALTER OWNER`, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов. Кроме того, для использования `SET SESSION AUTHORIZATION` при восстановлении нужны права суперпользователя, в то время как `ALTER OWNER` требует меньших привилегий.

`-?`
`--help`

Показать справку по аргументам командной строки pg_dump и завершиться.

Далее описаны параметры управления подключением.

`-d` *`имя_бд`*
`--dbname=`*`имя_бд`*

Указывает имя базы данных для подключения. Равнозначно указанию *`имя_бд`* в первом аргументе, не являющемся ключом, в командной строке. Вместо имени может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_dump запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_dump лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--role=`*`имя роли`*

Задаёт имя роли, которая будет осуществлять выгрузку. Получив это имя, pg_dump выполнит `SET ROLE` *`имя_роли`* после подключения к базе данных. Это полезно, когда проходящий проверку пользователь (указанный в `-U`) не имеет прав, необходимых для pg_dump, но может переключиться на роль, наделённую этими правами. В некоторых окружениях правила запрещают подключаться к серверу непосредственно суперпользователю, и этот параметр позволяет выполнить выгрузку, не нарушая их.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию.

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Диагностика

pg_dump на низком уровне выполняет команды `SELECT`. Если есть проблемы с работой pg_dump, убедитесь, что в базе данных можно выполнить `SELECT`, например из psql. Также следует учитывать, что при этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека libpq.

Обычно действия pg_dump в базе данных отслеживаются системой накопительной статистики. Если это нежелательно, то можно установить параметр `track_counts` в false в переменной окружения `PGOPTIONS` или в команде `ALTER USER`.

Примечания

Если в базу данных кластера `template1` устанавливались дополнительные объекты, то следует убедиться, что выгрузка pg_dump загружается в пустую базу данных. Иначе существует вероятность возникновения ошибок дублирования создаваемых объектов. Чтобы создать пустую базу данных, копируйте её из шаблона `template0`, вместо `template1`, например:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Если выгружаются только данные с одновременным использованием `--disable-triggers`, pg_dump сформирует команды для выключения табличных триггеров перед вставкой данных, а после них — команды, включающие триггеры обратно. Если восстановление будет прервано в середине процесса, системный каталог может остаться в неверном состоянии.

Сформированный pg_dump файл не содержит статистики, которую использует планировщик для принятия решений при планировании запросов. Поэтому после восстановления разумно будет выполнить `ANALYZE` для достижения оптимальной производительности; за дополнительными сведениями обратитесь к Подразделу 23.1.3 и Подразделу 23.1.6.

Так как pg_dump применяется для переноса данных в новые версии Postgres Pro, предполагается, что вывод pg_dump можно загрузить на сервер Postgres Pro более новой версии, чем версия pg_dump. pg_dump может также выгружать данные серверов Postgres Pro более старых версий, чем его собственная. (В настоящее время поддерживаются версии, начиная с PostgreSQL 9.2.) Однако утилита pg_dump не может выгружать данные с серверов Postgres Pro более новых основных версий; она не будет даже пытаться делать это, во избежание некорректной выгрузки. Также не гарантируется, что вывод pg_dump может быть загружен на сервере более старой основной версии — даже если данные были выгружены с сервера той же версии. Для загрузки такого файла на старом сервере может потребоваться вручную исправить в нём синтаксис, не воспринимаемый старой версией. Имея дело с разными версиями, рекомендуется применять параметр `--quote-all-identifiers`, так как он может предотвратить проблемы, возникающие при изменении множества зарезервированных слов в разных версиях PostgreSQL.

Выгружая подписки на логическую репликацию, pg_dump будет выдавать команды `CREATE SUBSCRIPTION` с указанием `connect = false`, так что при восстановлении подписки не будут устанавливаться удалённые подключения для создания слота репликации или для начального копирования таблиц. Таким образом, выгруженные данные могут быть восстановлены без сетевого доступа к удалённым серверам. Вновь активировать подписки должным образом — задача пользователя. Если задействованные серверы поменялись, возможно, придётся скорректировать информацию о подключении. Также может быть уместно опустошить целевые таблицы перед началом полного копирования таблиц. Если пользователь намерен копировать исходные данные во время обновления, он должен создать слот с параметром `two_phase = false`. После начальной синхронизации параметр `two_phase` будет автоматически включён подписчиком, если подписка изначально была создана с параметром `two_phase = true`.

Обычно рекомендуется использовать параметр `-X` (`--no-psqlrc`) при восстановлении базы данных из текстового скрипта pg_dump, чтобы обеспечить чистый процесс восстановления и избежать потенциальных конфликтов с нестандартными конфигурациями psql.

Примеры

Выгрузка базы данных `mydb` в файл SQL-скрипта:

```
$ pg_dump mydb > db.sql
```

Восстановление из ранее полученного скрипта в чистую базу `newdb`:

```
$ psql -X -d newdb -f db.sql
```

Выгрузка базы данных в специальном формате:

```
$ pg_dump -Fc mydb > db.dump
```

Выгрузка базы данных в формате каталога:

```
$ pg_dump -Fd mydb -f dumpdir
```

Выгрузка базы данных в формате каталога в 5 параллельных потоков:

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

Восстановление из архива в чистую новую базу данных `newdb`:

```
$ pg_restore -d newdb db.dump
```

Восстановление архива в ту же базу данных, из которой он был выгружен, с предварительным удалением текущего содержимого этой базы данных:

```
$ pg_restore -d postgres --clean --create db.dump
```

Выгрузка отдельной таблицы `mytab`:

```
$ pg_dump -t mytab mydb > db.sql
```

Выгрузка всех таблиц, имена которых начинаются с `emp` и которые принадлежат схеме `detroit`, кроме таблицы `employee_log`:

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

Выгрузка всех схем, имена которых начинаются с `east` или `west`, заканчиваются на `gsm` и не содержат `test`:

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

То же самое, но с использованием регулярного выражения:

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

Выгрузка всех объектов базы данных, кроме таблиц, имена которых начинаются с `ts_`:

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

Чтобы указать имя в верхнем или смешанном регистре в ключе `-t` и связанных с ним, это имя нужно заключить в кавычки; иначе оно будет приведено к нижнему регистру (см. Шаблоны поиска). Но кавычки являются спецсимволом для оболочки, поэтому и они, в свою очередь, должны заключаться в кавычки. Так, чтобы выгрузить одну таблицу с именем в смешанном регистре, нужно написать примерно следующее:

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

Чтобы выгрузить все таблицы, имена которых начинаются с `mytable`, кроме таблицы `mytable2`, укажите файл фильтра `filter.txt`, например:

```
include table mytable*
exclude table mytable2
```

```
$ pg_dump --filter=filter.txt mydb > db.sql
```

См. также

pg_dumpall, pg_restore, psql

pg_dumpall

pg_dumpall — выгрузить кластер баз данных Postgres Pro в файл скрипта

Синтаксис

`pg_dumpall` [*параметр-подключения*...] [*параметр*...]

Описание

Утилита pg_dumpall предназначена для записи («выгрузки») всех баз данных кластера Postgres Pro в один файл в формате скрипта. Этот файл содержит команды SQL, так что передав его на вход psql, можно восстановить все базы данных. Для формирования этого файла вызывается pg_dump для каждой базы данных в кластере. pg_dumpall также выгружает глобальные объекты, общие для всех баз данных, а именно роли, табличные пространства и права, выданные для параметров конфигурации. (Программа pg_dump не сохраняет эти объекты.)

Так как утилита pg_dumpall читает таблицы из всех баз данных, для получения полного содержимого баз запускать её, как правило, нужно от имени суперпользователя. Также права суперпользователя требуются при последующем выполнении сохранённого скрипта, чтобы он смог добавить роли и создать базы данных.

Генерируемый SQL-скрипт записывается в стандартное устройство вывода. Чтобы перенаправить его в файл, воспользуйтесь параметром `-f`/`--file` или операторами оболочки.

Утилите pg_dumpall требуется подключаться к серверу Postgres Pro несколько раз (к каждой базе по отдельности). Если вы проходите проверку подлинности по паролю, вам придётся каждый раз вводить пароль. Чтобы избежать этого, удобно иметь файл `~/.pgpass`. За дополнительными сведениями обратитесь к Разделу 32.16.

> ### Предупреждение
>
> При восстановлении выгрузки в целевой точке выполняется произвольный код по выбору суперпользователей исходной точки. Для частичных выгрузок и частичных восстановлений такого ограничения нет. Если суперпользователи исходной точки не являются доверенными, выгружаемые команды SQL нужно проверять перед восстановлением. Обратите внимание, что клиенту, выполняющий выгрузку и восстановление, не нужно доверять суперпользователям исходной или целевой точки.

Параметры

Параметры командной строки для управления содержимым и форматом вывода.

`-a`
`--data-only`

Выгружать только данные, без схемы (определений данных).

`-c`
`--clean`

Выводить SQL-команды для `DROP` всех выгруженных баз данных, ролей и табличных пространств перед их воссозданием. Этот параметр полезен, когда при восстановлении необходимо перезаписать существующий кластер. Без дополнительного указания `--if-exists` при этом могут выдаваться безвредные сообщения об ошибках, если таких объектов не окажется в целевой базе данных.

`-E` *`кодировка`*
`--encoding=`*`кодировка`*

Создать копию в заданной кодировке. По умолчанию копия создаётся в кодировке базы данных. (Другой способ достичь того же результата — задать желаемую кодировку в переменной окружения `PGCLIENTENCODING`.)

`-f` *`имя_файла`*
`--file=`*`имя_файла`*

Направить вывод в указанный файл. Если этот параметр опущен, скрипт записывается в стандартный вывод.

`--filter=`*`имя_файла`*

Указать имя файла, из которого будут считываться шаблоны для баз данных, исключаемых из выгрузки. Шаблоны интерпретируются согласно тем же правилам, что и для `--exclude-database`. Чтобы читать из `STDIN`, используйте `-` в качестве имени файла. Параметр `--filter` можно указывать вместе с `--exclude-database` для исключения баз данных, а также использовать несколько раз, если нужно задать несколько файлов с фильтрами.

Файл содержит по одной строке на каждый шаблон базы данных в следующем формате:

```
exclude database ШАБЛОН
```

Строки, начинающиеся с `#`, считаются комментариями и игнорируются. Комментарии также могут размещаться после строки с шаблоном объекта. Пустые строки также игнорируются. См. раздел Шаблоны поиска для информации о том, как применять экранирование в шаблонах.

`-g`
`--globals-only`

Выгружать только глобальные объекты (роли и табличные пространства), без баз данных.

`-O`
`--no-owner`

Не генерировать команды, устанавливающие владение объектами, как в исходной базе данных. По умолчанию, pg_dumpall генерирует команды `ALTER OWNER` или `SET SESSION AUTHORIZATION`, восстанавливающие исходных владельцев для создаваемых элементов схемы. Однако выполнить эти команды сможет только суперпользователь (или пользователь, владеющий всеми объектами, создаваемыми скриптом). Чтобы получить скрипт, который сможет восстановить любой пользователь (но при этом он станет владельцем всех объектов), используется `-O`.

`-r`
`--roles-only`

Выгружать только роли, без баз данных и табличных пространств.

`-s`
`--schema-only`

Выгружать только определения объектов (схемы), без данных.

`-S` *`имя_пользователя`*
`--superuser=`*`имя_пользователя`*

Указать суперпользователя, который будет использоваться для отключения триггеров. Параметр имеет значение только вместе с `--disable-triggers`. Обычно его лучше не использовать, а запускать полученный скрипт от имени суперпользователя.

`-t`
`--tablespaces-only`

Выгружать только табличные пространства, без баз данных и ролей.

`-v`
`--verbose`

Включить подробный режим. pg_dumpall будет выводить в стандартный поток ошибок подробные комментарии к объектам, включая время начала и окончания выгрузки, а также сообщения о прогрессе выполнения. Если повторить этот ключ, в стандартный поток ошибок будут выдаваться дополнительные отладочные сообщения. Этот ключ также передаётся программе pg_dump.

`-V`
`--version`

Сообщить версию pg_dumpall и завершиться.

`-x`
`--no-privileges`
`--no-acl`

Не выгружать права доступа (команды GRANT/REVOKE).

`--binary-upgrade`

Этот параметр предназначен для утилит обновления сервера. Использование для иных целей не рекомендуется и не поддерживается. Поведение параметра может быть изменено в последующих версиях без предварительного уведомления.

`--column-inserts`
`--attribute-inserts`

Выгружать данные в виде команд `INSERT` с явно задаваемыми именами столбцов (`INSERT INTO` *`таблица`* `(`*`столбец`*`, ...) VALUES ...`). При этом восстановление будет очень медленным; в основном это применяется для выгрузки данных, которые затем будут загружаться не в Postgres Pro.

`--disable-dollar-quoting`

Этот параметр запрещает заключать в доллары тело функций, что оставляет возможность только заключать их в кавычки, применяя стандартный синтаксис SQL.

`--disable-triggers`

Этот параметр действует только при выгрузке одних данных. С ним pg_dumpall добавляет команды, отключающие триггеры в целевых таблицах на время загрузки данных. Используйте его, если в ваших таблицах определены проверки ссылочной целостности или другие триггеры, которые вы не хотели бы выполнять в процессе загрузки данных.

В настоящее время команды, генерируемые с параметром `--disable-triggers`, должны исполняться от имени суперпользователя. Таким образом, необходимо также передавать флаг `-S`, либо при восстановлении выполнять скрипт от имени суперпользователя.

`--exclude-database=`*`шаблон`*

Не выгружать базы данных, имена которых соответствуют *`шаблону`*. Исключить имена по нескольким шаблонам можно, добавив несколько ключей `--exclude-database`. Параметр *`шаблон`* в данном аргументе обрабатывается по тем же правилам, что и в командах psql `\d` (см. Шаблоны поиска), что позволяет также исключить несколько баз данных, добавив в шаблон звёздочку. Используя звёздочку, заключайте шаблон в кавычки, если звёздочка может быть развёрнута оболочкой.

`--extra-float-digits=`*`число_цифр`*

Выводить числа с плавающей точкой не с максимальной точностью, а с заданным значением extra_float_digits. При выгрузке данных в целях резервного копирования данный параметр использовать не следует.

`--if-exists`

Использовать команды `DROP ... IF EXISTS` для удаления объектов в режиме `--clean`. При этом возможные ошибки «does not exist» (не существует) не выводятся. Этот параметр недействителен без указания `--clean`.

`--inserts`

Выгружать данные в виде команд `INSERT`, а не `COPY`. При этом восстановление значительно замедлится; в основном это применяется для выгрузки данных, которые затем будут загружаться не в Postgres Pro. Заметьте, что восстановление может вовсе не выполниться при изменении порядка столбцов в таблицах. В этом смысле параметр `--column-inserts` безопаснее, но восстановление будет ещё медленнее.

`--load-via-partition-root`

При выгрузке данных для секции таблицы выводить команды `COPY` или `INSERT`, ссылающиеся на корневую таблицу в иерархии секционирования, а не на эту секцию. В результате при загрузке данных подходящая секция будет выбираться заново для каждой строки. Это может быть полезно при восстановлении данных, когда на целевом сервере строки не всегда попадают в те же секции, в которых они находились на исходном. Это возможно, например, когда столбец секционирования имеет текстовый тип и в двух системах по-разному определено правило сортировки, по которому упорядочивается этот столбец.

`--lock-wait-timeout=`*`время_ожидания`*

Не ждать бесконечно получения разделяемых блокировок таблиц в начале процедуры выгрузки. Вместо этого выдать ошибку, если не удастся заблокировать таблицы за указанное *`время_ожидания`*. Это время можно задать в любом из форматов, принимаемых командой `SET statement_timeout`.

`--no-comments`

Не выгружать комментарии.

`--no-publications`

Не выгружать публикации.

`--no-role-passwords`

Не выгружать пароли ролей. При восстановлении все роли получат пароль NULL и не смогут пройти проверку подлинности, пока им не будут назначены пароли. Так как значения паролей не нужны, когда используется это указание, информация о ролях считывается из системного представления `pg_roles`, а не из `pg_authid`. Таким образом, данный вариант может быть также полезен, если доступ к `pg_authid` ограничен политикой безопасности.

`--no-security-labels`

Не выгружать метки безопасности.

`--no-subscriptions`

Не выгружать подписки.

`--no-sync`

По умолчанию `pg_dumpall` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_dumpall` завершается немедленно, то есть выполняется быстрее, но в случае неожиданного сбоя операционной системы выгруженные данные могут оказаться испорченными. Вообще этот параметр предназначен прежде всего для тестирования, для производственной среды он не подходит.

`--no-table-access-method`

Не выводить команды для выбора табличных методов доступа. При восстановлении все объекты будут создаваться с табличным методом доступа, выбираемым по умолчанию.

`--no-tablespaces`

Не выводить команды, создающие или выбирающие табличные пространства для объектов. С этим параметром все объекты будут созданы в пространстве по умолчанию, установленном во время восстановления.

`--no-toast-compression`

Не выдавать команды, задающие методы сжатия TOAST. С этим указанием все столбцы будут восстановлены с методом сжатия, выбранным по умолчанию.

`--no-unlogged-table-data`

Не выгружать содержимое нежурналируемых таблиц. Этот параметр не влияет на то, как выгружаются определения этих таблиц (схема); он отключает только выгрузку данных из них.

`--on-conflict-do-nothing`

Добавить предложения `ON CONFLICT DO NOTHING` в команды `INSERT`. Это указание допускается только при выборе режима `--inserts` или `--column-inserts`.

`--quote-all-identifiers`

Принудительно экранировать все идентификаторы. Этот параметр рекомендуется при выгрузке базы, когда основная версия сервера PostgreSQL, с которого выгружается база, отличается от версии pg_dumpall, или когда выгруженная копия предназначена для загрузки на сервере с другой основной версией. По умолчанию pg_dumpall экранирует только те идентификаторы, которые являются зарезервированными словами в собственной основной версии. Иногда это приводит к проблемам совместимости с серверами других версий, в которых множество зарезервированных слов может быть несколько другим. Применение параметра `--quote-all-identifiers` предотвращает подобные проблемы, ценой ухудшения читаемости скрипта с выгруженными данными.

`--restrict-key=`*`restrict_key`*

Использовать предоставленную строку в качестве ключа `\restrict` psql в выводе выгрузки. Если ключ ограничения не указан, pg_dumpall сгенерирует случайный ключ. Ключи могут содержать только буквенные и цифровые символы.

Этот параметр в основном предназначен для целей тестирования и других сценариев, требующих повторящегося создания вывода (например, для сравнения файлов выгрузки). Параметр не рекомендуется использовать в повседневной работе, поскольку, зная принципы функционирования ключа, злоумышленник может внедрить произвольный код, который будет выполнен на машине при помощи psql и вывода выгрузки.

`--rows-per-insert=`*`число_строк`*

Выгружать данные таблиц в виде команд `INSERT` вместо `COPY`. В данном параметре задаётся максимальное число строк для одной команды `INSERT`. Указанное в нём значение должно быть больше 0. При этом в случае каких-либо ошибок при восстановлении данных будут потеряны только строки `INSERT`, где возникли ошибки, но не всё содержимое таблицы.

`--use-set-session-authorization`

Выводить команды `SET SESSION AUTHORIZATION`, соответствующие стандарту, вместо `ALTER OWNER`, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов.

`-?`
`--help`

Показать справку по аргументам командной строки pg_dumpall и завершиться.

Далее описаны параметры управления подключением.

`-d` *`строка_подключения`*
`--dbname=`*`строка_подключения`*

Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Этот параметр называется `--dbname` для согласованности с другими клиентскими приложениями, но так как pg_dumpall подключается не к одной базе данных, имя базы в строке подключения игнорируется. Чтобы указать имя базы данных для начального подключения, которое будет использоваться для выгрузки глобальных объектов и обнаружения других выгружаемых баз, воспользуйтесь параметром `-l`.

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер баз данных. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-l` *`имя_бд`*
`--database=`*`имя_бд`*

Задаёт имя базы данных, через подключение к которой будут выгружаться глобальные объекты и находиться другие выгружаемые базы. По умолчанию используется база `postgres`, а в случае её отсутствия — `template1`.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_dumpall запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_dumpall лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Заметьте, что пароль будет запрашиваться повторно для выгрузки каждой базы данных. Поэтому обычно лучше настроить файл `~/.pgpass`, и не вводить пароль каждый раз вручную.

`--role=`*`имя роли`*

Задаёт имя роли, которая будет осуществлять выгрузку. Получив это имя, pg_dumpall выполнит `SET ROLE` *`имя_роли`* после подключения к базе данных. Это полезно, когда проходящий проверку пользователь (указанный в `-U`) не имеет прав, необходимых для pg_dumpall, но может переключиться на роль, наделённую этими правами. В некоторых окружениях правила запрещают подключаться к серверу непосредственно суперпользователю, и этот параметр позволяет выполнить выгрузку, не нарушая их.

Переменные окружения

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Примечания

Так как pg_dumpall внутри себя вызывает pg_dump, часть диагностических сообщений будет относиться к pg_dump.

Ключ `--clean` может быть полезен, даже если вы намереваетесь восстановить копию из скрипта в новом кластере. С `--clean` скрипт сможет удалить и пересоздать встроенные базы данных `postgres` и `template1`, так что они получат свойства, которые имели одноимённые базы в исходном кластере (например, локаль и кодировку). Без данного ключа эти базы сохранят свои свойства уровня базы данных, а также всё существующее содержимое.

После восстановления имеет смысл запустить `ANALYZE` для каждой базы данных, чтобы оптимизатор получил актуальную статистику. Также можно запустить анализ для всех баз данных, выполнив команду `vacuumdb -a -z`.

Не следует ожидать, что скрипт выгрузки будет выполняться абсолютно без ошибок. В частности, так как он будет содержать `CREATE ROLE` для каждой существующей в исходном кластере роли, при попытке создать суперпользователя определённо произойдёт ошибка «role already exists» (роль уже существует), если только целевой кластер был инициализирован не с другим начальным именем суперпользователя. Эта ошибка некритична и её следует просто игнорировать. С ключом `--clean` весьма вероятны другие незначительные сообщения об ошибках, связанные с несуществующими объектами; их число можно минимизировать, добавив ключ `--if-exists`.

При использовании pg_dumpall требуется, чтобы все необходимые каталоги табличных пространств существовали до восстановления; в противном случае создание баз данных в нестандартном размещении завершится ошибкой.

Обычно рекомендуется использовать параметр `-X` (`--no-psqlrc`) при восстановлении базы данных из скрипта pg_dumpall, чтобы обеспечить чистый процесс восстановления и избежать потенциальных конфликтов с нестандартными конфигурациями psql. Кроме того, поскольку скрипт pg_dumpall может содержать метакоманды psql, он может быть несовместим с клиентами, отличными от psql.

Примеры

Выгрузка всех баз данных:

```
$ pg_dumpall > db.out
```

Восстановить базы данных из этого файла можно так:

```
$ psql -X -f db.out -d postgres
```

К какой базе вы подключаетесь, в принципе, не имеет значения, так как скрипт, созданный утилитой pg_dumpall, будет содержать все команды, требующиеся для создания сохранённых баз данных и подключения к ним. Однако это важно, если применяется ключ `--clean` — тогда вы должны изначально подключиться к базе `postgres`; скрипт попытается прежде всего удалить остальные базы данных, но не сможет этого сделать для базы, к которой вы подключены.

См. также

Обратитесь к описанию pg_dump, чтобы узнать об условиях, при которых могут возникнуть проблемы.

pg_isready

pg_isready — проверить соединение с сервером Postgres Pro

Синтаксис

`pg_isready` [*`параметр-подключения`*...] [*`параметр`*...]

Описание

Утилита pg_isready предназначена для проверки соединения с сервером баз данных Postgres Pro. Результат проверки передаётся в коде завершения.

Параметры

`-d` *`dbname`*
`--dbname=`*`dbname`*

Указывает имя базы данных для подключения. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-h` *`компьютер`*
`--host=`*`компьютер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной среды `PGPORT`, если она установлена, либо числом, заданным при компиляции, обычно 5432.

`-q`
`--quiet`

Не выводить сообщение о состоянии. Это полезно в скриптах.

`-t` *`секунды`*
`--timeout=`*`секунды`*

Максимальное время ожидания (в секундах) при попытке подключения, по истечении которого констатируется, что сервер не отвечает. Значение по умолчанию — 3 секунды.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Подключиться к базе данных с заданным именем пользователя вместо подразумеваемого по умолчанию.

`-V`
`--version`

Сообщить версию pg_isready и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки pg_isready и завершиться.

Код завершения

Утилита pg_isready возвращает в оболочку `0`, если сервер принимает подключения, `1`, если он сбрасывает подключения (например, во время загрузки), `2`, если при попытке подключения не получен ответ, и `3`, если попытки подключения не было (например, из-за некорректных параметров).

Переменные окружения

Как и большинство других утилит Postgres Pro, `pg_isready` также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Чтобы получить состояние сервера, передавать имя пользователя, пароль и имя базы данных не требуется; но если передать некорректные значения, сервер выведет в журнал сообщение о неудачной попытке подключения.

Примеры

Обычное использование:

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

Запуск с параметрами подключения, во время загрузки кластера Postgres Pro:

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

Запуск с параметрами подключения, в случае, когда кластер Postgres Pro недоступен:

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

pg_receivewal

pg_receivewal — получить журналы предзаписи с сервера Postgres Pro

Синтаксис

`pg_receivewal` [*параметр*...]

Описание

Утилита pg_receivewal предназначена для приёма журнала предзаписи от работающего кластера Postgres Pro. Журнал предзаписи передаётся по протоколу потоковой репликации и записывается в локальный каталог. Затем этот каталог можно использовать в качестве архива для восстановления состояния на момент времени (см. Раздел 24.3).

pg_receivewal принимает журнал предзаписи в реальном времени по мере того, как он генерируется на сервере, и не ждёт завершения сегментов, как это делает archive_command и archive_library. Поэтому pg_receivewal можно использовать, не устанавливая archive_timeout.

В отличие от приёмника WAL, работающего на ведомом сервере Postgres Pro, pg_receivewal по умолчанию сохраняет на диск данные WAL, только когда файл WAL закрывается. Для сохранения данных WAL в реальном времени необходимо использовать ключ `--synchronous`. Так как приёмник pg_receivewal не применяет WAL, важно не допустить, чтобы он стал синхронным ведомым сервером, когда параметр synchronous_commit равен `remote_apply`. Если это произойдёт, он будет выглядеть как ведомый, который никогда не может нагнать ведущего, что приведёт к блокированию фиксации транзакций. Чтобы это предотвратить, нужно либо установить подходящее значение параметра synchronous_standby_names, либо задать для pg_receivewal такое имя приложения (`application_name`), которое не соответствует установленному имени, либо выбрать для `synchronous_commit` значение, отличное от `remote_apply`.

Журнал предзаписи передаётся через обычное подключение к Postgres Pro, с использованием протокола репликации. Подключение должен устанавливать пользователь с правом `REPLICATION` (см. Раздел 20.2) или суперпользователь, а в `pg_hba.conf` должно разрешаться подключение для репликации. Кроме того, параметр max_wal_senders на сервере должен быть достаточно большим, чтобы можно было создать ещё один сеанс для передачи потока.

Начальная точка передачи журнала предзаписи вычисляется при запуске pg_receivewal так:

1. Сначала сканируется каталог, в который помещаются файлы сегментов WAL, в нём выбирается последний завершённый файл сегмента, и начальной точкой считается начало следующего файла сегмента WAL.
2. Если вычислить начальную точку предыдущим способом не удаётся и при этом используется слот репликации, выдаётся дополнительная команда `READ_REPLICATION_SLOT` для получения значения `restart_lsn`, которое будет использоваться в качестве начальной точки. Этот вариант поддерживается только при потоковой передаче журналов предзаписи из Postgres Pro 15 и выше.
3. Если вычислить начальную точку предыдущим способом не удаётся, используется последняя позиция сохранённых данных в WAL, которую выдаёт команда `IDENTIFY_SYSTEM`.

Если подключение разорвалось или его с самого начала не удаётся установить из-за некритической ошибки, pg_receivewal будет бесконечно повторять попытки подключения и восстановит передачу, как только сможет. Чтобы отменить это поведение, воспользуйтесь параметром `-n`.

В отсутствие критических ошибок pg_receivewal будет выполняться до прерывания сигналом SIGINT (**Control**+**C**) или SIGTERM.

Параметры

`-D` *`каталог`*
`--directory=`*`каталог`*

Каталог, в который будут записываться данные.

Этот параметр является обязательным.

`-E` *`lsn`*
`--endpos=`*`lsn`*

Автоматически прекратить репликацию и завершить работу с кодом выхода 0 (без ошибки) при достижении заданного LSN.

Если будет получена запись с LSN, равным заданному *`lsn`*, она будет обработана.

`--if-not-exists`

Не выдавать ошибку, когда указан параметр `--create-slot` и слот с заданным именем уже существует.

`-n`
`--no-loop`

Не повторять цикл при ошибках подключения, а сразу завершать работу, возвращая ошибку.

`--no-sync`

С этим ключом `pg_receivewal` не будет принудительно сбрасывать данные WAL на диск. Этот вариант быстрее, но при последующем сбое операционной системы сегменты WAL могут оказаться испорченными. Обычно этот ключ полезен при тестировании, но при создании архива WAL в производственной среде использовать его не следует.

Этот ключ несовместим с `--synchronous`.

`-s` *`интервал`*
`--status-interval=`*`интервал`*

Указывает интервал в секундах между отправками серверу пакетов состояния. Это позволяет упростить мониторинг прогресса. Чтобы выключить периодическое обновление состояния, необходимо установить значение в ноль. При этом обновление будет отправляться по запросу сервера для избежания отсоединения по истечению времени. Значение по умолчанию составляет 10 секунд.

`-S` *`имя_слота`*
`--slot=`*`имя_слота`*

Указывает pg_receivewal использовать существующий слот репликации (см. Подраздел 25.2.6). Когда задан этот параметр, pg_receivewal будет сообщать серверу текущую позицию сохранения, отмечая, какой сегмент был сохранён на диске, чтобы сервер мог удалить этот сегмент, если он больше не нужен.

Когда клиент репликации pg_receivewal настроен на сервере как синхронный ведомый сервер, для используемого слота репликации серверу будет передаваться позиция сохранённых данных, но только когда файл WAL закрывается. Таким образом, в такой конфигурации транзакции на ведущем сервере будут ожидать завершения продолжительное время и по сути будут работать неудовлетворительно. Чтобы эта конфигурация работала корректно, нужно дополнительно указать параметр `--synchronous` (см. ниже).

`--synchronous`

Сохранять данные WAL на диск сразу после того, как они были получены. Также передавать пакет состояния сразу после сохранения, вне зависимости от `--status-interval`.

Этот параметр следует указывать, если клиент репликации pg_receivewal настроен на сервере как синхронный ведомый, чтобы обеспечить своевременную передачу ответа серверу.

`-v`
`--verbose`

Включает режим подробных сообщений.

`-Z` *`уровень`*
`-Z` *`метод`*`[:`*`дополнительная_информация`*`]`
`--compress=`*`уровень`*
`--compress=`*`метод`*`[:`*`дополнительная_информация`*`]`

Включает сжатие журналов предзаписи.

В качестве метода сжатия можно выбрать `gzip`, `lz4` (если Postgres Pro скомпилирован с параметром `--with-lz4`) или `none` (без сжатия). В качестве дополнительной информации можно передать параметры сжатия. Если в строке информации передаётся целое число, оно задаёт уровень сжатия. В противном случае она должна содержать список элементов, разделённых запятыми, в форме *`ключевое_слово`* или *`ключевое_слово=значение`*. На данный момент поддерживается только ключевое слово `level`.

Если уровень сжатия не указан, будет выбран уровень сжатия по умолчанию. Если указан только уровень сжатия, но не указан метод, будет применяться метод сжатия `gzip`, когда уровень больше 0, а когда уровень равен 0, сжатие не будет выполняться.

Когда используется метод `gzip`, ко всем именам файлов автоматически добавляется суффикс `.gz`, а когда `lz4` — суффикс `.lz4`.

Далее описаны параметры управления подключением.

`-d` *`строка_подключения`*
`--dbname=`*`строка_подключения`*

Указывает параметры подключения к серверу в формате строки подключения; они будут переопределять любые одноимённые параметры, заданные в командной строке.

Этот параметр называется `--dbname` для согласованности с другими клиентскими приложениями, но так как pg_receivewal не подключается к какой-либо конкретной базе данных, любое имя базы данных в строке подключения игнорируется сервером. Тем не менее переданное таким образом имя базы данных переопределяет имя по умолчанию (`replication`) для поиска пароля соединения репликации в `~/.pgpass`. Подобным образом это имя может использоваться при подключении через средний слой или прокси-сервер для, например, маршрутизации соединения.

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_receivewal запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_receivewal лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

pg_receivewal может выполнить одно из двух действий в отношении слотов физической репликации:

`--create-slot`

Создать слот физической репликации с именем, заданным аргументом `--slot`, и завершиться.

`--drop-slot`

Удалить слот репликации с именем, заданным аргументом `--slot`, и завершиться.

Другие флаги:

`-V`
`--version`

Сообщить версию pg_receivewal и завершиться.

`-?`
`--help`

Вывести справку по аргументам командной строки pg_receivewal и завершиться.

Код завершения

pg_receivewal завершится с кодом 0 при прерывании сигналом SIGINT или SIGTERM. (Это штатный способ его завершения, поэтому получение этого сигнала не считается ошибкой.) При критических ошибках или получении других сигналов код завершения будет ненулевым.

Переменные окружения

Как и большинство других утилит Postgres Pro, приложение также использует переменные окружения, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Применяя pg_receivewal вместо archive_command или archive_library в качестве основного способа резервного копирования WAL, настоятельно рекомендуется использовать слоты репликации. В противном случае сервер вполне может переписать или удалить файлы журнала предзаписи, прежде чем они будут скопированы, так как он не получает никакой информации, через archive_command/archive_library или слоты репликации, о том, как проходит архивация потока WAL. Учтите, однако, что при использовании слота репликации может заполниться всё место на диске, если принимающая сторона не будет успевать принимать данные WAL.

pg_receivewal сохранит разрешения для группы в полученных файлах WAL, если такие разрешения установлены в исходном кластере.

Примеры

Следующая команда принимает журнал предзаписи с сервера `mydbserver` и сохраняет его в локальном каталоге `/usr/local/pgsql/archive`:

```
$ pg_receivewal -h mydbserver -D /usr/local/pgsql/archive
```

См. также

pg_basebackup

pg_recvlogical

pg_recvlogical — управлять потоками логического декодирования Postgres Pro

Синтаксис

`pg_recvlogical` [*параметр*...]

Описание

Утилита `pg_recvlogical` управляет слотами логического декодирования и принимает данные из таких слотов репликации.

Она создаёт соединение в режиме репликации, так что на него распространяются те же ограничения, что и с pg_receivewal, плюс ограничения логической репликации (см. Главу 47).

`pg_recvlogical` не предоставляет возможностей, соответствующих режимам peek и get в SQL-интерфейсе логического декодирования. Она передаёт серверу подтверждения воспроизведения данных по мере их получения и при штатном выходе. Чтобы просмотреть данные, ожидающие передачи через слот, не принимая их, воспользуйтесь функцией `pg_logical_slot_peek_changes`.

В отсутствие критических ошибок pg_recvlogical будет выполняться до прерывания сигналом SIGINT (**Control**+**C**) или SIGTERM.

Когда утилита pg_recvlogical получает сигнал SIGHUP, она закрывает текущий выходной файл и открывает новый с именем файла, переданным с параметром `--file`. Таким образом обеспечивается ротация выходного файла сначала за счёт переименовывания текущего файла, а затем за счёт отправки сигнала SIGHUP утилите pg_recvlogical.

Параметры

Для выбора действия необходимо указать минимум один из этих параметров:

`--create-slot`

Создать новый слот логической репликации с именем, заданным аргументом `--slot`, используя модуль вывода, заданный аргументом `--plugin`, для базы данных, указанной в `--dbname`.

Вы можете использовать параметр `--two-phase` с `--create-slot`, чтобы включить декодирование подготовленных транзакций.

`--drop-slot`

Удалить слот репликации с именем, заданным аргументом `--slot`, и завершиться.

`--start`

Начать приём потока изменений из слота логической репликации с именем, заданным аргументом `--slot`, и продолжать до сигнала прерывания. Если передача потока прерывается на другой стороне из-за выключения или остановки сервера, цикл подключения и передачи повторяется (если не добавлен параметр `--no-loop`).

Формат потока определяется модулем вывода, выбранным при создании слота.

Для получения потока подключаться нужно к той же базе, для которой создавался слот.

Параметры `--create-slot` и `--start` исключают друг друга. Действие `--drop-slot` несовместимо с любыми другими действиями.

Следующие параметры командной строки управляют расположением и форматом выводимых данных, а также другим поведением репликации:

`-E` *`lsn`*
`--endpos=`*`lsn`*

В режиме `--start` автоматически закончить репликацию и выйти с кодом обычного завершения 0, когда при приёме данных достигается указанный LSN. Если этот ключ указывается не в режиме `--start`, выдаётся ошибка.

Если встречается запись с LSN, в точности равным *`lsn`*, эта запись будет выведена.

С указанием `--endpos` границы транзакций не отслеживаются, так что вывод программы может оказаться обрезанным посередине транзакции. Частично полученная транзакция не будет считаться принятой и будет воспроизведена заново при следующем чтении из этого слота. Отдельные сообщения не обрезаются никогда.

`-f` *`имя_файла`*
`--file=`*`имя_файла`*

Записывать полученные и декодированные данные транзакций в указанный файл. Для вывода в stdout укажите `-` (минус).

`-F` *`секунды`*
`--fsync-interval=`*`секунды`*

Устанавливает, как часто pg_recvlogical будет вызывать `fsync()`, чтобы гарантировать, что выходной файл надёжно сохранён на диске.

Сервер время от времени даёт клиенту команду сохранить данные и сообщить сохранённую позицию, но этот параметр позволяет выполнять сохранение чаще.

При значении, равном `0`, функция `fsync()` вообще не вызывается, но серверу сообщается новая позиция. Это может привести к потере данных в случае сбоя.

`-I` *`lsn`*
`--startpos=`*`lsn`*

В режиме `--start` репликация начнётся с данного LSN. Как это работает, подробно описывается в Главе 47 и Разделе 53.4. В других режимах игнорируется.

`--if-not-exists`

Не выдавать ошибку, когда указан параметр `--create-slot` и слот с заданным именем уже существует.

`-n`
`--no-loop`

Когда подключение к серверу потеряно, не повторять цикл, просто завершить работу.

`-o` *`имя`*`[=`*`значение`*`]`
`--option=`*`имя`*`[=`*`значение`*`]`

Передаёт параметр *`имя_параметра`* модулю вывода, при этом может быть передано и его *`значение`*. Набор параметров и их действия зависят от выбранного модуля вывода.

`-P` *`модуль`*
`--plugin=`*`модуль`*

Использовать указанный модуль вывода логического декодирования при создании слота. См. Главу 47. Этот параметр не действует, если слот уже существует.

`-s` *`секунды`*
`--status-interval=`*`секунды`*

Этот параметр действует так же, как одноимённый параметр pg_receivewal (см. его описание там).

`-S` *`имя_слота`*
`--slot=`*`имя_слота`*

Этот параметр задаёт имя слота логической репликации, который будет использоваться в режиме `--start`, создаваться в режиме `--create-slot` или удаляться в режиме `--drop-slot`.

`-t`
`--two-phase`

Включает декодирование подготовленных транзакций. Этот параметр можно указать только с `--create-slot`.

`-v`
`--verbose`

Включает режим подробных сообщений.

Далее описаны параметры управления подключением.

`-d` *`имя_бд`*
`--dbname=`*`имя_бд`*

Имя базы данных для подключения. Как именно используется данная база, рассказывается в описании действий программы. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. По умолчанию в качестве имени базы выбирается имя пользователя.

`-h` *`имя_компьютера-или-ip`*
`--host=`*`имя_компьютера-или-ip`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *`user`*
`--username=`*`user`*

Имя пользователя для подключения. По умолчанию это имя текущего пользователя операционной системы.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_recvlogical запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_recvlogical лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Также есть следующие дополнительные параметры:

`-V`
`--version`

Сообщить версию pg_recvlogical и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки pg_recvlogical и завершиться.

Код завершения

pg_recvlogical завершится с кодом 0 при прерывании сигналом SIGINT или SIGTERM. (Это штатный способ его завершения, поэтому получение этого сигнала не считается ошибкой.) При критических ошибках или получении других сигналов код завершения будет ненулевым.

Переменные окружения

Как и большинство других утилит Postgres Pro, приложение также использует переменные окружения, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

pg_recvlogical сохранит разрешения для группы в полученных файлах WAL, если такие разрешения установлены в исходном кластере.

Примеры

Примеры использования можно найти в Разделе 47.1.

См. также

pg_receivewal

pg_restore

pg_restore — восстановить базу данных Postgres Pro из файла архива, созданного командой pg_dump

Синтаксис

`pg_restore` [*параметр-подключения*...] [*параметр*...] [*имя_файла*]

Описание

Утилита pg_restore предназначена для восстановления базы данных Postgres Pro из архива, созданного командой pg_dump в любом из не текстовых форматов. Она выполняет команды, необходимые для восстановления того состояния базы данных, в котором база была сохранена. При наличии файлов архивов, pg_restore может восстанавливать данные избирательно или даже переупорядочить объекты перед восстановлением. Заметьте, что разработанный для файлов архива формат не привязан к архитектуре.

Утилита pg_restore может работать в двух режимах. Если указывается имя базы данных, pg_restore подключается к этой базе данных и восстанавливает содержимое архива непосредственно в неё. В противном случае создаётся SQL-скрипт с командами, необходимыми для пересоздания базы данных, который затем выводится в файл или в стандартное устройство вывода. Сформированный скрипт будет в точности соответствовать выводу pg_dump в простом текстовом формате. Поэтому некоторые из параметров, управляющих выводом, аналогичны параметрам pg_dump.

Разумеется, pg_restore может восстановить информацию, только если она присутствует в файле архива, и только в существующем виде. Например, если архив был создан с указанием «выгрузить данные в виде команд `INSERT`», pg_restore не сможет загрузить эти данные, используя операторы `COPY`.

Предупреждение

При восстановлении выгрузки в целевой точке выполняется произвольный код по выбору суперпользователей исходной точки. Для частичных выгрузок и частичных восстановлений такого ограничения нет. Если суперпользователи исходной точки не являются доверенными, выгружаемые команды SQL нужно проверять перед восстановлением. Выгрузки в нетекстовом формате можно проверить через параметр `--file` pg_restore. Обратите внимание, что клиенту, выполняющий выгрузку и восстановление, не нужно доверять суперпользователям исходной или целевой точки.

Параметры

Утилита pg_restore принимает следующие аргументы командной строки.

имя_файла

Указывает расположение восстанавливаемого файла архива (или каталога, для архива в формате каталога). По умолчанию используется устройство стандартного ввода.

`-a`
`--data-only`

Восстанавливать только данные, без схемы (определений данных). При этом восстанавливаются данные таблиц, большие объекты и значения последовательностей, имеющиеся в архиве.

Флаг похож на `--section=data`, но по историческим причинам не равнозначен ему.

`-c`
`--clean`

Прежде чем восстанавливать объекты базы данных, выполните команды `DROP` для всех восстанавливаемых объектов. Этот параметр полезен для перезаписи существующей базы данных. Без дополнительного указания `--if-exists` при этом могут выдаваться безвредные сообщения об ошибках, если таких объектов не окажется в целевой базе данных.

`-C`
`--create`

Создать базу данных, прежде чем восстанавливать данные. Если также указан параметр `--clean`, удалить и пересоздать целевую базу данных перед подключением к ней.

С ключом `--create` программа pg_restore также восстанавливает комментарий к базе данных (если он задан) и все назначения переменных конфигурации, связанные с базой данных, то есть все команды `ALTER DATABASE ... SET ...` и `ALTER ROLE ... IN DATABASE ... SET ...`, ссылающиеся на эту базу данных. Также восстанавливаются права доступа к самой базе данных, если не добавлен ключ `--no-acl`.

С этим параметром база, заданная параметром `-d`, применяется только для подключения и выполнения начальных команд `DROP DATABASE` и `CREATE DATABASE`. Все данные восстанавливаются в базу данных, имя которой записано в архиве.

`-d` *`имя_бд`*
`--dbname=`*`имя_бд`*

Подключиться к базе данных *`имя_базы`* и восстановить данные непосредственно в неё. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--exit-on-error`

Завершать работу в случае возникновения ошибки при выполнении команд SQL в базе данных. По умолчанию процесс восстановления продолжается, а по его окончании выдаётся число ошибок.

`-f` *`имя_файла`*
`--file=`*`имя_файла`*

Задаёт файл для вывода сгенерированного скрипта или списка объектов, получаемого с параметром `-l`. Чтобы выбрать stdout, используйте `-`.

`--filter=`*`имя_файла`*

Задаёт имя файла, из которого будут считываться шаблоны включения или исключения объектов при восстановлении. Шаблоны интерпретируются по тем же правилам, что и параметры `-n`/`--schema` для включения объектов в схемах, `-N`/`--exclude-schema` для исключения объектов в схемах, `-P`/`--function` для восстановления указанных функций, `-I`/`--index` для восстановления указанных индексов, `-t`/`--table` для восстановления указанных таблиц или `-T`/`--trigger` для восстановления триггеров. Для чтения из `STDIN` используйте `-` в качестве имени файла. Параметр `--filter` можно указывать в сочетании с вышеуказанными параметрами для включения/исключения объектов, а также использовать несколько раз, если нужно задать несколько файлов с фильтрами.

Файл содержит по одной строке на каждый шаблон базы данных в следующем формате:

```
{ include | exclude } { function | index | schema | table | trigger } ШАБЛОН
```

Первое ключевое слово указывает, будут ли объекты, соответствующие шаблону, включены или исключены. Второе ключевое слово указывает тип объектов, фильтруемых с помощью шаблона:

- `function`: функции; работает как параметр `-P`/`--function`. Это ключевое слово может использоваться только с ключевым словом `include`.
- `index`: индексы; работает как параметр `-I`/`--indexes`. Это ключевое слово может использоваться только с ключевым словом `include`.
- `schema`: схемы; работает как параметры `-n`/`--schema` и `-N`/`--exclude-schema`.
- `table`: таблицы; работает как параметр `-t`/`--table`. Это ключевое слово может использоваться только с ключевым словом `include`.
- `trigger`: триггеры; работает как `-T`/`--trigger`. Это ключевое слово может использоваться только с ключевым словом `include`.

Строки, начинающиеся с `#`, считаются комментариями и игнорируются. Комментарии также могут размещаться после строки с шаблоном объекта. Пустые строки также игнорируются. См. раздел Шаблоны поиска для информации о том, как применять экранирование в шаблонах.

`-F` *`формат`*
`--format=`*`формат`*

Задаёт формат архива. Указывать формат необязательно, так как pg_restore определяет формат автоматически. Но если формат задаётся, допускается один из этих вариантов:

`c`
`custom`

Архив сохранён в специальном формате pg_dump.

`d`
`directory`

Архив сохранён в каталоге.

`t`
`tar`

Архив сохранён в формате `tar`.

`-I` *`индекс`*
`--index=`*`индекс`*

Восстановить определение только заданного индекса. Добавив дополнительные ключи `-I`, можно указать несколько индексов.

`-j` *`число-заданий`*
`--jobs=`*`число-заданий`*

Выполнять наиболее длительные этапы pg_restore (в частности, загрузку данных, создание индексов или ограничений) параллельно, используя несколько заданий (в количестве, не превышающем *`число-заданий`*). Это позволяет кардинально сократить время восстановления большой базы данных, когда сервер работает на многопроцессорном компьютере. Данный параметр игнорируется, когда генерируется скрипт (нет прямого подключения к базе данных).

Каждое задание выполняется в отдельном задании или потоке, в зависимости от операционной системы, и использует отдельное подключение к серверу.

Оптимальное значение этого параметра зависит от аппаратной конфигурации сервера, клиента и сети. В частности, имеет значение количество процессорных ядер и устройство дискового хранилища. В качестве начального значения можно выбрать число ядер на сервере, но и при увеличении этого значения во многих случаях восстановление будет быстрее. Конечно, при слишком больших значениях производительность упадёт из-за перегрузки.

Этот параметр поддерживается только с архивом в специальном формате или в каталоге. Входные данные должны поступать из обычного файла или каталога, а не из канала или стандартного устройства ввода. Кроме того, несколько заданий не могут выполняться в сочетании с параметром `--single-transaction`.

`-l`
`--list`

Вывести оглавление архива. Вывод этой операции можно подать на вход этой же команде с параметром `-L`. Учтите, что когда вместе с `-l` применяются параметры фильтрации (например, `-n` или `-t`), они сокращают список выводимых объектов.

`-L` *`файл-список`*
`--use-list=`*`файл-список`*

Восстановить из архива только элементы, перечисленные в *`файле-списке`*, и в том порядке, в каком они идут в этом файле. Заметьте, что когда вместе с `-L` применяются параметры фильтрации (например, `-n` или `-t`), они дополнительно ограничивают восстанавливаемые объекты.

Данный *`файл-список`* обычно представляет собой отредактированный результат предыдущей операции `-l`. Строки в нём могут быть переставлены или удалены, а также могут быть закомментированы точкой с запятой (`;`), добавленной в начале строки. См. примеры ниже.

`-n` *`схема`*
`--schema=`*`схема`*

Восстановить только объекты в указанной схеме. Добавив дополнительные ключи `-n`, можно указать несколько схем. Этот параметр можно сочетать с `-t`, чтобы восстановить только определённую таблицу.

`-N` *`схема`*
`--exclude-schema=`*`схема`*

Не восстанавливать объекты в указанной схеме. Добавив дополнительные ключи `-N`, можно исключить несколько схем.

Когда и с ключом `-n`, и с ключом `-N` передаётся имя одной схемы, ключ `-N` выигрывает и схема исключается.

`-O`
`--no-owner`

Не генерировать команды, устанавливающие владение объектами, как в исходной базе данных. По умолчанию, pg_restore генерирует команды `ALTER OWNER` или `SET SESSION AUTHORIZATION`, восстанавливающие исходных владельцев создаваемых элементов схемы. Однако эти команды можно будет выполнить, только если к базе данных первоначально подключается суперпользователь (или пользователь, владеющими всеми объектами в скрипте). Чтобы получить скрипт, который сможет восстановить любой подключающийся пользователь (но при этом он станет владельцем всех созданных объектов), используется `-O`.

`-P` *`имя-функции(тип-аргумента[, ...])`*
`--function=`*`имя-функции(тип-аргумента[, ...])`*

Восстановить только указанную функцию. При этом важно записать имя функции и аргументы в точности так, как они фигурируют в оглавлении файла архива. Добавив дополнительные ключи `-P`, можно указать несколько функций.

`-R`
`--no-reconnect`

Параметр является устаревшим, но в целях совместимости ещё работает.

`-s`
`--schema-only`

Восстановить только схему (определения данных), без данных, в объёме, в котором элементы схемы представлены в архиве.

Действие параметра противоположно действию `--data-only`. Это похоже на указание `--section=pre-data --section=post-data`, но по историческим причинам не равнозначно ему.

(Не путайте этот параметр с `--schema`, где слово «схема» используется в другом значении.)

`-S` *`имя_пользователя`*
`--superuser=`*`имя_пользователя`*

Задаёт имя суперпользователя, полномочия которого будут использоваться для отключения триггеров. Этот параметр применяется только с параметром `--disable-triggers`.

`-t` *`таблица`*
`--table=`*`таблица`*

Восстановить определение и/или данные только указанной таблицы. В этом контексте под «таблицей» подразумеваются также представления, материализованные представления, последовательности и сторонние таблицы. Чтобы выбрать несколько таблиц, ключ `-t` можно указать несколько раз. Этот параметр можно скомбинировать с `-n`, чтобы выбрать таблицу(ы) в определённой схеме.

Примечание

Когда указывается `-t`, pg_restore не пытается восстанавливать объекты базы данных, от которых могут зависеть выбранные таблицы. Таким образом, в этом случае не гарантируется, что выгруженные таблицы будут успешно восстановлены в чистой базе данных.

Примечание

Этот флаг действует не совсем так, как флаг `-t` утилиты pg_dump. В настоящее время pg_restore не поддерживает выбор объектов по маске, а также не позволяет указать имя схемы с `-t`. И хотя pg_dump с флагом `-t` также выгружает подчинённые объекты (например, индексы) выбранных таблиц, pg_restore с флагом `-t` такие подчинённые объекты не обрабатывает.

Примечание

В версиях Postgres Pro до 9.6 этот флаг выбирал только таблицы, но не другие типы отношений.

`-T` *`триггер`*
`--trigger=`*`триггер`*

Восстановить только указанный триггер. Добавив дополнительные ключи `-T`, можно указать несколько триггеров.

`-v`
`--verbose`

Включить подробный режим. pg_restore будет выводить в стандартный поток ошибок подробные комментарии к объектам, включая время начала и окончания операции, а также сообщения о прогрессе выполнения. Если повторить этот ключ, в стандартный поток ошибок будут выдаваться дополнительные отладочные сообщения.

`-V`
`--version`

Сообщить версию pg_restore и завершиться.

`-x`
`--no-privileges`
`--no-acl`

Не восстанавливать права доступа (не выполнять команды GRANT/REVOKE).

`-1`
`--single-transaction`

Произвести восстановление в одной транзакции (то есть, завернуть выполняемые команды в `BEGIN`/`COMMIT`). При этом гарантируется, что либо все команды будут выполнены успешно, либо не будет никаких изменений. Этот режим подразумевает `--exit-on-error`.

`--disable-triggers`

Этот параметр действует только при выгрузке одних данных. С ним pg_restore выполняет команды, отключающие триггеры в целевых таблицах на время восстановления данных. Используйте его, если в ваших таблицах определены проверки ссылочной целостности или другие триггеры, которые вы не хотели бы выполнять в процессе восстановления данных.

В настоящее время команды, генерируемые с `--disable-triggers`, должны выполнятся суперпользователем. Поэтому необходимо также задать имя суперпользователя в параметре `-S` или, что предпочтительнее, запускать pg_restore от имени суперпользователя Postgres Pro.

`--enable-row-security`

Этот параметр имеет смысл только при восстановлении содержимого таблицы, для которой включена защита строк. По умолчанию pg_restore устанавливает для row_security значение off для уверенности, что в таблице восстановлены все данные. Если у пользователя недостаточно прав для обхода защиты строк, выдаётся ошибка. Этот параметр указывает pg_restore установить в row_security значение on, чтобы пользователь мог попытаться восстановить содержимое таблицы с включённой защитой строк. Однако и при этом возможна ошибка, если пользователь не будет иметь права добавлять в эту таблицу выгруженные строки данных.

Заметьте, что в настоящее время для этого требуется, чтобы выгрузка выполнялась в режиме `INSERT`, так как `COPY FROM` не поддерживает защиту строк.

`--if-exists`

Использовать команды `DROP ... IF EXISTS` для удаления объектов в режиме `--clean`. При этом возможные ошибки «does not exist» (не существует) не выводятся. Этот параметр недействителен без указания `--clean`.

`--no-comments`

Не выводить команды, восстанавливающие комментарии, даже если они содержатся в архиве.

`--no-data-for-failed-tables`

По умолчанию данные восстанавливаются даже при ошибке команды создания таблицы (например, когда она уже существует). С этим параметром данные в таком случае не восстанавливаются. Это поведение полезно, если в целевой таблице уже содержатся нужные данные. Например, вспомогательные таблицы для расширений Postgres Pro (в частности, PostGIS) могут быть уже заполнены; этот параметр позволяет предотвратить дублирование или загрузку устаревших данных в эти таблицы.

Этот параметр действует только при восстановлении непосредственно в базу данных (не при генерации SQL-скрипта).

`--no-publications`

Не выводить команды, восстанавливающие публикации, даже если они содержатся в архиве.

`--no-security-labels`

Не выводить команды, восстанавливающие метки безопасности, даже если они содержатся в архиве.

`--no-subscriptions`

Не выводить команды, восстанавливающие подписки, даже если они содержатся в архиве.

`--no-table-access-method`

Не выводить команды для выбора табличных методов доступа. При восстановлении все объекты будут создаваться с табличным методом доступа, выбираемым по умолчанию.

`--no-tablespaces`

Не формировать команды для указания табличных пространств. При восстановлении все объекты будут создаваться в табличном пространстве по умолчанию.

`--restrict-key=`*`restrict_key`*

Использовать предоставленную строку в качестве ключа `\restrict` psql в выводе выгрузки. Задать это можно только для вывода скрипта SQL, например, с помощью параметра `--file`. Если ключ ограничения не указан, pg_dumpall сгенерирует случайный ключ. Ключи могут содержать только буквенные и цифровые символы.

Этот параметр в основном предназначен для целей тестирования и других сценариев, требующих повторяющегося создания вывода (например, для сравнения файлов выгрузки). Параметр не рекомендуется использовать в повседневной работе, поскольку, зная принципы его функционирования, злоумышленник может внедрить произвольный код, который будет выполнен на машине при помощи psql и вывода выгрузки.

`--section=`*`имя секции`*

Восстановить только указанный раздел. В качестве имени раздела можно задать `pre-data`, `data` или `post-data`. Указав этот параметр неоднократно, можно выбрать несколько разделов. По умолчанию восстанавливаются все разделы.

Раздел «data» содержит собственно данные таблиц и определения больших объектов. В разделе «post-data» содержатся определения индексов, триггеров, правил и ограничений (кроме отдельно проверяемых). Раздел «pre-data» содержит все остальные определения.

`--strict-names`

Требует, чтобы каждому указанию схемы (`-n`/`--schema`) и таблицы (`-t`/`--table`) соответствовала минимум одна схема/таблица в файле, который будет восстановлен.

`--transaction-size=`*`N`*

Выполнить восстановление как серию транзакций, каждая из которых обрабатывает до *`N`* объектов базы данных. Этот параметр подразумевает использование `--exit-on-error`.

Параметр `--transaction-size` предлагает промежуточный вариант между поведением по умолчанию (одна транзакция на каждую SQL-команду) и параметром `-1`/`--single-transaction` (одна транзакция для всех восстанавливаемых объектов). Хотя `--single-transaction` минимизирует издержки, для больших баз данных этот вариант не всегда подойдёт, так как транзакция будет блокировать каждый восстанавливаемый объект, что приведёт к нехватке места в таблице блокировок сервера. Использование `--transaction-size` с размером в несколько тысяч объектов обеспечивает почти такие же преимущества в производительности, но с меньшими требованиями к пространству таблицы блокировок.

`--use-set-session-authorization`

Выводить команды `SET SESSION AUTHORIZATION`, соответствующие стандарту, вместо `ALTER OWNER`, для назначения владельцев объектов. В результате выгруженный скрипт будет более стандартизированным, но может не восстановиться корректно, в зависимости от истории объектов.

`-?`
`--help`

Показать справку по аргументам командной строки pg_restore и завершиться.

pg_restore также принимает в качестве параметров соединения следующие аргументы командной строки:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета. Значение по умолчанию берётся из переменной окружения `PGHOST`, если она установлена. В противном случае выполняется подключение к Unix-сокету.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной окружения `PGPORT`, если она установлена, либо числом, заданным при компиляции.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_restore запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_restore лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--role=`*имя роли*

Задаёт имя роли, которая будет осуществлять восстановление. Получив это имя, pg_restore выполнит `SET ROLE` *имя_роли* после подключения к базе данных. Это полезно, когда проходящий проверку пользователь (указанный в `-U`) не имеет прав, необходимых для pg_restore, но может переключиться на роль, наделённую этими правами. В некоторых окружениях правила запрещают подключаться к серверу непосредственно суперпользователю, и этот параметр позволяет выполнить восстановление, не нарушая их.

Переменные окружения

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Как и большинство других утилит Postgres Pro, эта утилита также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15). Однако она не учитывает `PGDATABASE`, когда имя базы не указано.

Диагностика

Когда с параметром `-d` устанавливается прямое подключение к базе данных, pg_restore выполняет обычные операторы SQL. При этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека libpq. Если вы сталкиваетесь с проблемами при запуске pg_restore, убедитесь в том, что вы можете получить информацию из базы данных, используя, например psql.

Примечания

Если в вашей инсталляции база данных `template1` содержит какие-либо дополнения, важно убедиться в том, что вывод pg_restore загружается в действительно пустую базу; иначе вы, скорее всего, получите ошибки из-за дублирования определений создаваемых объектов. Чтобы получить пустую базу данных без дополнительных объектов, выберите в качестве шаблона `template0`, а не `template1`, например так:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Ограничения pg_restore описаны ниже.

- При восстановлении данных в уже существующие таблицы с параметром `--disable-triggers`, pg_restore выполняет команды, отключающие триггеры в пользовательских таблицах до добавления данных, а затем, после добавления данных, выполняет команды, снова включающие эти триггеры. Если восстановление прервётся в середине, системные каталоги могут оказаться в некорректном состоянии.
- Утилита pg_restore не способна восстанавливать большие объекты избирательно; например, только для определённой таблицы. Если архив содержит большие объекты, они будут восстановлены все, либо не будут восстановлены никакие (если они были исключены параметрами `-L`, `-t` и т. п.).

Также обратитесь к документации pg_dump, чтобы узнать о связанных ограничениях pg_dump.

После восстановления имеет смысл запустить `ANALYZE` для каждой восстановленной таблицы, чтобы оптимизатор получил актуальную статистику; за дополнительными сведениями обратитесь к Подразделу 23.1.3 и Подразделу 23.1.6.

Примеры

Предположим, что мы выгрузили базу данных `mydb` в файл специального формата:

```
$ pg_dump -Fc mydb > db.dump
```

Мы можем удалить базу данных и восстановить её из копии:

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

В аргументе `-d` можно указать любую базу данных, существующую в кластере; pg_restore использует её, только чтобы выполнить команду `CREATE DATABASE` для базы `mydb`. С параметром `-C` данные всегда восстанавливаются в базу, имя которой записано в файле архива.

Восстановить данные в новую базу `newdb` можно так:

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

Обратите внимание, мы не используем параметр `-C`, а вместо этого подключаемся непосредственно к базе, в которую хотим восстановить данные. Также заметьте, что мы создаём базу данных из шаблона `template0`, а не `template1`, чтобы изначально она была гарантированно пустой.

Чтобы переупорядочить элементы базы данных, сначала необходимо получить оглавление архива:

```
$ pg_restore -l db.dump > db.list
```

Файл оглавления содержит заголовок и по одной строке для каждого элемента, например:

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

С точки с запятой начинаются комментарии, а число в начале строки обозначает внутренний идентификатор, назначаемый каждому элементу в архиве.

Строки в этом файле можно закомментировать, удалить или переместить. Например, список:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

можно передать утилите pg_restore, чтобы восстановить только элементы 10 и 6 в указанном порядке:

```
$ pg_restore -L db.list db.dump
```

См. также

pg_dump, pg_dumpall, psql

pg-wrapper

pg-wrapper — управлять символическими ссылками Postgres Pro

Синтаксис

`pg-wrapper links { update | remove }`

Описание

Программа pg-wrapper представляет собой скрипт оболочки, включённый в дистрибутив Postgres Pro и предназначенный для управления символическими ссылками на программы Postgres Pro и подключения страниц man по SQL в системах Linux. Так как Postgres Pro устанавливается в `/opt/pgpro/std-17`, это необходимо, чтобы клиентские и серверные программы находились в стандартном системном пути (`PATH`), а также находились новые страницы man по SQL. Эти действия не производятся автоматически во время установки (исключение составляет пакет быстрой установки `postgrespro-std-17`) во избежание возможных конфликтов с другими продуктами на базе PostgreSQL, которые могут уже быть установлены.

Скрипт pg-wrapper входит в состав пакета `postgrespro-std-17-client`. Установив этот пакет, вы сможете найти pg-wrapper в каталоге *`каталог-инсталляции`*`/bin`, где *`каталог-инсталляции`* обозначает путь `/opt/pgpro/std-17`.

Скрипт pg-wrapper должен запускать пользователь root.

Параметры

pg-wrapper принимает следующие аргументы командной строки:

`links {update | remove}`

Управление символическими ссылками Postgres Pro и страницами man по SQL:

- `update` — создать или заменить символические ссылки для серверных или клиентских программ, распространяемых в составе Postgres Pro, а также подключить страницы man по SQL в файле конфигурации страниц man.
- `remove` — удалить символические ссылки и поддержку страниц man по SQL для текущей версии Postgres Pro.

Замечания

Если вы устанавливаете Postgres Pro из пакета `postgrespro-std-17`, pg-wrapper вызывается автоматически.

Если вы устанавливаете отдельные пакеты Postgres Pro, вы можете запустить этот скрипт вручную, чтобы создать символические ссылки на предоставляемые клиентские и серверные программы, а также подключить страницы man по SQL в файле конфигурации страниц man.

При параллельной установке с другими продуктами на базе `PostgreSQL` поведение `pg-wrapper` зависит от того, поддерживается ли механизм update-alternatives вашей системой и уже установленным продуктом на базе `PostgreSQL`:

- Если поддерживается механизм update-alternatives, `pg-wrapper` добавляет символические ссылки на программы Postgres Pro в систему альтернатив, в соответствии с их приоритетом, а также подключает страницы man по SQL в файл конфигурации страницы man. Информацию об изменении приоритетов в системе альтернатив вы найдёте в руководстве man update-alternatives в вашей системе.

  Если в системе уже установлены страницы man по SQL от другого продукта, от новой инсталляции будут выводиться только новые страницы, а все совпадающие страницы man — от старой. Чтобы новая документация SQL выводилась для всех страниц, вам потребуется изменить системную конфигурацию, например, поменять значение `MANPATH`.

- Если механизм update-alternatives не поддерживается, `pg-wrapper` изменяет системную конфигурацию, только если отсутствуют конфликты с любыми другими установленными программами или страницами man. В случае таких конфликтов pg-wrapper не создаёт и не изменяет ссылки программ и пропускает интеграцию страниц man по SQL. В этом случае вы можете либо продолжать использовать ранее установленные версии программ и страниц man по SQL, либо изменить параметры `PATH` и `MANPATH` вручную.

Установка из двоичных пакетов в Linux подробно описана в Разделе 16.1.

pg_verifybackup

pg_verifybackup — проверить целостность базовой копии кластера Postgres Pro

Синтаксис

`pg_verifybackup` [*параметр*...]

Описание

pg_verifybackup позволяет проверить целостность копии кластера БД, сделанной программой `pg_basebackup`, по манифесту `backup_manifest`, созданному во время копирования. Копия должна быть представлена в формате «plain»; чтобы проверить копию в формате «tar», её нужно сначала разархивировать.

Важно отметить, что процедура контроля, выполняемая программой pg_verifybackup не включает и не может включать в себя все проверки, которые будет выполнять сервер, если его запустить с этой копией. Даже если вы пользуетесь этим средством, вам всё же следует выполнять тестовое восстановление данных, чтобы убедиться в том, что восстановленные базы работают ожидаемым образом и содержат нужные данные. Тем не менее pg_verifybackup может быстро выявить множество распространённых проблем, вызываемых неисправностью хранилища или ошибками пользователя.

Проверка копии выполняется в четыре этапа. На первом этапе `pg_verifybackup` читает файл `backup_manifest`. Если этот файл не существует, неправильно оформлен, не соответствует системному идентификатору `pg_control` в каталоге резервных копий или контрольной сумме, которая в нём содержится, `pg_verifybackup` завершает выполнение с критической ошибкой.

На втором этапе `pg_verifybackup` проверяет, что файлы данных, находящиеся на диске в настоящее время, в точностью совпадают с теми файлами, которые сервер должен был передать, за несколькими исключениями, описанными ниже. При этом выявляются все пропавшие или добавившиеся файлы, кроме некоторых игнорируемых. В частности, на этом этапе не принимается во внимание присутствие, отсутствие или какое-либо изменение файлов `postgresql.auto.conf`, `standby.signal` и `recovery.signal`, так как ожидается, что эти файлы могут создаваться или модифицироваться в процессе создания копии. Также из рассмотрения исключается файл `backup_manifest` в целевом каталоге и всё содержимое каталога `pg_wal`, несмотря на то, что эти файлы не будут описаны в манифесте копии. Проверяются только файлы; наличие или отсутствие каталогов контролируется только косвенно: если какой-либо каталог отсутствует, неизбежно будут отсутствовать и все файлы, которые должны в нём содержаться.

Затем `pg_verifybackup` проверяет контрольные суммы всех файлов, сравнивая их со значениями, указанными в манифесте, и выдаёт ошибки для файлов, у которых вычисленная контрольная сумма не совпадает с сохранённой в манифесте. Это действие не выполняется для файлов, вызвавших ошибки на предыдущем шаге, так как о проблемах с ними уже известно. Для файлов, которые игнорировались на втором этапе, контрольные суммы тоже не проверяются.

На последнем этапе `pg_verifybackup`, используя манифест, проверяет, имеются ли в журнале WAL все записи, необходимые для восстановления, и можно ли их успешно прочитать и разобрать. Файл `backup_manifest` содержит информацию о том, какие записи потребуются, благодаря чему `pg_verifybackup` может вызвать `pg_waldump`, чтобы разобрать и тем самым проверить эти записи журнала. При вызове программы `pg_waldump` передаётся флаг `--quiet`, так что она будет выдавать только ошибки, без каких-либо других сообщений. Хотя этот уровень проверки достаточен для выявления явных проблем, например, отсутствия файлов или несоответствия внутренних контрольных сумм, он всё же не даёт гарантию обнаружения всех проблем, которые могут возникнуть при попытке восстановления базы. Например, данный метод проверки бесполезен, если из-за внутренней ошибки сервера в WAL будут вноситься записи с правильными контрольными суммами, но бессмысленным содержимым.

Заметьте, что в случае наличия дополнительных файлов WAL, не требующихся для восстановления копии, они не будут проверяться этим средством, хотя проверить их можно, вызвав `pg_waldump` отдельно. Также учтите, что проверка WAL зависит от версии: для проверки целостности копии необходимо использовать ту версию `pg_verifybackup`, а значит и `pg_waldump`, с которой эта копия была получена. Проверки же целостности файлов должны работать с данными любой версии сервера, сформировавшего файл `backup_manifest`.

Параметры

Утилита pg_verifybackup принимает следующие аргументы командной строки:

`-e`
`--exit-on-error`

Завершиться при первой же выявленной проблеме. В отсутствие этого указания `pg_verifybackup` продолжает проверку копии после обнаружения первой ошибки и сообщает обо всех ошибках.

`-i` *`путь`*
`--ignore=`*`путь`*

Игнорировать указанный файл или каталог, который может быть задан относительным путём, при сравнении списка файлов данных, фактически присутствующих в копии, со списком в файле `backup_manifest`. Если указан путь к каталогу, из рассмотрения исключается всё дерево подкаталогов, начиная с указанного. В случае совпадения относительного пути файла с указанным никакие сообщения о дополнительных или пропавших файлах, а также об изменении размера или несовпадении контрольных сумм файлов, выдаваться не будут. Этот параметр можно задать несколько раз.

`-m` *`путь`*
`--manifest-path=`*`путь`*

Использовать файл манифеста по заданному пути вместо файла, расположенного в корневом каталоге копии.

`-n`
`--no-parse-wal`

Не пытаться разобрать данные журнала предзаписи, которые могут понадобиться для восстановления проверяемой копии.

`-P`
`--progress`

Включает вывод сообщений о прогрессе. Эти сообщения будут выводиться при проверке контрольных сумм.

Этот параметр нельзя использовать с параметром `--quiet`.

`-q`
`--quiet`

Не выводить ничего, если копия проходит проверку успешно.

`-s`
`--skip-checksums`

Не проверять контрольные суммы файлов данных. При этом тем не менее будет проверяться отсутствие или наличие файлов и их размеры. В таком режиме проверка выполняется гораздо быстрее, так как собственно содержимое файлов читать не требуется.

`-w` *`путь`*
`--wal-directory=`*`путь`*

Проверять файлы WAL, находящиеся в указанном каталоге, а не в `pg_wal`. Это полезно, если архив WAL сохраняется отдельно от основного содержимого копии.

Другие флаги:

`-V`
`--version`

Сообщить версию pg_verifybackup и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_verifybackup и завершиться.

Примеры

Создание базовой копии сервера `mydbserver` и проверка целостности копии:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ pg_verifybackup /usr/local/pgsql/data
```

Создание базовой копии сервера `mydbserver`, перемещение файла манифеста во внешний каталог и проверка копии:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/backup1234
$ mv /usr/local/pgsql/backup1234/backup_manifest /my/secure/location/
backup_manifest.1234
$ pg_verifybackup -m /my/secure/location/backup_manifest.1234 /usr/local/pgsql/
backup1234
```

Проверка копии с исключением файла, добавленного в каталог копии вручную, и без проверки контрольных сумм:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ edit /usr/local/pgsql/data/note.to.self
$ pg_verifybackup --ignore=note.to.self --skip-checksums /usr/local/pgsql/data
```

См. также

pg_basebackup

psql

psql — интерактивный терминал Postgres Pro

Синтаксис

`psql` [*параметр*...] [*имя_бд* [*имя_пользователя*]]

Описание

Программа psql — это терминальный клиент для работы с Postgres Pro. Она позволяет интерактивно вводить запросы, передавать их в Postgres Pro и видеть результаты. Также запросы могут быть получены из файла или из аргументов командной строки. Кроме того, psql предоставляет ряд метакоманд и различные возможности, подобные тем, что имеются у командных оболочек, для облегчения написания скриптов и автоматизации широкого спектра задач.

Параметры

`-a`
`--echo-all`

Отправляет в стандартный вывод все непустые входные строки по мере их чтения. (Это не относится к строкам, считанным в интерактивном режиме.) Эквивалентно установке переменной `ECHO` в значение `all`.

`-A`
`--no-align`

Переключает на невыровненный режим вывода. (По умолчанию используется другой режим, `aligned`.) Равнозначно команде `\pset format unaligned`.

`-b`
`--echo-errors`

Выводит все команды SQL с ошибками в стандартный поток ошибок. Равнозначно присваиванию переменной `ECHO` значения `errors`.

`-c` *команда*
`--command=`*команда*

Передаёт psql *команду* для выполнения. Этот ключ можно повторять и комбинировать в любом порядке с ключом `-f`. Когда указывается `-c` или `-f`, psql не читает команды со стандартного ввода; вместо этого она завершается сразу после обработки всех ключей `-c` и `-f` по порядку.

Заданная *команда* должна быть либо командной строкой, которая полностью интерпретируется сервером (т. е. не использует специфические функции psql), либо одиночной командой с обратной косой чертой. Таким образом, используя `-c`, нельзя смешивать метакоманды SQL и psql. Но это можно сделать, передав несколько ключей `-c` или передав строку в psql через канал:

```
psql -c '\x' -c 'SELECT * FROM foo;'
```

или

```
echo '\x \\ SELECT * FROM foo;' | psql
```

(`\\` — разделитель метакоманд.)

Каждая строка SQL-команд, заданная ключом `-c`, передаётся на сервер как один запрос. Поэтому сервер выполняет её в одной транзакции, даже когда эта строка содержит несколько команд SQL, если только в ней не содержатся явные команды `BEGIN`/`COMMIT`, разделяющие её на несколько транзакций. (Подробнее о том, как сервер обрабатывает строки, включающие несколько команд, рассказывается в Подразделе 53.2.2.1.)

Если вы не хотите, чтобы несколько команд выполнялись в одной транзакции, используйте несколько ключей `-c` или передайте несколько команд на стандартный ввод psql, применяя либо echo, как показано выше, либо создаваемый прямо в оболочке текст, например:

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

`--csv`

Переключает в режим вывода CSV (Comma Separated Values, Значения, разделённые запятыми). Равнозначно команде `\pset format csv`.

`-d` *имя_бд*
`--dbname=`*имя_бд*

Указывает имя базы данных для подключения. Равнозначно указанию *имя_бд* в первом аргументе, не являющемся ключом, в командной строке. Вместо имени может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--echo-queries`

Посылает все команды SQL, отправленные на сервер, ещё и на стандартный вывод. Эквивалентно установке переменной `ECHO` в значение `queries`.

`-E`
`--echo-hidden`

Отображает фактические запросы, генерируемые `\d` и другими командами, начинающимися с \. Это можно использовать для изучения внутренних операций в psql. Эквивалентно установке переменной `ECHO_HIDDEN` значения `on`.

`-f` *имя_файла*
`--file=`*имя_файла*

Читает команды из файла *имя_файла*, а не из стандартного ввода. Этот ключ можно повторять и комбинировать в любом порядке с ключом `-c`. Если указан ключ `-c` или `-f`, программа psql не читает команды со стандартного ввода; вместо этого она завершается после обработки всех ключей `-c` и `-f` по очереди. Не считая этого, данный ключ по большому счёту равнозначен метакоманде `\i`.

Если *имя_файла* задано символом `-` (минус), считывается стандартный ввод до признака конца файла или до метакоманды `\q`. Это позволяет перемежать интерактивный ввод с вводом из файлов. Однако заметьте, что Readline в этом случае не применяется (так же, как и с ключом `-n`).

Использование этого параметра немного отличается от `psql <` *имя_файла*. В основном, оба варианта будут делать то, что вы ожидаете, но с `-f` доступны некоторые полезные свойства, такие как сообщения об ошибках с номерами строк. Также есть небольшая вероятность, что запуск в таком режиме будет быстрее. С другой стороны, вариант с перенаправлением ввода из командного интерпретатора (в теории) гарантирует получение точно такого же вывода, какой вы получили бы, если бы ввели всё вручную.

`-F` *разделитель*
`--field-separator=`*разделитель*

Использование *разделитель* в качестве разделителя полей при невыровненном режиме вывода. Эквивалентно `\pset fieldsep` или `\f`.

`-h` *`компьютер`*
`--host=`*`компьютер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-H`
`--html`

Переключает в режим вывода HTML. Равнозначно команде `\pset format html` или `\H`.

`-l`
`--list`

Выводит список всех доступных баз данных и завершает работу. Другие параметры, не связанные с соединением, игнорируются. Это похоже на метакоманду `\list`.

Когда используется этот аргумент, psql будет подключаться к базе данных `postgres`, если только в командной строке не задана другая база данных (в параметре `-d` или не через параметры, а, например, через запись службы, но не через переменную окружения).

`-L` *`имя_файла`*
`--log-file=`*`имя_файла`*

В дополнение к обычному выводу, записывает вывод результатов всех запросов в файл *`имя_файла`*.

`-n`
`--no-readline`

Отключает использование Readline для редактирования командной строки и использование истории команд (см. Подраздел «Редактирование командной строки» ниже).

`-o` *`имя_файла`*
`--output=`*`имя_файла`*

Записывает вывод результатов всех запросов в файл *`имя_файла`*. Эквивалентно команде `\o`.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения. Значение по умолчанию определяется переменной среды `PGPORT`, если она установлена, либо числом, заданным при компиляции, обычно 5432.

`-P` *`присваивание`*
`--pset=`*`присваивание`*

Задаёт параметры печати, в стиле команды `\pset`. Обратите внимание, что имя параметра и значение разделяются знаком равенства, а не пробела. Например, чтобы установить формат вывода в LaTeX, нужно написать `-P format=latex`.

`-q`
`--quiet`

Указывает, что psql должен работать без вывода дополнительных сообщений. По умолчанию, выводятся приветствия и различные информационные сообщения. Этого не произойдёт с использованием данного параметра. Полезно вместе с параметром `-c`. Этот же эффект можно получить, установив для переменной `QUIET` значение `on`.

`-R` *`разделитель`*
`--record-separator=`*`разделитель`*

Использовать *`разделитель`* как разделитель записей при невыровненном режиме вывода. Равнозначно команде `\pset recordsep`.

`-s`
`--single-step`

Запуск в пошаговом режиме. Это означает, что пользователь будет подтверждать выполнение каждой команды, отправляемой на сервер, с возможностью отменить выполнение. Используется для отладки скриптов.

`-S`
`--single-line`

Запуск в однострочном режиме, при котором символ новой строки завершает SQL-команды, так же как это делает точка с запятой.

Примечание

Этот режим реализован для тех, кому он нужен, но это не обязательно означает, что и вам нужно его использовать. В частности, если смешивать в одной строке команды SQL и метакоманды, порядок их выполнения может быть не всегда понятен для неопытного пользователя.

`-t`
`--tuples-only`

Отключает вывод имён столбцов и результирующей строки с количеством выбранных записей. Равнозначно команде `\t` или `\pset tuples_only`.

`-T` *`параметры_таблицы`*
`--table-attr=`*`параметры_таблицы`*

Задаёт атрибуты, которые будут вставлены в тег HTML `table`. За подробностями обратитесь к описанию `\pset tableattr`.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Использовать для подключения к базе данных *`имя_пользователя`* вместо подразумеваемого по умолчанию. (Разумеется, это потребует соответствующего разрешения.)

`-v` *`присваивание`*
`--set=`*`присваивание`*
`--variable=`*`присваивание`*

Выполняет присваивание значения переменной, как метакоманда `\set`. Обратите внимание, что необходимо разделить имя переменной и значение (при наличии) знаком равенства в командной строке. Чтобы сбросить переменную, оставьте имя переменной без знака равенства. Чтобы установить пустое значение, добавьте знак равенства, но опустите значение. Эти присваивания выполняются во время обработки командной строки, так что переменные, отражающие состояние соединения, будут перезаписаны позже.

`-V`
`--version`

Выводит версию psql и завершает работу.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль нельзя получить из других источников, например из файла `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

Обратите внимание, что этот параметр действует на протяжении всего сеанса и, таким образом, влияет на метакоманду `\connect`, так же как и на первую попытку соединения.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных, даже если он не будет использоваться.

Если сервер требует аутентификацию по паролю и пароль нельзя получить из других источников, например из файла `.pgpass`, psql запросит пароль в любом случае. Однако чтобы понять, что требуется пароль, psql лишний раз подключится к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Обратите внимание, что этот параметр действует на протяжении всего сеанса и, таким образом, влияет на метакоманду `\connect`, так же как и на первую попытку соединения.

`-x`
`--expanded`

Включает режим развёрнутого вывода таблицы. Равнозначно команде `\x` или `\pset expanded`.

`-X`
`--no-psqlrc`

Не читать стартовые файлы (ни общесистемный файл `psqlrc`, ни пользовательский файл `~/.psqlrc`).

`-z`
`--field-separator-zero`

Установить нулевой байт в качестве разделителя полей для невыровненного режима вывода. Равнозначно команде `\pset fieldsep_zero`.

`-0`
`--record-separator-zero`

Установить нулевой байт в качестве разделителя записей для невыровненного режима вывода. Это полезно при взаимодействии с другими программами, например, с `xargs -0`. Равнозначно команде `\pset recordsep_zero`.

`-1`
`--single-transaction`

Этот параметр может применяться только в сочетании с одним или несколькими параметрами `-c` и/или `-f`. С ним psql выполняет команду `BEGIN` перед обработкой первого такого параметра и `COMMIT` после последнего, заворачивая таким образом все команды в одну транзакцию. Если какая-либо из команд завершилась ошибкой и была установлена переменная `ON_ERROR_STOP`, вместо `COMMIT` передаётся команда `ROLLBACK`. Это гарантирует, что либо все команды завершатся успешно, либо никакие изменения не сохранятся.

Если в самих этих командах содержатся операторы `BEGIN`, `COMMIT` или `ROLLBACK`, этот параметр не даст желаемого эффекта. Кроме того, если какая-либо отдельная команда не может выполняться внутри блока транзакции, с этим параметром вся транзакция прервётся с ошибкой.

`-?`
`--help[=`*`тема`*`]`

Показать справку по psql и завершиться. Необязательный параметр *`тема`* (по умолчанию `options`) выбирает описание интересующей части psql: `commands` описывает команды psql с обратной косой чертой; `options` описывает параметры командной строки, которые можно передать psql; а `variables` выдаёт справку по переменным конфигурации psql.

Код завершения

При нормальном завершении psql возвращает 0 в командную оболочку ОС, 1 — если произошла критическая ошибка в самом psql (например, нехватка памяти, файл не найден), 2 — при неудачном соединении с сервером неинтерактивного сеанса, 3 — при ошибке в скрипте и установленной переменной `ON_ERROR_STOP`.

Использование

Подключение к базе данных

psql это клиент для Postgres Pro. Для подключения к базе данных нужно знать имя базы данных, имя сервера, номер порта сервера и имя пользователя БД, под которым вы хотите подключиться. Эти свойства можно задать через аргументы командной строки, а именно `-d`, `-h`, `-p` и `-U` соответственно. Если в командной строке есть аргумент, который не относится к параметрам psql, то он используется в качестве имени базы данных (или имени пользователя БД, если база данных уже задана). Задавать все эти аргументы необязательно, у них есть разумные значения по умолчанию. Если опустить имя сервера, psql будет подключаться через Unix-сокет к локальному серверу, либо подключаться к `localhost` по TCP/IP в Windows. Номер порта по умолчанию определяется во время компиляции. Поскольку сервер базы данных использует то же значение по умолчанию, чаще всего указывать номер порта не нужно. Имя пользователя по умолчанию совпадает с именем пользователя в операционной системе. После определения имени пользователя БД оно используется как имя базы данных по умолчанию. Заметьте, что просто так подключаться к любой базе данных под любым именем пользователя БД вы не сможете. Узнать о ваших правах можно у администратора баз данных.

Если значения по умолчанию не подходят, можно сэкономить на вводе параметров подключения, установив переменные среды `PGDATABASE`, `PGHOST`, `PGPORT` и/или `PGUSER`. (Другие переменные среды описаны в Разделе 32.15.) Также удобно иметь файл `~/.pgpass`, чтобы не вводить пароли снова и снова. За дополнительными сведениями обратитесь к Разделу 32.16.

Альтернативный вариант указания параметров подключения — использование строки *`conninfo`* или URI вместо имени базы данных. Этот механизм даёт широкие возможности для управления соединением. Например:

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

Этот способ также позволяет использовать LDAP для получения параметров подключения, как описано в Разделе 32.18. Более полно все имеющиеся параметры соединения описаны в Подразделе 32.1.2.

Если соединение не может быть установлено по любой причине (например, нет прав, сервер не работает и т. д.), psql вернёт ошибку и прекратит работу.

Если и стандартный ввод, и стандартный вывод являются терминалом, то psql установит кодировку клиента в «auto», и подходящая клиентская кодировка будет определяться из локальных установок (переменная окружения `LC_CTYPE` в Unix). Если это работает не так, как ожидалось, кодировку клиента можно изменить, установив переменную окружения `PGCLIENTENCODING`.

Ввод SQL-команд

Как правило, приглашение psql состоит из имени базы данных, к которой psql в данный момент подключён, а затем строки `=>`. Например:

```
$ psql testdb
psql (17.6.1)
Type "help" for help.

testdb=>
```

В командной строке пользователь может вводить команды SQL. Обычно введённые строки отправляются на сервер, когда встречается точка с запятой, завершающая команду. Конец строки не завершает команду. Это позволяет разбивать команды на несколько строк для лучшего понимания. Если команда была отправлена и выполнена без ошибок, то результат команды выводится на экран.

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления доступных им для записи схем из пути поиска (`search_path`). Для этого можно добавить `options=-csearch_path=` в строку подключения или выполнить команду `SELECT pg_catalog.set_config('search_path', '', false)` перед другими командами SQL. Это касается не только psql, но и любых других интерфейсов для выполнения произвольных SQL-команд.

При каждом выполнении команды psql также проверяет асинхронные уведомления о событиях, генерируемые командами `LISTEN` и `NOTIFY`.

Комментарии в стиле C передаются для обработки на сервер, в то время как комментарии в стандарте SQL psql удаляет перед отправкой.

Метакоманды

Всё, что вводится в psql не взятое в кавычки и начинающееся с обратной косой черты, является метакомандой psql и обрабатывается самим psql. Эти команды делают psql полезным для задач администрирования и разработки скриптов.

Формат команды psql следующий: обратная косая черта, сразу за ней команда, затем аргументы. Аргументы отделяются от команды и друг от друга любым количеством пробелов.

Чтобы включить пробел в значение аргумента, нужно заключить его в одинарные кавычки. Чтобы включить одинарную кавычку в значение аргумента, нужно написать две одинарные кавычки внутри текста в одинарных кавычках. Всё, что содержится в одинарных кавычках подлежит заменам, принятым в языке C: `\n` (новая строка), `\t` (табуляция), `\b` (backspace), `\r` (возврат каретки), `\f` (подача страницы), `\цифры` (восьмеричное число), и `\xцифры` (шестнадцатеричное число). Если внутри текста в одинарных кавычках встречается обратная косая перед любым другим символом, то она экранирует этот символ.

Если внутри аргумента не в кавычках встречается имя переменной psql с предшествующим двоеточием (`:`), оно заменяется значением переменной, как описано в разделе Интерполяция SQL ниже. Также будут работать описанные там формы `:'имя_переменной'` и `:"имя_переменной"`. Конструкция `:{?имя_переменной}` позволяет проверить, определена ли переменная. Она заменяется значением TRUE или FALSE. Экранирование обратной косой чертой защищает двоеточие от замены.

Текст аргумента, заключённый в обратные кавычки (`` ` ``), считается командной строкой, которая передаётся в командную оболочку ОС. Вывод от этой команды (с удалёнными в конце символами новой строки) заменяет текст в обратных кавычках. В содержимом этого текста не обрабатываются никакие спецпоследовательности или особые знаки, за исключением того, что все вхождения `:имя_переменной`, где *имя_переменной* — это имя переменной psql, заменяются значением этой переменной. Также вхождения `:'имя_переменной'` заменяются значением переменной, заключённым в апострофы с тем, что это было одним аргументом команды оболочки. (Последняя форма почти всегда более предпочтительна, если только вы не абсолютно точно знаете, чего ожидать в переменной.) Так как символы перевода строки и возврата каретки могут быть надёжно экранированы не на всех платформах, форма `:'имя_переменной'` выводит сообщение об ошибке и подстановка значения переменной не производится, когда это значение содержит такие символы.

Некоторые команды принимают идентификатор SQL (например, имя таблицы) в качестве аргумента. Такие аргументы следуют правилам синтаксиса SQL: буквы, не взятые в кавычки, преобразуются в нижний регистр, буквы, взятые в двойные кавычки (`"`) предотвращают преобразование регистра и позволяют включать пробелы в идентификатор. Внутри двойных кавычек две двойные кавычки сокращаются до одной. Например, `FOO"BAR"BAZ` интерпретируется как `fooBARbaz`, а `"A weird"" name"` становится `A weird" name`.

Разбор аргументов останавливается в конце строки или когда встречается другая обратная косая черта, не внутри кавычек. Обратная косая не внутри кавычек рассматривается как начало новой метакоманды. Специальная последовательность `\\` (две обратных косых черты) обозначает окончание аргументов, далее продолжается разбор команд SQL, если таковые имеются. Таким образом, команды SQL и psql можно свободно смешивать в одной строке. Но в любом случае аргументы метакоманды не могут выходить за пределы текущей строки.

Многие из метакоманд оперируют с *буфером текущего запроса*. Этот буфер содержит произвольный текст команд SQL, который был введён, но ещё не отправлен серверу для выполнения. В него будут входить и предыдущие строки, а также текст, расположенный в строке метакоманды перед ней.

Определены следующие метакоманды:

`\a`

Если текущий режим вывода таблицы невыровненный, то он переключается на выровненный режим. Если текущий режим выровненный, то устанавливается невыровненный. Эта команда поддерживается для обратной совместимости. См. `\pset` для более общего решения.

`\bind` [*параметр*] ...

Задаёт значения параметров для следующего выполнения запроса, передаваемые в качестве местозаполнителей (`$1` и т. д.).

Пример:

```
INSERT INTO tbl1 VALUES ($1, $2) \bind 'first value' 'second value' \g
```

Эта метакоманда работает не только с `\g`, но и с другими командами выполнения запросов, например `\gx` и `\gset`.

С этой командой используется не простой протокол запросов, как в ходе обычной работы psql, а расширенный протокол (см. Подраздел 53.1.2). Поэтому эту команду можно использовать для тестирования расширенного протокола запросов из psql. (Расширенный протокол используется даже для запросов без параметров, и когда в этой команде параметры не передаются.) Команда влияет только на следующий выполняемый запрос, а дальнейшие запросы по умолчанию будут передаваться по простому протоколу.

`\c` или `\connect [ -reuse-previous=`*on|off*` ] [ `*имя_бд*` [ `*имя_пользователя*` ] [ `*компьютер*` ] [ `*порт*` ] | `*строка_подключения*` ]`

Устанавливает новое подключение к серверу Postgres Pro. Параметры подключения можно указывать как позиционно (один или несколько по списку: база данных, пользователь, компьютер и порт), так и передавая аргумент *строка_подключения* (подробнее о строках подключения рассказывается в Подразделе 32.1.1). Если аргументы отсутствуют, новое подключение устанавливается с теми же параметрами, что и предыдущее.

Указание в параметре *имя_бд*, *имя_пользователя*, *компьютер* или *порт* значения `-` равносильно опущению этого параметра.

Новое подключение может повторно использовать параметры предыдущего — не только имя базы данных, пользователя, компьютер и порт, но и, например, *режим_ssl*. По умолчанию параметры используются повторно при позиционной записи, но не когда задана *строка_подключения*. Это поведение может переопределяться первым аргументом, `-reuse-previous=on` или `-reuse-previous=off`. В случае повторного использования параметров любой параметр, явно не заданный в виде позиционного или в *строке_подключения*, заимствуется из параметров текущего подключения. Исключение составляет параметр *hostaddr* — если он был задан в предыдущем подключении, а в новом в позиционной записи задаётся другое значение *host*, значение *hostaddr* сбрасывается. Кроме того, пароль существующего подключения будет использоваться повторно, только если имя пользователя, узел и порт не меняются. Если какой-либо параметр не указан в этой команде и не используется повторно, действует принятое в libpq значение по умолчанию.

Если новое подключение успешно установлено, предыдущее подключение закрывается. Если попытка подключения не удалась (неверное имя пользователя, доступ запрещён и т. д.), то предыдущее соединение останется активным, если psql находится в интерактивном режиме. Но если скрипт выполняется неинтерактивно, предыдущее соединение закрывается и выдаётся ошибка. При этом скрипт может завершиться или не завершиться; в последнем случае все команды, обращающиеся к базе данных, будут неуспешными, пока не будет успешно выполнена ещё одна команда `\connect`. Различное поведение выбрано для удобства пользователя в качестве защиты от опечаток с одной стороны и в качестве меры безопасности, не позволяющей случайно запустить скрипты в неправильной базе, с другой. Заметьте, что когда команда `\connect` пытается повторно использовать параметры, она использует параметры последнего удачного подключения, а не какой-либо из последующих неудачных попыток. Однако в случае неудачи при неинтерактивном выполнении команды `\connect` никакие параметры не будут использоваться впоследствии, так как скрипт, скорее всего, рассчитывает на то, что должны повторно использоваться параметры именно этой команды `\connect`, а она не была успешной.

Примеры:

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10 sslmode=disable"
=> \c -reuse-previous=on sslmode=require    -- меняется только sslmode
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

`\C [ `*`заголовок`*` ]`

Задаёт заголовок, который будет выводиться для результатов любых запросов или отменяет установленный ранее заголовок. Эта команда эквивалентна `\pset title` *`заголовок`*. (Название этой команды происходит от «caption» (заголовок), так как изначально она применялась только для задания заголовков HTML-таблиц.)

`\cd [ `*`каталог`*` ]`

Сменяет текущий рабочий каталог на *`каталог`*. Без аргумента устанавливается домашний каталог текущего пользователя.

Подсказка

для печати текущего рабочего каталога используйте `\! pwd`.

`\conninfo`

Выводит информацию о текущем подключении к базе данных.

`\copy {` *`таблица`* `[ (` *`список_столбцов`* `) ] } from {` *`'имя_файла'`* `| program` *`'команда'`* `| stdin | pstdin } [ [ with ] (` *`параметр`* `[, ...] ) ] [ where` *`условие`* `]`
`\copy {` *`таблица`* `[ (` *`список_столбцов`* `) ] | (` *`query`* `) } to {` *`'имя_файла'`* `| program` *`'команда'`* `| stdout | pstdout } [ [ with ] (` *`параметр`* `[, ...] ) ]`

Производит копирование данных с участием клиента. При этом выполняется SQL-команда `COPY`, но вместо чтения или записи в файл на сервере, psql читает или записывает файл и пересылает данные между сервером и локальной файловой системой. Это означает, что для доступа к файлам используются привилегии локального пользователя, а не сервера, и не требуются привилегии суперпользователя SQL.

С указанием `program` psql выполняет *`команду`* и данные, поступающие из/в неё, передаются между сервером и клиентом. Это опять же означает, что для выполнения программ используются привилегии локального пользователя, а не сервера, и не требуются привилегии суперпользователя SQL.

При выполнении `\copy ... from stdin` строки с данными считываются из источника, выполнившего команду, и считываются до тех пор, пока не встретится `\.` или не будет достигнут конец файла. Этот параметр полезен для заполнения таблиц прямо в SQL-скриптах. При выполнении `\copy ... to stdout` вывод направляется в то же место, что и вывод psql команд. Статус команды `COPY *count*` не отображается, чтобы не перепутать его со строкой данных. Для чтения/записи стандартного ввода-вывода psql, вне зависимости от источника текущей команды или параметра `\o`, используйте `from pstdin` или `to pstdout`.

Синтаксис команды похож на синтаксис SQL-команды `COPY`. Все параметры, кроме источника и получателя данных, задаются так же, как и в `COPY`. Поэтому при обработке метакоманды `\copy` применяются другие правила разбора. В отличие от большинства других метакоманд, для неё остаток строки всегда воспринимается как аргументы `\copy`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

Подсказка

Альтернативный способ получить тот же результат, что и с `\copy ... to` — использовать SQL-команду `COPY ... TO STDOUT` и завершить её командой `\g *имя*` или `\g |*программа*`. В отличие от `\copy`, этот подход позволяет разбивать команду на несколько строк, а также использовать интерполяцию переменных и раскрытие обратных кавычек (`` ` ``).

Подсказка

Эти операции не так эффективны, как SQL-команда `COPY`, в которой источником или получателем данных является файл или программа, потому что все данные перемещаются между клиентом и сервером. Для больших объёмов данных SQL-команда может быть предпочтительнее. Кроме того, из-за этого метода сквозной передачи `\copy ... from` в режиме CSV ошибочно обрабатывает единственное в строке значение данных `\.` в качестве маркера конца ввода.

`\copyright`

Показывает информацию об авторских правах и условиях распространения Postgres Pro.

`\crosstabview [ *столбВ* [ *столбГ* [ *столбТ* [ *столбсортГ* ] ] ] ]`

Выполняет содержимое буфера текущего запроса (как `\g`) и показывает результат в виде перекрёстной таблицы. Заданный запрос должен возвращать минимум три столбца. Столбец результата, заданный параметром *столбВ*, будет образовывать вертикальные заголовки, а столбец, заданный параметром *столбГ*, — горизонтальные. Заданный параметром *столбТ* столбец будет поставлять данные для отображения внутри таблицы. Столбец, выбранный параметром *столбсортГ*, будет необязательным столбцом сортировки горизонтальных заголовков.

Каждое указание столбца может представлять собой имя или номер столбца (начиная с 1). К именам применяются обычные принятые в SQL правила учёта регистра и кавычек. По умолчанию в качестве *столбВ* подразумевается столбец 1, а в качестве *столбГ* — столбец 2. Если *столбВ* и *столбГ* задаются явно, они должны различаться. Если *столбТ* не задан, в результате запроса должно быть ровно три столбца, и в качестве *столбТ* выбирается столбец, отличный от *столбВ* и *столбГ*.

Вертикальный заголовок, выводимый в самом левом столбце, содержит значения из столбца *столбВ*, в том же порядке, в каком их возвращает запрос, но без дубликатов.

Горизонтальный заголовок, выводимый в первой строке, содержит значения из столбца *столбГ*, без дубликатов. По умолчанию они располагаются в том порядке, в каком их возвращает запрос. Но если задан необязательный аргумент *столбсортГ*, он определяет столбец, который должен содержать целые числа, и тогда значения из *столбГ* будут располагаться в горизонтальном заголовке по порядку значений в *столбсортГ*.

Внутри перекрёстной таблицы для каждого уникального значения `x` в *столбГ* и каждого уникального значения `y` в *столбВ*, ячейка, размещённая на пересечении `(x,y)` содержит значение `столбТ` в строке результата запроса, в которой значение *столбГ* равно `x`, а значение *столбВ* — `y`. Если такой строки не находится, ячейка остаётся пустой. Если же находится несколько таких строк, выдаётся ошибка.

`\d[S+] [ шаблон ]`

Для каждого отношения (таблицы, представления, материализованного представления, индекса, последовательности, внешней таблицы) или составного типа, соответствующих *шаблону*, показывает все столбцы, их типы, табличное пространство (если оно изменено) и любые специальные атрибуты, такие как `NOT NULL` или значения по умолчанию. Также показываются связанные индексы, ограничения, правила и триггеры. Для сторонних таблиц также показывается связанный сторонний сервер. («Соответствие шаблону» определяется ниже в Шаблоны поиска.)

Для некоторых типов отношений `\d` показывает дополнительную информацию по каждому столбцу: значения столбца для последовательностей, индексируемые выражения для индексов и параметры обёртки сторонних данных для сторонних таблиц.

Вариант команды `\d+` похож на `\d`, но выводит больше информации: комментарии к столбцам таблицы, наличие в таблице OID, свойство replica identity, если оно было изменено, имя метода доступа, если для отношения определён метод доступа, а для представления показывается его определение.

По умолчанию отображаются только объекты, созданные пользователем. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

Примечание

Если `\d` используется без аргумента *шаблон*, эта команда удобства ради воспринимается как `\dtvmsE` и выдаёт список всех видимых таблиц, представлений, мат. представлений, последовательностей и сторонних таблиц.

`\da[S] [ шаблон ]`

Выводит список агрегатных функций вместе с типом возвращаемого значения и типами данных, которыми они оперируют. Если указан *шаблон*, показываются только те агрегатные функции, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

`\dA[+] [ шаблон ]`

Выводит список методов доступа. Если указан *шаблон*, показываются только те методы доступа, имена которых соответствуют ему. При добавлении `+` к команде для каждого метода доступа показывается его описание и связанная функция-обработчик.

`\dAc[+] [шаблон-методов-доступа [шаблон-входных-типов]]`

Выводит список классов операторов (см. Подраздел 36.16.1). Если указан *шаблон-методов-доступа*, показываются только классы, имена которых соответствует ему. Если указан *шаблон-входных-типов*, показываются только классы операторов, связанные с входными типами, имена которых соответствуют шаблону. При добавлении `+` к команде для каждого класса операторов дополнительно будет выводиться его владелец и связанное с ним семейство операторов.

`\dAf[+] [шаблон-методов-доступа [шаблон-входных-типов]]`

Выводит список семейств операторов (см. Подраздел 36.16.5). Если указан *шаблон-методов-доступа*, показываются только семейства операторов, связанные с методами доступа, имена которых соответствуют шаблону. Если указан *шаблон-входных-типов*, показываются только семейства операторов, связанные с входными типами, имена которых соответствуют шаблону. При добавлении `+` к команде для каждого семейства операторов дополнительно будет выводиться его владелец.

`\dAo[+] [шаблон-методов-доступа [шаблон-семейств-операторов]]`

Выводит список операторов, связанных с семействами операторов (см. Подраздел 36.16.2). Если указан *шаблон-методов-доступа*, показываются только те члены семейств операторов, которые связаны с методами доступа с соответствующими шаблону именами. Если указан *шаблон-семейства-операторов*, показываются только члены тех семейств, имена которых соответствуют ему. При добавлении `+` к команде для каждого оператора дополнительно будет выводиться его семейство сортировки (если это оператор упорядочивания).

`\dAp[+] [шаблон-методов-доступа [шаблон-семейств-операторов]]`

Выводит список опорных функций, связанных с семействами операторов (см. Подраздел 36.16.3). Если указан *шаблон-методов-доступа*, показываются только те функции семейств операторов, которые связаны с методами доступа с соответствующими шаблону именами. Если указан *шаблон-семейства-операторов*, показываются только функции семейств, имена которых соответствуют ему. При добавлении `+` к команде функции выводятся в развёрнутом виде, со списками фактических параметров.

`\db[+] [ шаблон ]`

Выводит список табличных пространств. Если указан *шаблон*, показываются только те табличные пространства, имена которых соответствуют ему. При добавлении `+` к команде для каждого объекта дополнительно выводятся параметры, объём на диске, права доступа и описание.

`\dc[S+] [ шаблон ]`

Выводит список преобразований между кодировками наборов символов. Если указан *шаблон*, показываются только те преобразования кодировок, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого объекта дополнительно будет выводиться описание.

`\dconfig[+] [ шаблон ]`

Выводит список параметров конфигурации сервера и их значений. Если указан *шаблон*, выводятся только те параметры, имена которых соответствуют ему. Без *шаблона* выводятся только те параметры, для которых заданы значения, отличные от значений по умолчанию. (Чтобы увидеть все параметры, введите `\dconfig *`.) При добавлении `+` к команде каждый параметр отображается с типом данных, контекстом, в котором параметр может быть установлен, и правами доступа (если они отличаются от прав доступа по умолчанию).

`\dC[+] [ шаблон ]`

Выводит список приведений типов. Если указан *шаблон*, показываются только те приведения типов, имена которых соответствуют ему. При добавлении `+` к команде для каждого объекта дополнительно будет выводиться описание.

`\dd[S] [ шаблон ]`

Показывает описания объектов следующих видов: `ограничение`, `класс операторов`, `семейство операторов`, `правило` и `триггер`. Описания остальных объектов можно посмотреть соответствующими метакомандами для этих типов объектов.

`\dd` показывает описания для объектов, соответствующих *шаблону*, или для доступных объектов указанных типов, если аргументы не заданы. Но в любом случае выводятся только те объекты, которые имеют описание. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

Описания объектов создаются SQL-командой `COMMENT`.

`\dD[S+] [ шаблон ]`

Выводит список доменов. Если указан *шаблон*, показываются только те домены, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого объекта дополнительно будут выводиться права доступа и описание.

`\ddp [ шаблон ]`

Выводит список прав доступа по умолчанию. Выводится строка для каждой роли (и схемы, если применимо), для которой права доступа по умолчанию отличаются от встроенных. Если указан *шаблон*, выводятся строки только для тех ролей и схем, имена которых соответствуют ему.

Права доступа по умолчанию устанавливаются командой `ALTER DEFAULT PRIVILEGES`. Смысл отображаемых прав объясняется в Разделе 5.8.

`\dE[S+] [ шаблон ]`
`\di[S+] [ шаблон ]`
`\dm[S+] [ шаблон ]`
`\ds[S+] [ шаблон ]`
`\dt[S+] [ шаблон ]`
`\dv[S+] [ шаблон ]`

В этой группе команд буквы `E`, `i`, `m`, `s`, `t` и `v` обозначают соответственно: внешнюю таблицу, индекс, материализованное представление, последовательность, таблицу и представление. Можно указывать все или часть этих букв, в произвольном порядке, чтобы получить список объектов этих типов. Например, `\dit` выводит список таблиц и индексов. При добавлении `+` к имени команды для каждого объекта дополнительно будут выводиться состояние хранения (постоянный, временный, нежурналируемый), физический размер на диске и описание, при наличии. Если указан *шаблон*, выводятся только объекты, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

`\des[+] [ шаблон ]`

Выводит список сторонних серверов (мнемоника: «external servers»). Если указан *шаблон*, выводятся только те серверы, имена которых соответствуют ему. Если используется форма `\des+`, то выводится полное описание каждого сервера, включая права доступа, тип, версию, параметры и описание.

`\det[+] [ шаблон ]`

Выводит список сторонних таблиц (мнемоника: «external tables»). Если указан *шаблон*, выводятся только те записи, имя таблицы или схемы которых соответствуют ему. Если используется форма `\det+`, то дополнительно выводятся общие параметры и описание сторонней таблицы.

`\deu[+] [ шаблон ]`

Выводит список сопоставлений пользователей (мнемоника: «external users»). Если указан *шаблон*, выводятся только сопоставления, в которых имена пользователей соответствуют ему. Если используется форма `\deu+`, то выводится дополнительная информация о каждом сопоставлении пользователей.

Внимание

`\deu+` также может отображать имя и пароль удалённого пользователя, поэтому следует позаботиться о том, чтобы не раскрывать их.

`\dew[+] [ шаблон ]`

Выводит список обёрток сторонних данных (мнемоника: «external wrappers»). Если указан *шаблон*, выводятся только те обёртки сторонних данных, имена которых соответствуют ему. Если используется форма `\dew+`, то дополнительно выводятся права доступа, параметры и описание обёртки.

`\df[anptwS+] [ шаблон [ арг_шаблон ... ] ]`

Выводит список функций с типами данных их результатов, аргументов и классификацией: «agg» (агрегатная), «normal», (обычная), «procedure» (процедурная), «trigger» (триггерная) или «window» (оконная). Чтобы получить функции только определённого вида (видов), добавьте в команду соответствующие буквы `a`, `n`, `p`, `t` или `w`. Если задан *шаблон*, показываются только те функции, имена которых соответствуют ему. Любые дополнительные аргументы задают шаблоны имён типов, которые сопоставляются с именами типов первого, второго и последующих аргументов функции. (У сопоставленных функций может быть больше аргументов, чем вы задали. Чтобы исключить это, запишите минус (`-`) в качестве последнего аргумента *арг_шаблон*.) По умолчанию показываются только объекты, созданные пользователями; для включения системных объектов нужно задать шаблон или добавить модификатор `S`. Если используется форма `\df+`, то дополнительно выводятся характеристики каждой функции: изменчивость, допустимость распараллеливания, владелец, классификация по безопасности, права доступа, язык, внутреннее имя (только для функций на C и внутренних функций) и описание. Исходный код отдельной функции можно посмотреть, воспользовавшись `\sf`.

`\dF[+] [ шаблон ]`

Выводит список конфигураций текстового поиска. Если указан *шаблон*, показываются только те конфигурации, имена которых соответствуют ему. Если используется форма `\dF+`, то выводится полное описание для каждой конфигурации, включая базовый синтаксический анализатор и используемые словари для каждого типа фрагментов.

`\dFd[+] [ шаблон ]`

Выводит список словарей текстового поиска. Если указан *шаблон*, показываются только словари, имена которых соответствуют ему. Если используется форма `\dFd+`, то выводится дополнительная информация о каждом словаре, включая базовый шаблон текстового поиска и параметры инициализации.

`\dFp[+] [ шаблон ]`

Выводит список анализаторов текстового поиска. Если указан *шаблон*, показываются только те анализаторы, имена которых соответствуют ему. Если используется форма `\dFp+`, то выводится полное описание для каждого анализатора, включая базовые функции и список типов фрагментов.

`\dFt[+] [ шаблон ]`

Выводит список шаблонов текстового поиска. Если указан *шаблон*, показываются только шаблоны, имена которых соответствуют ему. Если используется форма `\dFt+`, то выводится дополнительная информация о каждом шаблоне, включая имена основных функций.

`\dg[S+] [ шаблон ]`

Выводит список ролей базы данных. (Так как понятия «пользователи» и «группы» были объединены в «роли», эта команда теперь эквивалентна `\du`.) По умолчанию выводятся только роли, созданные пользователями: чтобы увидеть и системные роли, добавьте модификатор `S`. Если указан *шаблон*, выводятся только те роли, имена которых соответствуют ему. Если используется форма `\dg+`, то выводится дополнительная информация о каждой роли; в настоящее время это комментарий роли.

`\dl[+]`

Псевдоним для `\lo_list`, показывает список больших объектов. При добавлении `+` к команде для каждого большого объекта отображаются права доступа к нему, если они назначены.

`\dL[S+] [ шаблон ]`

Выводит список процедурных языков. Если указан *шаблон*, выводятся только те языки, имена которых соответствуют ему. По умолчанию показываются только языки, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого языка дополнительно будут выводиться: обработчик вызова, функция проверки, права доступа и является ли язык системным объектом.

`\dn[S+] [ шаблон ]`

Выводит список схем (пространств имён). Если указан *шаблон*, выводятся только те схемы, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого объекта дополнительно будут выводиться права доступа и описание, при наличии.

`\do[S+] [ шаблон [ арг_шаблон [ арг_шаблон ] ] ]`

Выводит список операторов, их операндов и типы результата. Если указан *шаблон*, выводятся только те операторы, имена которых соответствуют ему. Если указан один *арг_шаблон*, будут выводиться только префиксные операторы, тип правого аргумента которых соответствуют этому шаблону. Если же указано два параметра *арг_шаблон*, будут выводиться только бинарные операторы, типы аргументов которых соответствуют шаблонам. (Также можно записать `-` в качестве неиспользуемого аргумента унарного оператора.) По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого оператора будет выводиться дополнительная информация; сейчас это имя функции, на которой основан оператор.

`\dO[S+] [ шаблон ]`

Выводит список правил сортировки. Если указан *шаблон*, выводятся только те правила, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`. При добавлении `+` к команде для каждого объекта дополнительно будет выводиться описание, при наличии. Обратите внимание, что отображаются только правила сортировки, применимые к кодировке текущей базы данных, поэтому результат команды может отличаться для различных баз данных этой же установки PostgreSQL.

`\dp[S] [ шаблон ]`

Выводит список таблиц, представлений и последовательностей с их правами доступа. Если указан *шаблон*, отображаются только таблицы, представления и последовательности, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

Для установки прав доступа используются команды `GRANT` и `REVOKE`. Смысл отображаемых прав объясняется в Разделе 5.8.

`\dP[itn+] [ шаблон ]`

Выводит список секционированных отношений. Если указан *шаблон*, выводятся только те отношения, имена которых соответствуют ему. При добавлении к команде модификатора `t` (tables, таблицы) и `i` (indexes, индексы) список будет фильтроваться по типу отношения. По умолчанию выводятся и секционированные таблицы, и индексы.

При добавлении модификатора `n` («nested», вложенные) или указании шаблона выводятся также вложенные секционированные отношения и дополнительный столбец, показывающий родителя каждого секционированного отношения.

Если к команде добавляется `+`, также выводится суммарный размер всех секций каждого отношения и его описание. Если `+` дополняет модификатор `n`, будут показаны два размера: размер секций, непосредственно связанных с отношением, и общий размер всех секций, включая те, что связаны с отношением опосредованно.

`\drds [ шаблон-ролей [ шаблон-баз ] ]`

Выводит список установленных параметров конфигурации. Эти параметры могут быть специфичными для роли, специфичными для базы данных, или обеих. Параметры *`шаблон-ролей`* и *`шаблон-баз`* используются для отбора определённых ролей и баз данных, соответственно. Если они опущены, или указано `*`, выводятся все параметры конфигурации, в том числе не относящиеся к ролям или базам данных.

Команды `ALTER ROLE` и `ALTER DATABASE` используются для определения параметров конфигурации, специфичных для роли или базы данных.

`\drg[S] [ шаблон ]`

Выводит информацию о членстве в ролях с указанием назначенных атрибутов (`ADMIN`, `INHERIT` и/или `SET`) и праводателя. За информацией о членстве в ролях обратитесь к описанию команды `GRANT`.

По умолчанию выводится информация только о членстве в ролях, созданных пользователями: чтобы увидеть и системные роли, добавьте модификатор `S`. Если указан *`шаблон`*, выводятся только те роли, имена которых соответствуют ему.

`\dRp[+] [ шаблон ]`

Выводит список реплицируемых публикаций. Если указан *`шаблон`*, выводятся только те подписки, имена которых соответствуют ему. При добавлении `+` к команде для каждой публикации показываются также связанные с ней таблицы и схемы.

`\dRs[+] [ шаблон ]`

Выводит список подписок на репликацию. Если указан *`шаблон`*, выводятся только те подписки, имена которых соответствуют ему. При добавлении `+` к команде выводятся дополнительные свойства подписок.

`\duP [ шаблон ]`

Выводит профили. Если указан *`шаблон`*, отображаются только те профили, имена которых соответствуют ему. Данная команда доступна только суперпользователям.

`\dT[S+] [ шаблон ]`

Выводит список типов данных. Если указан *`шаблон`*, выводятся только те типы, имена которых соответствуют ему. При добавлении `+` к команде для каждого типа данных дополнительно будет выводиться: внутреннее имя типа, размер, допустимые значения для типа `enum` и права доступа. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

`\du[S+] [ шаблон ]`

Выводит список ролей базы данных. (Так как понятия «пользователи» и «группы» были объединены в «роли», эта команда теперь равнозначна `\dg`.) По умолчанию выводятся только роли, созданные пользователями: чтобы увидеть и системные роли, добавьте модификатор `S`. Если указан *`шаблон`*, выводятся только те роли, имена которых соответствуют ему. Если используется форма `\du+`, то выводится дополнительная информация о каждой роли; в настоящее время это комментарий роли.

`\dx[+] [ шаблон ]`

Выводит список установленных расширений. Если указан *`шаблон`*, выводятся только расширения, имена которых соответствуют ему. Если используется форма `\dx+`, то для каждого расширения выводятся все принадлежащие ему объекты.

`\dX [ шаблон ]`

Выводит объекты расширенной статистики. Если указан *`шаблон`*, отображаются только те объекты, имена которых соответствуют ему.

Состояние каждого вида расширенной статистики показывается в столбце с именем, отражающим её вид (например, Ndistinct). Состояние `defined` (определён) означает, что этот вид был запрошен при создании статистики; в противном случае отображается NULL. Чтобы узнать, производился ли анализ (`ANALYZE`) статистики и доступна ли она для планировщика, обратитесь к представлению `pg_stats_ext`.

`\dy[+] [ шаблон ]`

Выводит список событийных триггеров. Если указан *`шаблон`*, выводятся только те событийные триггеры, имена которых соответствуют ему. При добавлении + к команде для каждого объекта дополнительно будет выводиться описание.

`\e` или `\edit [имя_файла] [номер_строки]`

Если указано *`имя_файла`*, файл открывается для редактирования; после выхода из редактора содержимое файла копируется в буфер текущего запроса. Если *`имя_файла`* не задано, буфер текущего запроса копируется во временный файл, который затем редактируется тем же образом. Либо, если буфер текущего запроса пуст, во временный файл копируется последний выполненный запрос, и он затем так же редактируется.

Если после редактирования файла или предыдущего вопроса выйти из редактора без сохранения файла, буфер запросов будет очищен. В ином случае новое содержимое буфера запроса разбирается согласно обычным правилам psql, при этом весь буфер обрабатывается как одна строка. Все законченные запросы немедленно выполняются; то есть, если буфер запроса содержит точку с запятой или заканчивается ей, всё его содержимое до этого знака выполняется и удаляется из буфера. Всё остальное содержимое буфера остаётся в нём и повторно выводится на экран. Вы можете ввести точку с запятой или `\g`, чтобы передать его, либо `\r`, чтобы сбросить, очистив буфер запроса.

Прочтение буфера как одной строки в основном отражается на метакомандах: всё, что находится в буфере после метакоманды, будет воспринято как аргументы метакоманды, даже если этот текст продолжается на нескольких строках. (Поэтому таким способом нельзя выполнять скрипты с метакомандами. Для таких скриптов используйте `\i`.)

Если указан номер строки, psql будет позиционировать курсор на указанную строку файла или буфера запроса. Обратите внимание, что если указан один аргумент и он числовой, psql предполагает, что это номер строки, а не имя файла.

Подсказка

Как настроить редактор и изменить его поведение, рассказывается в разделе Переменные окружения.

`\echo` *`текст`* `[ ... ]`

Выводит вычисленные аргументы в устройство стандартного вывода, разделяя их пробелами, с переводом строки в конце. Таким образом можно добавлять дополнительную информацию в вывод скриптов. Например:

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

Если в первом аргументе передаётся `-n` без кавычек, перевод строки в конце не добавляется (и этот первый аргумент не выводится).

Подсказка

Если вы перенаправляете вывод запросов, используя `\o`, возможно, вместо этой команды следует применить `\qecho`. См. также описание `\warn`.

`\ef [`*`описание_функции`*` [`*`номер_строки`*`]]`

Эта команда извлекает из базы определение заданной функции или процедуры в форме команды `CREATE OR REPLACE FUNCTION` или `CREATE OR REPLACE PROCEDURE` и открывает его для редактирования. Редактирование осуществляется таким же образом, как и при выполнении `\edit`. Если выйти из редактора без сохранения, тело команды сбрасывается. Иначе полученная из сохранённого файла команда немедленно выполняется, если вы добавили к ней точку с запятой. В противном случае она повторно выводится на экран; вы можете ввести точку с запятой или `\g`, чтобы передать её серверу, либо `\r`, чтобы сбросить.

Для функции может быть задано только имя или имя и аргументы, например `foo(integer, text)`. Типы аргументов необходимы, если существует более чем одна функция с тем же именем.

Если функция не указана, для редактирования открывается пустая заготовка `CREATE FUNCTION`.

Если указан номер строки, psql будет позиционировать курсор на указанную строку тела функции. (Обратите внимание, что тело функции обычно не начинается на первой строке файла).

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\ef`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

Подсказка

Как настроить редактор и изменить его поведение, рассказывается в разделе Переменные окружения.

`\encoding [` *`кодировка`* `]`

Устанавливает кодировку набора символов на клиенте. Без аргумента команда показывает текущую кодировку.

`\errverbose`

Повторяет последнее серверное сообщение об ошибке с максимальным уровнем детализации, как если бы переменная `VERBOSITY` имела значение `verbose`, а `SHOW_CONTEXT` — `always`.

`\ev [`*`имя_представления`*` [`*`номер_строки`*`]]`

Эта команда извлекает из базы определение заданного представления в форме команды `CREATE OR REPLACE VIEW` и открывает его для редактирования. Редактирование осуществляется таким же образом, как и при выполнении `\edit`. Если выйти из редактора без сохранения, тело команды сбрасывается. Иначе полученная из сохранённого файла команда немедленно выполняется, если вы добавили к ней точку с запятой. В противном случае она повторно выводится на экран; вы можете ввести точку с запятой или `\g`, чтобы передать её серверу, либо `\r`, чтобы сбросить.

Если представление не указано, для редактирования открывается пустая заготовка `CREATE VIEW`.

Если указан номер строки, psql установит курсор на заданную строку в определении представления.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\ev`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\f [ `*`строка`*` ]`

Устанавливает разделитель полей для невыровненного режима вывода запросов. По умолчанию используется вертикальная черта (`|`). Равнозначно команде `\pset fieldsep`.

`\g [ (`*`параметр`*`=`*`значение`*` [...]) ] [ `*`имя_файла`*` ]`
`\g [ (`*`параметр`*`=`*`значение`*` [...]) ] [ |`*`команда`*` ]`

Передаёт содержимое буфера текущего запроса серверу для выполнения.

Если после `\g` идут скобки, внутри них содержится разделённый пробелами список определений параметров форматирования в виде *`параметр`*`=`*`значение`*. Эти определения воспринимаются так же, как и в команде `\pset` *`параметр значение`*, но устанавливаются только на время выполнения запроса. В этом списке пробелы не должны находится рядом со знаками `=`, но должны разделять определения разных параметров. Если `=`*`значение`* опускается, данный *`параметр`* меняется так же, как и при выполнении `\pset` *`параметр`* без явно заданного *`значения`*.

Если в аргументе передаётся *`имя_файла`* или `|`*`команда`*, вывод запроса записывается в указанный файл или передаётся через поток заданной команде оболочки, а не отображается как обычно. Вывод направляется в файл или команду, только если запрос успешно вернул 0 или более строк, но не когда запрос завершился неудачно или выполнялась команда, не возвращающая данные.

Если буфер текущего запроса пуст, вместо этого повторно выполняется предыдущий запрос. За исключением этой особенности метакоманда `\g` без аргументов по сути равнозначна точке с запятой. Метакоманда `\g` с аргументами является «одноразовой» альтернативой команде `\o`, и при этом в ней на время выполнения можно задать параметры форматирования вывода, которые обычно устанавливаются командой `\pset`.

Когда последний аргумент начинается с `|`, весь остаток строки воспринимается как *`команда`*, подлежащая выполнению, в которой не производится ни подстановка переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

`\gdesc`

Показывает описание (то есть имена и типы данных столбцов) результата текущего запроса в буфере. Сам запрос при этом не выполняется; но если он содержит какие-либо синтаксические ошибки, они выдаются обычным образом.

Если буфер текущего запроса пуст, будет повторно описан последний переданный запрос.

`\getenv `*`переменная_psql переменная_среды`*

Получает значение *`переменной_среды`* и присваивает его *`переменной_psql`*. Если *`переменная_среды`* не определена в среде процесса psql, *`переменная_psql`* не изменяется. Пример:

```
=> \getenv home HOME
=> \echo :home
```

```
/home/postgres
```

`\gexec`

Отправляет буфер текущего запроса на сервер, а затем обрабатывает содержимое каждого столбца каждой строки результата запроса (если он непустой) как SQL-оператор, то есть исполняет его. Например, следующая команда создаст индексы по каждому столбцу `my_table`:

```
=> SELECT format('create index on my_table(%I)', attname)
-> FROM pg_attribute
-> WHERE attrelid = 'my_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

Генерируемые запросы выполняются в том порядке, в каком возвращаются строки, слева направо, если возвращается несколько столбцов. Поля NULL игнорируются. Эти запросы передаются для обработки на сервер буквально, так что это не могут быть метакоманды psql или запросы, использующие переменные psql. В случае сбоя в одном из запросов, выполнение оставшихся запросов продолжается, если только не установлена переменная `ON_ERROR_STOP`. На выполнение каждого запроса оказывает влияние параметр `ECHO`. (Применяя команду `\gexec`, рекомендуется устанавливать в `ECHO` режим `all` или `queries`.) Такие расширенные средства, как протоколирование запросов, пошаговый режим, замер времени и т. п., так же действуют при выполнении каждого генерируемого запроса.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

`\gset [ `*`префикс`*` ]`

Отправляет буфер текущего запроса на сервер для выполнения и сохраняет результат запроса в переменных psql (см. раздел Переменные ниже). Выполняемый запрос должен возвращать ровно одну строку. Каждый столбец строки результата сохраняется в отдельной переменной, которая называется так же, как и столбец. Например:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset
=> \echo :var1 :var2
hello 10
```

Если указан *префикс*, то он добавляется в начале к именам переменных:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

Если значение столбца NULL, то вместо присваивания значения соответствующая переменная удаляется.

Если запрос завершается ошибкой или не возвращает одну строку, то никакие переменные не меняются.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

`\gx [ (`*`параметр`*`=`*`значение`*` [...]) ] [ `*`имя_файла`*` ]`
`\gx [ (`*`параметр`*`=`*`значение`*` [...]) ] [ |`*`команда`*` ]`

Метакоманда `\gx` равнозначна `\g` за исключением того, что она принудительно включает расширенный режим вывода для текущего запроса, так же как делает указание `expanded=on` в списке параметров `\pset`.

`\h` или `\help [ команда ]`

Выводит подсказку по синтаксису указанной команды SQL. Если *команда* не указана, то psql выводит список всех команд, для которых доступна справка. Если в качестве command указана звёздочка (`*`), то выводится справка по всем командам SQL.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\help`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

Примечание

Для упрощения ввода команды, состоящие из нескольких слов, можно не заключать в кавычки. Таким образом, можно просто писать **`\help alter table`**.

`\H` или `\html`

Включает вывод запросов в формате HTML. Если формат HTML уже включён, происходит переключение обратно на выровненный формат. Эта команда используется для совместимости и удобства, но в описании `\pset` вы можете узнать о других вариантах вывода.

`\i` или `\include` *имя_файла*

Читает ввод из файла *имя_файла* и выполняет его, как будто он был набран на клавиатуре.

Если *имя_файла* задано как `-` (минус), читается стандартный ввод до признака конца файла или до метакоманды `\q`. Это может быть полезно для совмещения интерактивного ввода со вводом команд из файлов. Заметьте, что при этом поведение Readline будет применяться, только если оно активно на внешнем уровне.

Примечание

Если вы хотите видеть строки файла на экране по мере их чтения, необходимо установить для переменной `ECHO` значение `all`.

`\if` *выражение*
`\elif` *выражение*
`\else`
`\endif`

Эта группа команд реализует вложенные условные блоки. Условный блок должен начинаться командой `\if` и заканчиваться `\endif`. Между ними может быть любое количество предложений `\elif`, которые могут быть дополнительно завершаться одним предложением `\else`. Между командами, формирующими блок условия, могут размещаться (и обычно размещаются) обычные запросы и другие типы команд в обратных кавычках.

Команды `\if` и `\elif` считывают свои аргументы и вычисляют их как логические выражения. Если выражение выдаёт `true`, обработка продолжается как обычно; в противном случае входные строки пропускаются до достижения соответствующих команд `\elif`, `\else` или `\endif`. Как только проверка `\if` или `\elif` оказывается успешной, аргументы последующих команд `\elif` в том же блоке не вычисляются, а считаются ложными. Строки, следующие за `\else`, обрабатываются только если ни одна из предыдущих проверок `\if` или `\elif` не была успешной.

В аргументе *выражение* команд `\if` и `\elif` производится подстановка переменных и раскрытие кавычек, как и в аргументе любой другой команды с обратной косой чертой. После этого полученное значение оценивается как значение переменной да/нет. Так что допустимым значением будет любое однозначное вхождение без учёта регистра одной из строк (или подстрок): `true`, `false`, `1`, `0`, `on`, `off`, `yes`, `no`. Например, строки `t`, `T` и `tR` все будут восприниматься как `true`.

Если выражения не приводятся к значениям true или false, будет выдано предупреждение, а их результат будет считаться ложным.

Пропускаемые строки разбираются как обычно (в них выявляются запросы и команды с обратной косой чертой), но не передаются серверу, а команды с обратной косой чертой, отличные от условных (`\if`, `\elif`, `\else`, `\endif`), просто игнорируются. В командах условий проверяется только правильность вложенности. Ссылки на переменные в пропускаемых строках не разворачиваются, как не выполняется и раскрытие обратных кавычек.

Все команды с обратной косой чертой в одном условном блоке должны содержаться в одном исходном файле. Если до того, как будут закрыты все локальные блоки `\if`, в основном файле команд будет достигнут конец файла или встретится включение другого файла (команда `\include`), psql выдаст ошибку.

Например:

```
-- проверка существования двух отдельных записей в базе данных и сохранение
-- результатов в двух разных переменных psql
SELECT
    EXISTS(SELECT 1 FROM customer WHERE customer_id = 123) as is_customer,
    EXISTS(SELECT 1 FROM employee WHERE employee_id = 456) as is_employee
\gset
\if :is_customer
    SELECT * FROM customer WHERE customer_id = 123;
\elif :is_employee
    \echo 'is not a customer but is an employee'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'not a customer or employee'
    \else
        \echo 'this will never print'
    \endif
\endif
```

`\ir` или `\include_relative` *`имя_файла`*

Команда `\ir` похожа на `\i`, но по-разному интерпретирует относительные имена файлов. При выполнении в интерактивном режиме две команды ведут себя одинаково. Однако при вызове из скрипта `\ir` интерпретирует имена файлов относительно каталога, в котором расположен скрипт, а не текущего рабочего каталога.

`\l[+]` или `\list[+] [` *`шаблон`* `]`

Выводит список баз данных на сервере и показывает их имена, владельцев, кодировку набора символов и права доступа. Если указан *`шаблон`*, выводятся только базы данных, имена которых соответствуют ему. При добавлении `+` к команде также отображаются: размер базы данных, табличное пространство по умолчанию и описание. (Информация о размере доступна только для баз данных, к которым текущий пользователь может подключиться.)

`\lo_export` *`oid_БО имя_файла`*

Читает большой объект с заданным *`oid_БО`* из базы данных и записывает его в файл *`имя_файла`*. Обратите внимание, что это несколько отличается от серверной функции `lo_export`, действующей с правами пользователя, от имени которого работает сервер базы данных, и в файловой системе сервера.

Подсказка

Используйте `\lo_list` для получения OID больших объектов.

`\lo_import` *`имя_файла`* `[` *`комментарий`* `]`

Сохраняет файл в большом объекте Postgres Pro. При этом с объектом может быть связан указанный комментарий. Пример:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

Ответ указывает на то, что большой объект получил OID 152801, который может быть использован для доступа к вновь созданному объекту в будущем. Для удобства чтения рекомендуется всегда связывать объекты с понятными комментариями. OID и комментарии можно посмотреть с помощью команды `\lo_list`.

Обратите внимание, что это немного отличается от функции сервера `lo_import`, так как действует от имени локального пользователя в локальной файловой системе, а не пользователя сервера в файловой системе сервера.

`\lo_list[+]`

Показывает список всех больших объектов Postgres Pro, хранящихся в базе данных, вместе с предоставленными комментариями. При добавлении `+` к команде для каждого большого объекта отображаются права доступа к нему, если они назначены.

`\lo_unlink` *`oid_БО`*

Удаляет большой объект с *`oid_БО`* из базы данных.

> ### Подсказка
>
> Используйте `\lo_list` для получения OID больших объектов.

`\o` или `\out` `[` *`имя_файла`* `]`
`\o` или `\out` `[ |`*`команда`* `]`

Результаты запросов будут сохраняться в файле *`имя_файла`* или перенаправляться команде оболочки (заданной аргументом *`команда`*). Если аргумент не указан, результаты запросов перенаправляются на стандартный вывод.

Если аргумент начинается с `|`, весь остаток строки воспринимается как *`команда`*, подлежащая выполнению, в которой не производится ни подстановка переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

«Результаты запросов» включают в себя все таблицы, ответы команд, уведомления, полученные от сервера баз данных, а также вывод от метакоманд, обращающихся к базе (таких как `\d`), но не сообщения об ошибках.

> ### Подсказка
>
> Чтобы вставить текст между результатами запросов, используйте `\qecho`.

`\p` или `\print`

Печатает содержимое буфера текущего запроса в стандартный вывод. Если этот буфер пуст, будет напечатан последний выполненный запрос.

`\password` `[` *`имя_пользователя`* `]`

Изменяет пароль указанного пользователя (по умолчанию, текущего пользователя). Эта команда запрашивает новый пароль, шифрует и отправляет его на сервер в виде команды `ALTER ROLE`. Это гарантирует, что новый пароль не отображается в открытом виде в истории команд, журнале сервера или в другом месте.

`\prompt [ текст ] имя`

Предлагает пользователю ввести значение, которое будет присвоено переменной *имя*. Дополнительно можно указать *текст* подсказки. (Если подсказка состоит из нескольких слов, то её текст нужно взять в одинарные кавычки).

По умолчанию, `\prompt` использует терминал для ввода и вывода. Однако если используется параметр командной строки `-f`, `\prompt` использует стандартный ввод и стандартный вывод.

`\pset [ параметр [ значение ] ]`

Эта команда устанавливает параметры, влияющие на вывод результатов запросов. Указание *параметр* определяет, какой параметр требуется установить. Семантика *значения* зависит от выбранного параметра. Для некоторых параметров отсутствие *значения* означает переключение значения, либо сброс значения, как описано ниже в разделе конкретного параметра. Если такое поведение не упоминается, то пропуск *значения* приводит к отображению текущего значения параметра.

`\pset` без аргументов выводит текущий статус всех параметров команды.

Имеются следующие параметры:

`border`

Здесь *значение* должно быть числом. В целом, чем больше это число, тем больше границ и линий будет в таблицах, но детали зависят от формата. В формате HTML заданное значение напрямую отображается в атрибут `border=....` Для большинства других форматов имеют смысл только значения 0 (нет границы), 1 (внутренние разделительные линии) и 2 (граница таблицы), а значения больше 2 воспринимаются как `border = 2`. Форматы `latex` и `latex-longtable` дополнительно поддерживают значение 3, добавляющее разделительные линии между строками данных.

`columns`

Устанавливает максимальную ширину для формата `wrapped`, а также ограничение по ширине, свыше которого будет требоваться постраничник или произойдёт переключение в вертикальное отображение при режиме `expanded auto`. При значении 0 (по умолчанию) максимальная ширина управляется переменной среды `COLUMNS` или шириной экрана, если `COLUMNS` не установлена. Кроме того, если `columns` равно нулю, то формат `wrapped` влияет только на вывод на экран. Если `columns` не равно 0, то это также влияет на вывод в файл или в другую команду через канал.

`csv_fieldsep`

Устанавливает разделитель полей для формата вывода CSV. Если символ-разделитель оказывается внутри значения поля, оно выводится в двойных кавычках согласно стандартным правилам CSV. По умолчанию разделителем является запятая.

`expanded` (или `x`)

Указанное *значение* допускает варианты `on` или `off`, которые включают или отключают развёрнутый режим, либо `auto`. Если *значение* опущено, команда включает/отключает режим. Когда развёрнутый режим включён, результаты запроса выводятся в две колонки: имя столбца в левой, данные в правой. Этот режим полезен, если данные не помещаются на экране в обычном «горизонтальном» режиме. При выборе `auto` развёрнутый режим используется, когда результат запроса содержит несколько столбцов и по ширине не умещается на экране; в противном случае используется обычный режим. Режим `auto` распространяется только на форматы `aligned` и `wrapped`. С другими форматами он всегда равнозначен отключённому состоянию.

`fieldsep`

Устанавливает разделитель полей для невыровненного режима вывода запросов. Таким образом, можно формировать вывод, в котором значения будут разделены, например, табуляцией. Это может быть предпочтительным для использования в других программах. Для установки символа табуляции в качестве разделителя полей введите `\pset fieldsep '\t'`. По умолчанию используется вертикальная черта (`'|'`).

`fieldsep_zero`

Устанавливает разделитель полей для невыровненного режима вывода в нулевой байт.

`footer`

Возможны два варианта *`значения`*: `on` или `off`, которые включают или отключают вывод результирующей строки с количеством выбранных записей `(n строк)`. Если *`значение`* не указано, то команда переключает текущее значение в `on` или `off`.

`format`

Устанавливает один из следующих форматов вывода: `aligned`, `asciidoc`, `csv`, `html`, `latex`, `latex-longtable`, `troff-ms`, `unaligned` или `wrapped`. Допускается сокращение слова до уникального значения.

Формат `aligned` это стандартный, удобочитаемый, хорошо отформатированный текстовый вывод. Используется по умолчанию.

В формате `unaligned` все столбцы выводятся в одной строке и отделяются друг от друга активным разделителем полей. Это полезно для получения вывода, который будут читать другие программы, например для вывода данных с разделителем Tab или через запятую. При этом, если разделитель окажется в содержимом столбца, этот символ будет выведен как есть; в таких случаях полезнее будет формат CSV.

В формате `csv` значения столбцов выводятся через запятую и могут заключаться в кавычки по правилам, описанным в *RFC 4180*. Вывод этого формата совместим с форматом CSV серверной команды `COPY`. Первой выводится строка с именами столбцов, если только не включён режим `tuples_only`. Верхний и нижний колонтитулы не выводятся. Строки разделяются символами конца строки, зависящими от ОС; обычно это символ новой строки (`\n`) в Unix-подобных системах и комбинация символов перевода каретки и новой строки (`\r\n`) в Microsoft Windows. Отличный от запятой разделитель полей можно установить командой `\pset csv_fieldsep`.

Формат `wrapped` похож на `aligned`, но переносит длинные значения столбцов на новые строки, чтобы общий вывод поместился в заданную ширину. Задание ширины вывода описано в параметре `columns`. Обратите внимание, что psql не будет пытаться переносить на новые строки заголовки столбцов. Поэтому формат `wrapped` работает так же, как `aligned` если общая ширина, требуемая для всех заголовков столбцов, превышает установленную максимальную ширину.

Форматы `asciidoc`, `html`, `latex`, `latex-longtable` и `troff-ms` выводят таблицы, которые предназначены для включения в документы с помощью соответствующего языка разметки. Они не являются полноценными документами! Это может быть не критично для HTML, но для LaTeX обязателен документ-контейнер. Формат `latex` использует среду LaTeX `tabular`, а формат `latex-longtable` требует наличия пакетов `longtable` и `booktabs`.

`linestyle`

Задаёт стиль отрисовки линий границы: `ascii`, `old-ascii` или `unicode`. Допускается сокращение слова до уникального значения. (Это значит, что одной буквы будет достаточно.) Значение по умолчанию: `ascii`. Этот параметр действует только в форматах `aligned` и `wrapped`.

Стиль `ascii` использует обычные символы ASCII. Символы новой строки в данных показываются с использованием символа `+` в правом поле. Когда при формате `wrapped` происходит перенос данных на новую строку (без символа новой строки), ставится точка (`.`) в правом поле первой строки и точка в левом поле следующей строки.

Стиль `old-ascii` использует обычные символы ASCII в стиле PostgreSQL 8.4 и раньше. Символы новой строки в данных отображаются, используя символ `:` вместо левого разделителя полей. Когда происходит перенос данных на новую строку без символа новой строки, символ `;` используется вместо левого разделителя полей.

Стиль `unicode` использует символы Юникода для рисования линий. Символы новой строки в данных показываются с использованием символа возврата каретки в правом поле. Когда при формате `wrapped` происходит перенос данных на новую строку (без символа новой строки), ставится символ многоточия в правом поле первой строки и в левом поле следующей строки.

Когда значение `border` больше нуля, параметр `linestyle` также определяет символы, которыми будут рисоваться границы. Обычные символы ASCII работают везде, но символы Юникода смотрятся лучше на терминалах, распознающих их.

`null`

Устанавливает строку, которая будет напечатана вместо значения null. По умолчанию не печатается ничего, что можно ошибочно принять за пустую строку. Например, можно было бы предпочесть `\pset null '(null)'`.

`numericlocale`

Если задаётся *`значение`*, возможны два варианта: `on` или `off`, которые включают или отключают отображение специфичного для локали символа, разделяющего группы цифр левее десятичной точки. Если *`значение`* не указано, то команда переключает вывод чисел с локализованного на обычный и обратно.

`pager`

Управляет использованием постраничника для просмотра результатов запросов и справочной информации psql. Если `pager` имеет значение `off`, программа постраничного просмотра (постраничник) не используется. Если `pager` имеет значение `on`, эта программа используется при необходимости, т. е. когда вывод на терминал не помещается на экране. Параметр `pager` также может иметь значение `always`, при этом постраничник будет использоваться всегда, независимо от того, помещается вывод на экран терминала или нет. Команда `\pset pager` без указания *`значения`* переключает варианты `on` и `off`.

Если установлена переменная среды `PSQL_PAGER` или `PAGER`, вывод передаётся указанной программе. В противном случае используется платформозависимая программа по умолчанию (например, `more`).

Команда `\watch`, которая повторяет один и тот же запрос, для выбора постраничника вместо этого использует переменную среды `PSQL_WATCH_PAGER` (в системах Unix). Для такого вывода постраничник выбирается отдельно, потому что традиционные постраничники могут обрабатывать его некорректно, но вы можете передавать его программам, распознающим этот формат вывода psql (например, `pspg --stream`).

`pager_min_lines`

Если в `pager_min_lines` задаётся число, превышающее высоту страницы, программа постраничного вывода не будет вызываться, пока не наберётся заданное число строк для вывода. Значение по умолчанию — 0.

`recordsep`

Устанавливает разделитель записей (строк) для невыровненного режима вывода. По умолчанию используется символ новой строки.

`recordsep_zero`

Устанавливает разделитель записей для невыровненного режима вывода в нулевой байт.

`tableattr` (или `T`)

Устанавливает атрибуты, которые будут помещены в тег `table`, при формате вывода HTML. Например, это может быть `cellpadding` или `border`. Заметьте, что, вероятно, не нужно здесь задавать `border`, так как для этого уже есть `\pset border`. Если *`значение`* не задано, атрибуты таблицы удаляются.

В формате `latex-longtable` этот параметр контролирует пропорциональную ширину каждого столбца, данные которого выровнены по левому краю. Он указывается как список разделённых пробелами значений, например `'0.2 0.2 0.6'`. Для столбцов, которым не хватает значений, используется последнее из заданных.

`title` (или `C`)

Устанавливает заголовок таблицы для любых впоследствии выводимых таблиц. Это можно использовать для задания описательных тегов при формировании вывода. Если *`значение`* не задано, заголовок таблицы удаляется.

`tuples_only` (или `t`)

Возможны два варианта *`значения`*: `on` или `off`, которые включают или отключают режим вывода только кортежей. Если *`значение`* не указано, то команда переключает с режима вывода только кортежей на обычный режим и обратно. Обычный вывод включает в себя дополнительную информацию, такую как заголовки столбцов и различные колонтитулы. В режиме вывода только кортежей отображаются только фактические табличные данные.

`unicode_border_linestyle`

Устанавливает стиль рисования границ для стиля линий `unicode`: `single` (одинарный) или `double` (двойной).

`unicode_column_linestyle`

Устанавливает стиль рисования колонок для стиля линий `unicode`: `single` (одинарный) или `double` (двойной).

`unicode_header_linestyle`

Устанавливает стиль рисования заголовка для стиля линий `unicode`: `single` (одинарный) или `double` (двойной).

`xheader_width`

Устанавливает максимальную ширину заголовка для расширенного режима вывода. Допустимые значения: `full` (значение по умолчанию), `column`, `page` или *`целое_число`*.

`full`: заголовок в расширенном режиме не обрезается, строка форматируется по самому широкому столбцу.

`column`: строка заголовка обрезается до ширины первого столбца вывода.

`page`: строка заголовка обрезается до ширины окна терминала.

`целое_число`: строка заголовка обрезается по указанной максимальной ширине.

Иллюстрацию того, как могут выглядеть различные форматы, можно увидеть в разделе Примеры ниже.

Подсказка

Для некоторых параметров `\pset` есть короткие команды. См. `\a`, `\C`, `\f`, `\H`, `\t`, `\T` и `\x`.

`\q` или `\quit`

Выход из psql. При использовании в скрипте прекращается только выполнение этого скрипта.

`\qecho` *`текст`* `[ ... ]`

Эта команда идентична `\echo` за исключением того, что вывод будет записываться в канал вывода запросов, установленный `\o`.

`\r` или `\reset`

Сбрасывает (очищает) буфер запроса.

`\restrict` *`restrict_key`*

Включает ограниченный режим с заданным ключом. В данном режиме доступна только одна метакоманда: `\unrestrict`, она отключает ограниченный режим. Ключ может содержать только буквенно-цифровые символы.

Данная команда в первую очередь используется для выгрузок текстовых файлов, сгенерированных с помощью утилит pg_dump, pg_dumpall и pg_restore, однако может применяться и для других целей.

`\s [` *`имя_файла`* `]`

Записывает историю команд psql в файл *`имя_файла`*. Если *`имя_файла`* не указано, история команд выводится в стандартный вывод (с использованием постраничника, когда уместно). Этот параметр недоступен, если psql был собран без поддержки Readline.

`\set [` *`имя`* `[` *`значение`* `[ ... ] ] ]`

Задаёт для переменной psql *`имя`* указанное *`значение`* или, если указано несколько значений, все эти значения, соединённые вместе. Если присутствует только один аргумент, значением переменной становится пустая строка. Для сброса переменной используйте команду `\unset`.

`\set` без аргументов выводит имена и значения всех psql переменных, установленных в настоящее время.

Имена переменных могут содержать буквы, цифры и знаки подчёркивания. Подробнее см. раздел Переменные ниже. Имена переменных чувствительны к регистру.

Некоторые переменные отличаются от остальных, тем что управляют поведением psql или устанавливаются автоматически, отражая состояние соединения. Они описаны ниже, в разделе Переменные.

Примечание

Эта команда не имеет отношения к SQL-команде `SET`.

`\setenv` *`имя`* `[` *`значение`* `]`

Задаёт для переменной среды *`имя`* *`значение`* или, если *`значение`* не задано, удаляет переменную среды. Пример:

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

`\sf[+]` *`описание_функции`*

Эта команда извлекает из базы и выводит определение заданной функции или процедуры в форме команды `CREATE OR REPLACE FUNCTION` или `CREATE OR REPLACE PROCEDURE`. Определение выдаётся в текущий канал вывода запроса, установленный `\o`.

Для функции может быть задано только имя или имя и аргументы, например `foo(integer, text)`. Типы аргументов необходимы, если существует более чем одна функция с тем же именем.

При добавлении + к команде строки вывода нумеруются, первая строка тела функции получит номер 1.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\sf`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\sv[+]` *`имя_представления`*

Извлекает из базы данных и выводит определение указанного представления в форме команды `CREATE OR REPLACE VIEW`. Определение выводится в текущий канал вывода запросов, установленный `\o`.

При добавлении + к команде строки вывода нумеруются, начиная с 1.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\sv`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек.

`\t`

Включает/выключает отображение имён столбцов и результирующей строки с количеством выбранных записей для запросов. Эта команда эквивалентна `\pset tuples_only` и предоставлена для удобства.

`\T` *`параметры_таблицы`*

Устанавливает атрибуты, которые будут помещены в тег `table` при формате вывода HTML. Эта команда эквивалентна `\pset tableattr` *`параметры_таблицы`*.

`\timing [` *`on`* `|` *`off`* `]`

С параметром данная команда, в зависимости от него, включает/отключает отображение времени выполнения каждого SQL-оператора. Без параметра она меняет состояние отображения на противоположное. Время выводится в миллисекундах; интервалы больше 1 секунды выводятся в формате минуты:секунды, а при необходимости в вывод также добавляются часы и дни.

`\unrestrict` *`restrict_key`*

Выключает ограниченный режим (режим, в котором все другие метакоманды заблокированы) при условии, что используемый ключ совпадает с ключом, заданным при включении ограниченного режима командой `\restrict`.

Данная команда в первую очередь используется для выгрузок текстовых файлов, сгенерированных с помощью утилит pg_dump, pg_dumpall и pg_restore, однако может применяться и для других целей.

`\unset` *`имя`*

Удаляет psql переменную *`имя`*.

Большинство переменных, управляющих поведением psql, нельзя сбросить; команда `\unset` для них воспринимается как установка значений по умолчанию. См. раздел Переменные ниже.

`\w` или `\write` *`имя_файла`*
`\w` или `\write |`*`команда`*

Выводит буфер текущего запроса в файл *`имя_файла`* или через канал в команду оболочки *`команда`*. Если этот буфер пуст, будет выведен последний выполненный запрос.

Если аргумент начинается с `|`, весь остаток строки воспринимается как *`команда`*, подлежащая выполнению, в которой не производится ни подстановка переменных, ни раскрытие обратных кавычек. Это продолжение строки просто передаётся оболочке в буквальном виде.

`\warn` *`текст`* `[ ... ]`

Эта команда идентична `\echo` за исключением того, что её вывод выдаётся в поток вывода ошибок psql, а не в поток стандартного вывода.

`\watch [ i[nterval]=`*`секунды`*` ] [ c[ount]=`*`количество_выполнений`*` ] [ m[in_rows]=`*`строки`*` ] [` *`секунды`* `]`

Выполнять текущий запрос в буфере (как `\g`) заданное количество раз или пока команда не будет прервана, не возникнет ошибка, не будет достигнут лимит числа выполнений или запрос не перестанет возвращать минимальное количество строк. Аргумент задаёт количество секунд ожидания между выполнениями запроса (по умолчанию 2). Для обратной совместимости задавать *`секунды`* можно с префиксом `interval=` или без него. Результат каждого запроса выводится с заголовком, включающим строку `\pset title` (если она задана), время запуска запроса и интервал задержки.

Если буфер текущего запроса пуст, будет повторно выполнен последний переданный запрос.

`\x [` *`on`* `|` *`off`* `|` *`auto`* `]`

Устанавливает или переключает режим развёрнутого вывода таблицы. Это эквивалентно `\pset expanded`.

`\z[S] [` *`шаблон`* `]`

Выводит список таблиц, представлений и последовательностей с их правами доступа. Если указан *`шаблон`*, отображаются только таблицы, представления и последовательности, имена которых соответствуют ему. По умолчанию показываются только объекты, созданные пользователями. Для включения системных объектов нужно задать шаблон или добавить модификатор `S`.

Это псевдоним для `\dp` («показать права доступа»).

`\! [` *`команда`* `]`

Без аргументов запускает подчинённую оболочку; когда эта оболочка завершается, psql продолжает работу. Если добавлен аргумент, запускает команду оболочки *`команда`*.

В отличие от большинства других метакоманд весь остаток строки всегда воспринимается как аргументы `\!`, и в этих аргументах не выполняется ни подстановка переменных, ни раскрытие обратных кавычек. Этот текст просто передаётся оболочке в буквальном виде.

`\? [` *`тема`* `]`

Показывает справочную информацию. Необязательный параметр *`тема`* (по умолчанию `commands`) выбирает описание интересующей части psql: `commands` описывает команды psql с обратной косой чертой; `options` описывает параметры командной строки, которые можно передать psql; а `variables` выдаёт справку по переменным конфигурации psql.

`\;`

Точка с запятой после косой черты не является метакомандой в том смысле, что предыдущие; при её вводе в буфер просто добавляется точка с запятой без обработки.

Обычно psql передаёт SQL-команду серверу, как только встречает завершающую команду точку с запятой, даже если текущая строка на этом не заканчивается. Так, например, при вводе

```
select 1; select 2; select 3;
```

на сервер по отдельности будут переданы три SQL-команды, и результат каждой команды будет выведен перед переходом к следующей. Однако если точка с запятой вводится как `\;`, команда не передаётся на обработку, так что команды до и после этих символов по сути объединяются и передаются серверу в одном запросе. Поэтому, например, при выполнении

```
select 1\; select 2\; select 3;
```

серверу передаются сразу три SQL-команды при достижении неэкранированной точки с запятой. Сервер выполняет такой запрос как одну транзакцию, если только в строку не включены явные команды `BEGIN`/`COMMIT`, которые разделят её на несколько транзакций. (Подробнее о том, как сервер обрабатывает строки, включающие несколько команд, рассказывается в Подразделе 53.2.2.1.)

Шаблоны поиска

Различные `\d` команды принимают параметр *`шаблон`* для указания имени (имён) объектов для отображения. В простейшем случае шаблон — это точное имя объекта. Символы внутри шаблона обычно приводятся к нижнему регистру, как и для имён SQL-объектов; к примеру `\dt FOO` выводит таблицу с именем `foo`. Как и для SQL имён, двойные кавычки вокруг шаблона предотвращают перевод в нижний регистр. Для включения символа двойной кавычки в шаблон используются два символа двойных кавычек подряд внутри шаблона в двойных кавычках. Опять же, это соответствует правилам для SQL-идентификаторов. Например `\dt "FOO""BAR"` будет выводить таблицу с именем `FOO"BAR` (но не `foo"bar`). В отличие от обычных правил для SQL-имён, можно взять в двойные кавычки только часть шаблона, например `\dt FOO"FOO"BAR` будет выводить таблицу с именем `fooFOObar`.

Если *`шаблон`* вообще не указан, команды `\d` выводят все объекты, видимые с текущим путём поиска схем. Это эквивалентно указанию `*` в качестве шаблона. (Объект считается *видимым*, если схема, к которой он относится, находится в пути поиска, и объект с таким же типом и именем в пути поиска ещё не встречался. Это эквивалентно утверждению, что на объект можно ссылаться по имени, без явного указания схемы.) Чтобы увидеть все объекты в базе данных, независимо от видимости, используйте в качестве шаблона `*.*`.

Внутри шаблона `*` обозначает любое количество символов, включая отсутствие символов. `?` соответствует любому одному символу. (Это соответствует шаблонам имён файлов в Unix.) Например, `\dt int*` отображает все таблицы, имена которых начинаются на `int`. Однако внутри двойных кавычек `*` и `?` теряют своё специальное значение и становятся обычными символами.

Шаблон отношения, содержащий точку (`.`), интерпретируется как шаблон имени схемы, за которым следует шаблон имени объекта. Например, `\dt foo*.*bar*` отображает все таблицы, имена которых включают `bar`, и расположенные в схемах, имена которых начинаются с `foo`. Шаблону, не содержащему точку, могут соответствовать только объекты текущей схемы. Опять же, точка внутри двойных кавычек теряет своё специальное значение. Шаблон отношения, содержащий 2 точки (`.`), интерпретируется как имя БД, за которым следуют шаблон имени схемы и шаблон имени объекта. Компонент с именем БД не будет восприниматься как шаблон, в нём должно указываться имя текущей подключённой базы; в противном случае возникнет ошибка.

Шаблон схемы, содержащий точку (`.`), интерпретируется как имя БД, за которым следует шаблон имени схемы. Например, `\dn mydb.*foo*` отображает все схемы, имена которых включают `foo`. Компонент с именем БД не будет восприниматься как шаблон, в нём должно указываться имя текущей подключённой базы; в противном случае возникнет ошибка.

Опытные пользователи могут использовать возможности регулярных выражений, такие как классы символов. Например `[0-9]` соответствует любой цифре. Все специальные символы регулярных выражений работают как описано в Подразделе 9.7.3, за исключением: `.` используется в качестве разделителя, как говорилось выше; `*` соответствует регулярному выражению `.*`; `?` соответствует `.`, а также символ `$`, который не имеет специального значения. При необходимости эти символы можно эмулировать указывая `?` для эмуляции `.`, `(R+|)` для `R*`, `(R|)` для `R?`. `$` не требуется, как символ регулярного выражения, потому что шаблон должен соответствовать имени целиком, в отличие от обычной интерпретации регулярных выражений (другими словами, `$` автоматически добавляется в шаблон). Используйте `*` в начале и/или в конце, если не хотите, чтобы шаблон закреплялся. Обратите внимание, что внутри двойных кавычек, все специальные символы регулярных выражений теряют своё специальное значение и соответствуют сами себе. Также, специальные символы регулярных выражений не действуют в шаблонах для имён операторов (т. е. в аргументе команды `\do`).

Расширенные возможности

Переменные

psql предоставляет возможности подстановки переменных подобные тем, что используются в командных оболочках Unix. Переменные представляют собой пары имя/значение, где значением может быть любая строка любой длины. Имя должно состоять из букв (включая нелатинские буквы), цифр и знаков подчёркивания.

Чтобы установить переменную, используется метакоманда psql `\set`. Например:

```
testdb=> \set foo bar
```

присваивает переменной `foo` значение `bar`. Чтобы получить значение переменной, нужно поставить двоеточие перед её именем, например:

```
testdb=> \echo :foo
bar
```

Это работает как в обычных SQL-командах, так и в метакомандах; подробности в разделе Интерполяция SQL ниже.

При вызове `\set` без второго аргумента переменной присваивается пустая строка. Для сброса (то есть удаления) переменной используйте команду `\unset`. Чтобы посмотреть значения всех переменных, вызовите `\set` без аргументов.

Примечание

На аргументы `\set` распространяются те же правила подстановки, что и для других команд. Таким образом можно создавать интересные ссылки, например `\set :foo 'something'`, получая «мягкие ссылки» в Perl или «переменные переменных» в PHP. К сожалению (или к счастью?), с этими конструкциями нельзя сделать ничего полезного. С другой стороны, `\set bar :foo` является прекрасным способом копирования переменной.

Некоторые переменные обрабатываются в psql особым образом. Они представляют собой определённые параметры, которые могут быть изменены во время выполнения путём присваивания нового значения, а в некоторых переменных содержится изменяемое состояние psql. По соглашению, имена специальных переменных состоят только из заглавных ASCII-букв (и, возможно, цифр и знаков подчёркивания). Для максимальной совместимости в будущем старайтесь не использовать такие имена для собственных переменных.

Переменные, управляющие поведением psql, обычно нельзя сбросить или задать для них недопустимые значения. Команда `\unset` для них допускается, но воспринимается как установка значения по умолчанию. Команда `\set` без второго аргумента воспринимается как присваивание переменной значения `on`, для управляющих переменных, принимающих это значение, и не принимается для других. Также управляющие переменные, принимающие значения `on` и `off`, примут и другие общепринятые написания логических значений, например `true` и `false`.

Специальные переменные:

`AUTOCOMMIT`

При значении `on` (по умолчанию) после каждой успешно выполненной команды выполняется фиксация изменений. Чтобы отложить фиксацию изменений в этом режиме, нужно выполнить

SQL-команду `BEGIN` или `START TRANSACTION`. При значении `off` или если переменная не определена, фиксация изменений не происходит до тех пор, пока явно не выполнена команда `COMMIT` или `END`. При значении `off` неявно выполняется `BEGIN` непосредственно перед любой командой, за исключением случаев когда: команда уже в транзакционном блоке; перед самой командой `BEGIN` или другой командой управления транзакциями; перед командой, которая не может выполняться внутри транзакционного блока (например `VACUUM`).

Примечание

Если режим `autocommit` отключён, необходимо явно откатывать изменения в неуспешных транзакциях, выполняя команду `ABORT` или `ROLLBACK`. Также имейте в виду, что при выходе из сеанса без фиксации изменений несохранённые изменения будут потеряны.

Примечание

Включённый режим `autocommit` является традиционным для Postgres Pro, а выключенный режим ближе к спецификации SQL. Если вы предпочитаете отключить режим `autocommit`, это можно сделать в общесистемном файле `psqlrc` или в персональном файле `~/.psqlrc`.

`COMP_KEYWORD_CASE`

Определяет, какой регистр букв будет использован при автоматическом завершении ключевых слов SQL. Если установлено в `lower` или `upper`, будет использоваться нижний или верхний регистр соответственно. Если установлено в `preserve-lower` или `preserve-upper` (по умолчанию), то завершаемое слово будет в том же регистре, что и уже введённое начало слова, но последующие слова, завершаемые полностью, будут в нижнем или верхнем регистре соответственно.

`DBNAME`

Имя базы данных, к которой вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно изменить или сбросить.

`ECHO`

Со значением `all` все непустые входящие строки выдаются в стандартный вывод по мере их чтения. (Это не относится к строкам, считываемым интерактивно.) Чтобы выбрать такое поведение при запуске программы, добавьте ключ `-a`. Со значением `queries` psql выдаёт каждый запрос, отправляемый серверу, в стандартный вывод. Этому значению соответствует ключ `-e`. Со значением `errors` в стандартный поток ошибок выдаются только запросы, вызвавшие ошибки. Ему соответствует ключ `-b`. Со значением `none` (по умолчанию), никакие запросы не выводятся.

`ECHO_HIDDEN`

Если эта переменная имеет значение `on` и метакоманда обращается к базе данных, сначала выводится текст нижележащего запроса. Это помогает изучать внутреннее устройство Postgres Pro и реализовывать похожую функциональность в своих программах. (Чтобы включить такое поведение при запуске программы, воспользуйтесь ключом `-E`.) Если вы зададите для этой переменной значение `noexec`, запросы будут просто показываться, но не будут отправляться на сервер и выполняться. Значение по умолчанию — `off`.

`ENCODING`

Текущая кодировка символов на стороне клиента. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы) и при смене кодировки командой `\encoding`, но эту переменную можно изменить или сбросить.

`ERROR`

`true` в случае ошибки последнего SQL-запроса, `false`, если он был выполнен успешно. См. также `SQLSTATE`.

`FETCH_COUNT`

Если значение этой переменной — целое число больше нуля, результаты запросов `SELECT` извлекаются из базы данных и отображаются группами с заданным количеством строк, в отличие от поведения по умолчанию, когда перед отображением результирующий набор накапливается целиком. Это позволяет использовать ограниченный размер памяти независимо от размера выборки. При включении этой функциональности обычно используются значения от 100 до 1000. Имейте в виду, что запрос может завершиться ошибкой после отображения некоторого количества строк.

Подсказка

Хотя можно использовать любой формат вывода, формат по умолчанию `aligned` как правило выглядит хуже, потому что каждая группа по `FETCH_COUNT` строк форматируется отдельно, что может привести к разной ширине столбцов в разных группах. Остальные форматы вывода работают лучше.

`HIDE_TABLEAM`

Если эта переменная равна `true`, информация о методах доступа таблицы не выводится. Это полезно прежде всего для регрессионных тестов.

`HIDE_TOAST_COMPRESSION`

Если эта переменная равна `true`, информация о методах сжатия столбцов не выводится. Это полезно прежде всего для регрессионных тестов.

`HISTCONTROL`

Если переменная имеет значение `ignorespace`, строки, начинающиеся с пробела, не сохраняются в истории. Если она имеет значение `ignoredups`, в историю не добавляются строки, которые в ней уже есть. Значение `ignoreboth` объединяет эти два варианта. Со значением `none` (по умолчанию) в истории сохраняются все строки, считываемые в интерактивном режиме.

Примечание

Эта функциональность была бессовестно списана с Bash.

`HISTFILE`

Имя файла, в котором будет сохраняться список истории команд. Если эта переменная не определена, имя файла берётся из переменной окружения `PSQL_HISTORY`. Если и она не задана, используется имя по умолчанию — `~/.psql_history` или `%APPDATA%\postgresql\psql_history` в Windows. Например, если установить:

```
\set HISTFILE ~/.psql_history-:DBNAME
```

в `~/.psqlrc`, psql будет вести отдельный файл истории для каждой базы данных.

Примечание

Эта функциональность была бессовестно списана с Bash.

`HISTSIZE`

Максимальное число команд, которые будут сохраняться в истории команд (по умолчанию 500). Если задано отрицательное значение, ограничение не накладывается.

> **Примечание**
>
> Эта функциональность была бессовестно списана с Bash.

`HOST`

Имя компьютера, где работает сервер базы данных, к которому вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно изменить или сбросить.

`IGNOREEOF`

Если равно 1 или меньше, символ конца файла (EOF, обычно передаётся сочетанием клавиш **Control+D**) в интерактивном сеансе psql завершит работу приложения. Если значение больше 1, оно определяет, сколько последовательных символов EOF нужно ввести, чтобы завершить интерактивный сеанс. Если значение переменной не является числовым, оно воспринимается как 10. По умолчанию — 0.

> **Примечание**
>
> Эта функциональность была бессовестно списана с Bash.

`LASTOID`

Содержит значение последнего OID, полученного командой `INSERT` или `\lo_import`. Корректное значение переменной гарантируется до тех пор, пока не будет отображён результат следующей SQL-команды. Серверы Postgres Pro с 12 версии не поддерживают системные столбцы OID, поэтому при обращении к таким серверам после `INSERT` всегда будет выдаваться нулевой LASTOID.

`LAST_ERROR_MESSAGE`
`LAST_ERROR_SQLSTATE`

Основное сообщение об ошибке и связанный код SQLSTATE для последнего неудавшегося запроса в текущем сеансе psql либо пустая строка и `00000`, если в текущем сеансе не происходили ошибки.

`ON_ERROR_ROLLBACK`

При значении `on`, если команда в блоке транзакции выдаёт ошибку, ошибка игнорируется и транзакция продолжается. Со значением `interactive` такие ошибки игнорируются только в интерактивных сеансах, но не в скриптах. Со значением `off` (по умолчанию) команда в блоке транзакции, выдающая ошибку, прерывает всю транзакцию. Для реализации режима отката транзакции за вас неявно выполняется команда `SAVEPOINT` непосредственно перед каждой командой в блоке транзакции, а в случае ошибки команды происходит откат к этой точке сохранения.

`ON_ERROR_STOP`

По умолчанию, после возникновения ошибки обработка команд продолжается. Если эта переменная установлена в значение `on`, обработка команд будет немедленно прекращена. В интерактивном режиме psql вернётся в командную строку; иначе psql прекратит работу с кодом возврата 3, чтобы отличить этот случай от критических ошибок, для которых используется код возврата 1. В любом случае выполнение всех запущенных скриптов (высокоуровневый скрипт и любые другие, которые он мог запустить) будет немедленно прекращено. Если высокоуровневая командная строка содержит несколько SQL-команд, выполнение завершится на текущей команде.

`PORT`

Содержит порт сервера базы данных, к которому вы сейчас подключены. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно изменить или сбросить.

`PROMPT1`
`PROMPT2`
`PROMPT3`

Указывают, как должны выглядеть приглашения psql. См. раздел Настройка приглашений ниже.

`QUIET`

Установка значения `on` эквивалента параметру командной строки `-q`. Это, вероятно, не слишком полезно в интерактивном режиме.

`ROW_COUNT`

Число строк, возвращённых или обработанных последним SQL-запросом, либо 0, если запрос завершился неудачно или не возвратил количество строк.

`SERVER_VERSION_NAME`
`SERVER_VERSION_NUM`

Номер версии сервера в виде строки, например `9.6.2`, `10.1` или `11beta1`, и в числовом виде, например, `90602` или `100001`. Они устанавливаются при каждом подключении к базе данных (в том числе при запуске программы), но их можно изменить или сбросить.

`SHELL_ERROR`

`true`, если последняя команда оболочки завершилась ошибкой, `false` — если она была выполнена успешно. Устанавливается для команд оболочки, вызываемых метакомандами `\!`, `\g`, `\o`, `\w` и `\copy`, а также командами в обратных кавычках (`` ` ``). Обратите внимание, что для `\o` эта переменная обновляется, когда канал вывода закрывается следующей командой `\o`. См. также `SHELL_EXIT_CODE`.

`SHELL_EXIT_CODE`

Код завершения, возвращаемый последней командой оболочки. 0–127 представляют собой коды завершения программы, 128–255 указывают на завершение работы по сигналу, -1 указывает, что невозможно запустить программу или получить код завершения. Устанавливается для команд оболочки, вызываемых метакомандами `\!`, `\g`, `\o`, `\w` и `\copy`, а также командами в обратных кавычках (`` ` ``). Обратите внимание, что для `\o` эта переменная обновляется, когда канал вывода закрывается следующей командой `\o`. См. также `SHELL_ERROR`.

`SHOW_ALL_RESULTS`

Когда эта переменная имеет значение `off` (выкл.), выводятся не все результаты объединённого (с использованием `\;`) запроса, а только последний. Это поведение реализовано для совместимости со старыми версиями psql. По умолчанию `on` (вкл.).

`SHOW_CONTEXT`

Этой переменной можно присвоить значения `never` (никогда), `errors` (ошибки) или `always` (всегда), определяющие, когда в сообщениях с сервера будут выводиться поля `КОНТЕКСТ`. По умолчанию выбран вариант `errors` (который означает, что контекст будет выводиться в сообщениях об ошибках, но не в предупреждениях и уведомлениях). Этот параметр не действует, когда установлен уровень `VERBOSITY terse` или `sqlstate`. (Когда вам потребуется подробная версия только что выданной ошибки, может быть полезна команда `\errverbose`.)

`SINGLELINE`

Установка значения `on` эквивалентна параметру командной строки `-S`.

`SINGLESTEP`

Эта переменная эквивалентна параметру командной строки `-s`.

`SQLSTATE`

Код ошибки (см. Приложение A), связанной с неудачным выполнением последнего SQL-запроса, либо `00000` в случае его успешного завершения.

`USER`

Содержит имя пользователя базы данных, который сейчас подключён. Устанавливается всякий раз при подключении к базе данных (в том числе при старте программы), но эту переменную можно изменить или сбросить.

`VERBOSITY`

Этой переменной можно присвоить значения `default`, `verbose`, `terse` или `sqlstate` для изменения уровня детализации в сообщениях об ошибках. (См. также команду `\errverbose`, полезную, когда требуется подробная версия только что выданной ошибки.)

`VERSION`
`VERSION_NAME`
`VERSION_NUM`

Эти переменные устанавливаются при запуске программы и отражают версию psql соответственно в виде развёрнутой строки, краткой строки (например, `9.6.2`, `10.1` или `11beta1`) и числа (например, `90602` или `100001`). Их можно изменить или сбросить.

Интерполяция SQL

Ключевой особенностью переменных psql является возможность подставлять («интерполировать») их в команды SQL, так же как и в аргументы метакоманд. Кроме того, psql предоставляет средства для корректного использования кавычек для значений переменных, которые используются как литералы или идентификаторы SQL. Чтобы подставить значение без кавычек, нужно добавить перед именем переменной двоеточие (`:`). Например:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

будет запрашивать таблицу `my_table`. Обратите внимание, что это может быть небезопасным: значение переменной копируется буквально, поэтому оно может содержать непарные кавычки или даже метакоманды. При применении необходимо убедиться, что это имеет смысл.

Когда значение будет использоваться в качестве SQL литерала или идентификатора, безопаснее заключить его в кавычки. Если значение переменной используется как SQL литерал, то после двоеточия нужно написать имя переменной в одинарных кавычках. Если значение переменной используется как SQL идентификатор, то после двоеточия нужно написать имя переменной в двойных кавычках. Эти конструкции корректно работают с кавычками и другими специальными символами, которые могут содержаться в значении переменной. Предыдущий пример более безопасно выглядит так:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

Подстановка переменных не будет выполняться, если SQL литералы или идентификаторы заключены в кавычки. Поэтому конструкция `':foo'` не превратится во взятое в кавычки значение переменной (и это было бы небезопасно, если бы работало, так как обработка кавычек внутри значения переменной была бы некорректной).

Один из примеров использования данного механизма — это копирование содержимого файла в столбец таблицы. Сначала загрузим содержимое файла в переменную, затем подставим значение переменной как строку в кавычках:

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

(Отметим, что это пока не будет работать, если `my_file.txt` содержит байт NUL. psql не поддерживает NUL в значениях переменных.)

Так как двоеточие может легально присутствовать в SQL-командах, попытка подстановки (например для `:name`, `:'name'` или `:"name"`) не выполняется, если переменная не установлена. В любом случае можно экранировать двоеточие с помощью обратной косой черты, чтобы предотвратить подстановку.

Специальная конструкция `:{?`*имя*`}` возвращает TRUE или FALSE в зависимости от того, существует ли переменная, и таким образом всегда подменяется значением, если только двоеточие не экранировано обратной косой чертой.

Использование двоеточия для переменных является стандартом SQL для встраиваемых языков запросов, таких как ECPG. Использование двоеточия для срезов массивов и приведения типов является расширениями Postgres Pro, что иногда может конфликтовать со стандартным использованием. Использование двоеточия и кавычек для экранирования значения переменной при подстановке в качестве SQL литерала или идентификатора — это расширение psql.

Настройка приглашений

Приглашения, выдаваемые psql, можно настроить по своему вкусу. Три переменные `PROMPT1`, `PROMPT2` и `PROMPT3` содержат строки и спецпоследовательности, задающие внешний вид приглашения. Приглашение 1 (`PROMPT1`) — это обычное приглашение, которое выдаётся, когда psql ожидает ввода новой команды. Приглашение 2 (`PROMPT2`) выдаётся, когда при вводе команды ожидается дополнительные строки, например потому что команда не была завершена точкой с запятой или не закрыты кавычки. Приглашение 3 (`PROMPT3`) выдаётся при выполнении SQL-команды `COPY FROM STDIN`, когда в терминале нужно ввести значение новой строки.

Значения этих переменных выводятся буквально, за исключением случаев, когда в них встречается знак процента (`%`). В зависимости от следующего символа будет подставляться определённый текст. Существуют следующие подстановки:

`%M`

Полное имя компьютера (с именем домена) сервера базы данных или `[local]`, если подключение выполнено через Unix-сокет, либо `[local:`*/каталог/имя*`]`, если при компиляции был изменён путь Unix-сокета по умолчанию.

`%m`

Имя компьютера, где работает сервер баз данных, усечённое до первой точки или `[local]`, если подключение выполнено через Unix-сокет.

`%>`

Номер порта, который прослушивает сервер базы данных.

`%n`

Имя пользователя базы данных для текущего сеанса. (Это значение может меняться в течение сеанса в результате выполнения команды `SET SESSION AUTHORIZATION`.)

`%/`

Имя текущей базы данных.

`%~`

Похоже на `%/`, но выводит тильду `~`, если текущая база данных совпадает с базой данных по умолчанию.

`%#`

Если пользователь текущего сеанса является суперпользователем базы данных, то выводит `#`, иначе `>`. (Это значение может меняться в течение сеанса в результате выполнения команды `SET SESSION AUTHORIZATION`.)

`%p`

PID обслуживающего процесса для текущего подключения.

`%R`

В приглашении 1 это обычно символ `=`, но `@`, если сеанс находится в неактивной ветви блока условия, либо `^` в однострочном режиме либо `!`, если сеанс не подключён к базе данных (что возможно при ошибке `\connect`). В приглашении 2 `%R` заменяется символом, показывающим, почему psql ожидает дополнительный ввод: `-`, если команда просто ещё не была завершена, но `*`, если не завершён комментарий `/* ... */`; апостроф, если не завершена строка в апострофах; кавычки, если не завершён идентификатор в кавычках; знак доллара, если не завершена строка в долларах; либо `(`, если после открывающей скобки не хватает закрывающей. В приглашении 3 `%R` не выдаёт ничего.

`%x`

Состояние транзакции: пустая строка, если не в транзакционном блоке; `*`, когда в транзакционном блоке; `!`, когда в транзакционном блоке, в котором произошла ошибка и `?`, когда состояние транзакции не определено (например, нет подключения к базе данных).

`%l`

Номер строки в текущем операторе, начиная с `1`.

`%`*цифры*

Подставляется символ с указанным восьмеричным кодом.

`%:`*имя*`:`

Значение переменной psql *имя*. За подробностями обратитесь к разделу Переменные выше.

`` %` ``*команда*`` ` ``

Подставляется вывод *команды*, как и в обычной подстановке с обратными апострофами.

`%[` ... `%]`

Приглашения могут содержать управляющие символы терминала, которые, например, изменяют цвет, фон и стиль текста приглашения или изменяют заголовок окна терминала. Для того, чтобы возможности редактирования Readline работали правильно, непечатаемые символы нужно расположить между `%[` и `%]`, чтобы сделать невидимыми. Можно делать несколько таких включений в приглашение. Например:

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

выдаст жирное (`1;`), желтое на черном (`33;40`) приглашение для VT100 совместимых цветных терминалов.

`%w`

Пробелы, дающие тот же отступ, что и в выданном последним приглашении `PROMPT1`. Это значение можно использовать в `PROMPT2`, чтобы многострочные операторы были выровнены по первой строке, но при этом вторичного приглашения не было видно.

Чтобы вставить знак процента, нужно написать `%%`. По умолчанию для `PROMPT1` и `PROMPT2` используется значение `'%/%R%x%# '`, а для `PROMPT3` — `'>> '`.

Примечание

Эта функциональность была бессовестно списана с tcsh.

Редактирование командной строки

Программа psql использует библиотеку Readline или libedit, если она доступна, для удобства редактирования текущей строки команд и для возврата к предыдущим. История команд автоматически сохраняется при выходе из psql и загружается при запуске. Вызвать предыдущие строки из истории можно, нажимая клавишу «стрелка вверх» или Control-P.

Вы также можете использовать дополнение табуляцией для завершения частично введённых ключевых слов и имён объектов SQL во многих (но не во всех) контекстах. Например, если в начале команды ввести `ins` и нажать `TAB`, вы получите `insert into`. Если затем набрать несколько символов имени таблицы или схемы и нажать клавишу `TAB`, неоконченное имя будет дополнено либо будет предложен список возможных вариантов дополнения, если их несколько. (В зависимости от используемой библиотеки может потребоваться нажать `TAB` более одного раза, чтобы вызвать этот список.)

Чтобы дополнение табуляцией работало для имён объектов SQL, необходимо передавать серверу запросы для получения возможных вариантов. В некоторых контекстах это может мешать другим операциям. Например, выполнять после `BEGIN` команду `SET TRANSACTION ISOLATION LEVEL` будет поздно, если между ними вклинится запрос, выполняемый для дополнения табуляцией. Если вам вообще не нужно дополнение табуляцией, вы можете отключить его навсегда, добавив следующие строки в файл с именем `.inputrc` в вашем домашнем каталоге:

```
$if psql
set disable-completion on
$endif
```

(Это возможность Readline, а не psql. За подробностями обратитесь к документации Readline.)

Чтобы отказаться от использования Readline в текущем сеансе psql, также можно воспользоваться параметром командной строки `-n` (`--no-readline`). Этот параметр отключает дополнение табуляцией, использование и запись истории команд, а также редактирование многострочных команд. Это может быть полезно, когда нужно скопировать и вставить текст, содержащий символы `TAB`.

Переменные окружения

`COLUMNS`

Если `\pset columns` равно нулю, управляет шириной формата вывода `wrapped`, а также определяет, нужно ли использовать постраничник и нужно ли переключаться в вертикальный формат в режиме `expanded auto`.

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию (см. Раздел 32.15).

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

Редактор, используемый командами `\e`, `\ef` и `\ev`. Эти переменные рассматриваются в том же порядке; в силу вступает первое установленное значение. Если ни одна из переменных не установлена, по умолчанию в Unix-системах используется `vi`, а в Windows — `notepad.exe`.

`PSQL_EDITOR_LINENUMBER_ARG`

Если в командах `\e`, `\ef` или `\ev` указан номер строки, эта переменная задаёт аргумент командной строки, с которым номер строки может быть передан в редактор. Например, для редакторов Emacs и vi это знак плюс. Добавьте в конец значения пробел, если он требуется для отделения имени аргумента от номера строки. Примеры:

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

Значение по умолчанию `+` в Unix-подобных системах (соответствует редактору по умолчанию `vi` и многим другим распространённым редакторам). На платформе Windows нет значения по умолчанию.

`PSQL_HISTORY`

Альтернативное расположение файла с историей команд. Допускается использование тильды (`~`).

`PSQL_PAGER`
`PAGER`

Если результат запроса не помещается на экране, он пропускается через эту программу. Обычно это `more` или `less`. От использования постраничника можно отказаться, присвоив переменной `PSQL_PAGER` или `PAGER` пустую строку, либо изменив соответствующие параметры с помощью команды `\pset`. Данные переменные просматриваются в этом же порядке; используется первая установленная. Если не установлена ни одна из переменных, в большинстве платформ используется `more`, а в Cygwin — `less`.

`PSQL_WATCH_PAGER`

Когда команда `\watch` повторяет один и тот же запрос, постраничник по умолчанию не используется. Это поведение можно изменить, указав в `PSQL_WATCH_PAGER` команду для вызова постраничника (в системах Unix). Постраничник `pspg` (не являющийся частью PostgreSQL, но имеющийся во многих дистрибутивах открытого программного обеспечения) может отображать вывод `\watch`, если передать ему ключ `--stream`.

`PSQLRC`

Альтернативное расположение пользовательского файла `.psqlrc`. Допускается использование тильды (`~`).

`SHELL`

Команда операционной системы, выполняемая метакомандой `\!`.

`TMPDIR`

Каталог для хранения временных файлов. По умолчанию `/tmp`.

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Файлы

`psqlrc` и `~/.psqlrc`

При запуске без параметра `-X` программа psql пытается считать и выполнить команды из общесистемного стартового файла (`psqlrc`), а затем из персонального стартового файла пользователя (`~/.psqlrc`), после подключения к базе данных, но перед получением обычных команд. Этими файлами можно воспользоваться для настройки клиента и/или сервера, как правило, с помощью команд `\set` и `SET`.

Общесистемный стартовый файл называется `psqlrc`, по умолчанию он будет искаться в каталоге установки «конфигурация системы». Для того чтобы узнать этот каталог, надёжнее всего выполнить команду `pg_config --sysconfdir`. Обычно он расположен в `../etc/` относительно каталога, содержащего исполняемые файлы Postgres Pro. Путь к этому каталогу также можно задать явно в переменной окружения `PGSYSCONFDIR`.

Персональный стартовый файл пользователя называется `.psqlrc`, он будет искаться в домашнем каталоге вызывающего пользователя. В Windows путь к персональному стартовому файлу — `%APPDATA%\postgresql\psqlrc.conf`. В любом случае подразумеваемый по умолчанию путь можно переопределить, установив переменную окружения `PSQLRC`.

Оба стартовых файла, общесистемный и персональный, можно привязать к конкретной версии psql, добавив минус и идентификатор основной или дополнительной версии Postgres Pro к имени файла, например: `~/.psqlrc-17` или `~/.psqlrc-17.6.1`. При совпадении нескольких указаний с текущей версией psql приоритетнее будет файл с более полным указанием. Суффиксы версий при поиске файла добавляются после того, как описанным выше образом будет определён путь к файлу.

`.psql_history`

История командной строки хранится в файле `~/.psql_history` или `%APPDATA%\postgresql\psql_history` на Windows.

Расположение файла истории можно задать явно через переменную psql `HISTFILE` или через переменную окружения `PSQL_HISTORY`.

Примечания

- psql лучше всего работает с серверами той же или более старой основной версии. Сбой метакоманды наиболее вероятен, если версия сервера новее, чем версия psql. Однако команды семейства `\d` должны работать с версиями сервера до 9.2, хотя и необязательно с серверами новее, чем сам psql. Общая функциональность запуска SQL-команд и отображения результатов запросов также должна работать на серверах с более новой основной версией, но это не гарантируется во всех случаях.

  Если вы хотите, применяя psql, подключаться к нескольким серверам с различными основными версиями, рекомендуется использовать последнюю версию psql. Также можно собрать копии psql от каждой основной версии и использовать ту, которая соответствует версии сервера. Но на практике в этих дополнительных сложностях нет необходимости.
- В Postgres Pro до версии 9.6 параметр `-c` подразумевал `-X` (`--no-psqlrc`); теперь это не так.
- В PostgreSQL до 8.4 программа psql могла принять первый аргумент однобуквенной команды с обратной косой чертой сразу после команды, без промежуточного пробела. Теперь разделительный пробельный символ обязателен.

Замечания для пользователей Windows

psql создан как «консольное приложение». Поскольку в Windows консольные окна используют кодировку символов отличную от той, что используется для остальной системы, нужно проявить особую осторожность при использовании 8-битных символов. Если psql обнаружит проблемную кодовую страницу консоли, он предупредит вас при запуске. Чтобы изменить кодовую страницу консоли, необходимы две вещи:

- Задать кодовую страницу, выполнив **`cmd.exe /c chcp 1251`**. (1251 это кодовая страница для России, замените на ваше значение.) При использовании Cygwin, эту команду можно записать в `/etc/profile`.
- Установите консольный шрифт в `Lucida Console`, потому что растровый шрифт не работает с кодовой страницей ANSI.

По умолчанию psql работает в кодировке UTF-8 и использует для вывода в консоли API Windows Unicode. Чтобы все символы, поддерживаемые вашим консольным шрифтом Windows отображались корректно, необходимо установить для этой консоли кодовую страницу 65001.

Инсталлятор Postgres Pro для Windows устанавливает постраничник `less.exe` с поддержкой UTF-8 и создаёт ярлык, открывающий консольное окно со шрифтом `Lucida Console` и кодовой страницей 65001. Если вы используете другой постраничник, убедитесь в том, что он тоже поддерживает UTF-8.

Вы можете переопределить кодировку, выбранную в psql по умолчанию, установив переменную окружения `PGCLIENTENCODING`.

Примеры

Первый пример показывает, что для ввода одной команды может потребоваться несколько строк. Обратите внимание, как меняется приглашение:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

Теперь посмотрим на определение таблицы:

```
testdb=> \d my_table
             Таблица "public.my_table"
 Столбец |   Тип   | Правило сортировки | Допустимость NULL | По умолчанию
 first   | integer |                    | not null          |            0
 second  | text    |                    |                   |
```

Теперь изменим приглашение на что-то более интересное:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Предположим, что вы внесли данные в таблицу и хотите на них посмотреть:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | Один
     2 | Два
     3 | Три
     4 | Четыре
(4 строки)
```

Таблицу можно вывести разными способами при помощи команды `\pset`:

```
peter@localhost testdb=> \pset border 2
Установлен стиль границ 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
```

```
+-------+--------+
|     1 | Один   |
|     2 | Два    |
|     3 | Три    |
|     4 | Четыре |
+-------+--------+
(4 строки)

peter@localhost testdb=> \pset border 0
Установлен стиль границ 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 один
    2 два
    3 три
    4 четыре
(4 строки)

peter@localhost testdb=> \pset border 1
Установлен стиль границ 1.
peter@localhost testdb=> \pset format csv
Формат вывода: csv.
peter@localhost testdb=> \pset tuples_only
Режим вывода только кортежей включён.
peter@localhost testdb=> SELECT second, first FROM my_table;
один,1
два,2
три,3
четыре,4
peter@localhost testdb=> \pset format unaligned
Формат вывода: unaligned.
peter@localhost testdb=> \pset fieldsep '\t'
Разделитель полей: "    ".
peter@localhost testdb=> SELECT second, first FROM my_table;
один    1
два     2
три     3
четыре  4
```

Также можно использовать короткие команды:

```
peter@localhost testdb=> \a \t \x
Формат вывода: aligned.
Режим вывода только кортежей выключен.
Расширенный вывод включён.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | Один
-[ RECORD 2 ]-
first  | 2
second | Два
-[ RECORD 3 ]-
first  | 3
second | Три
-[ RECORD 4 ]-
first  | 4
second | Четыре
```

Кроме того, эти параметры формата можно задать только для одного запроса, выполняя команду `\g`:

```
peter@localhost testdb=> SELECT * FROM my_table
peter@localhost testdb-> \g (format=aligned tuples_only=off expanded=on)
-[ RECORD 1 ]-
first  | 1
second | Один
-[ RECORD 2 ]-
first  | 2
second | Два
-[ RECORD 3 ]-
first  | 3
second | Три
-[ RECORD 4 ]-
first  | 4
second | Четыре
```

Пример использования команды `\df` для получения только функций, имя которых соответствует шаблону `int*pl`, а второй аргумент имеет тип `bigint`:

```
testdb=> \df int*pl * bigint
                          Список функций
   Схема    |   Имя   | Тип данных результата | Типы данных аргументов | Тип
------------+---------+-----------------------+------------------------+------
 pg_catalog | int28pl | bigint                | smallint, bigint       | func
 pg_catalog | int48pl | bigint                | integer, bigint        | func
 pg_catalog | int8pl  | bigint                | bigint, bigint         | func
(3 строки)
```

Когда это уместно, результаты запроса можно просмотреть в виде перекрёстной таблицы с помощью команды `\crosstabview`:

```
testdb=> SELECT first, second, first > 2 AS gt2 FROM my_table;
 first | second | gt2
-------+--------+-----
     1 | one    | f
     2 | two    | f
     3 | three  | t
     4 | four   | t
(4 rows)

testdb=> \crosstabview first second
 first | one | two | three | four
-------+-----+-----+-------+------
     1 | f   |     |       |
     2 |     | f   |       |
     3 |     |     | t     |
     4 |     |     |       | t
(4 rows)
```

Второй пример показывает таблицу умножения, строки в которой отсортированы в обратном числовом порядке, а столбцы — независимо, по возрастанию числовых значений.

```
testdb=> SELECT t1.first as "A", t2.first+100 AS "B", t1.first*(t2.first+100) as "AxB",
testdb-> row_number() over(order by t2.first) AS ord
testdb-> FROM my_table t1 CROSS JOIN my_table t2 ORDER BY 1 DESC
testdb-> \crosstabview "A" "B" "AxB" ord
 A | 101 | 102 | 103 | 104
---+-----+-----+-----+-----
 4 | 404 | 408 | 412 | 416
```

```
  3 | 303 | 306 | 309 | 312
  2 | 202 | 204 | 206 | 208
  1 | 101 | 102 | 103 | 104
(4 rows)
```

reindexdb

reindexdb — переиндексировать базу данных Postgres Pro

Синтаксис

`reindexdb` [*`параметр-подключения`*...] [*`параметр`*...] [`-S` | `--schema` *`схема`*] ... [`-t` | `--table` *`таблица`*] ... [`-i` | `--index` *`индекс`*] ... [`-s` | `--system`] [*`dbname`* | `-a` | `--all`]

Описание

Утилита reindexdb предназначена для перестроения индексов в базе данных Postgres Pro.

Утилита reindexdb представляет собой обёртку SQL-команды `REINDEX`. Переиндексация базы данных с её помощью по сути не отличается от переиндексации при обращении к серверу другими способами.

Параметры

reindexdb принимает следующие аргументы командной строки:

`-a`
`--all`

Переиндексировать все базы данных.

`--concurrently`

Использовать режим `CONCURRENTLY`. Все особенности этого режима подробно описаны в REINDEX.

`[-d]` *`dbname`*
`[--dbname=]`*`dbname`*

Указывает имя базы данных для переиндексации, когда не используется параметр `-a`/`--all`. Если это указание отсутствует, имя базы определяется переменной окружения `PGDATABASE`. Если эта переменная не установлена, именем базы будет имя пользователя, указанное для подключения. В аргументе *`имя_бд`* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`-e`
`--echo`

Выводить команды, которые reindexdb генерирует и передаёт серверу.

`-i` *`индекс`*
`--index=`*`индекс`*

Пересоздать только указанный *`индекс`*. Добавив дополнительные ключи `-i`, можно пересоздать несколько индексов.

`-j` *`число_заданий`*
`--jobs=`*`число_заданий`*

Выполнять команды переиндексации в параллельном режиме, запуская их одновременно в количестве *`число_заданий`*. Это может сократить время обработки, но при этом увеличить нагрузку на сервер.

reindexdb будет устанавливать несколько подключений к базе данных (в количестве *`число_заданий`*), так что убедитесь в том, что значение max_connections достаточно велико, чтобы все эти подключения были приняты.

Заметьте, что этот параметр несовместим с параметром `--system`.

`-q`
`--quiet`

Подавлять вывод сообщений о прогрессе выполнения.

`-s`
`--system`

Переиндексировать только системные каталоги базы данных.

`-S` *`схема`*
`--schema=`*`схема`*

Переиндексировать только указанную *`схему`*. Переиндексировать несколько схем можно, добавив несколько ключей `-S`.

`-t` *`таблица`*
`--table=`*`таблица`*

Переиндексировать только указанную *`таблицу`*. Переиндексировать несколько таблиц можно, добавив несколько ключей `-t`.

`--tablespace=`*`табличное_пространство`*

Указывает табличное пространство, в котором перестраиваются индексы. (Это имя обрабатывается как идентификатор в двойных кавычках.)

`-v`
`--verbose`

Вывести подробную информацию во время процесса.

`-V`
`--version`

Сообщить версию reindexdb и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки reindexdb и завершиться.

Утилита reindexdb также принимает следующие аргументы командной строки в качестве параметров подключения:

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как reindexdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, reindexdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*`dbname`*

Указывает имя базы данных, к которой будет выполняться подключение для определения подлежащих переиндексации баз данных, когда используется ключ `-a`/`--all`. Если это имя не указано, будет выбрана база `postgres`, а если она не существует — `template1`. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Диагностика

В случае возникновения трудностей, обратитесь к описаниям REINDEX и psql, где обсуждаются потенциальные проблемы и сообщения об ошибках. Учтите, что на целевом компьютере должен работать сервер баз данных. При этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека libpq.

Примеры

Переиндексирование базы данных `test`:

```
$ reindexdb test
```

Переиндексирование таблицы `foo` и индекса `bar` в базе данных `abcd`:

```
$ reindexdb --table=foo --index=bar abcd
```

См. также

REINDEX

vacuumdb

vacuumdb — выполнить очистку и анализ базы данных Postgres Pro

Синтаксис

`vacuumdb` [*параметр-подключения*...] [*параметр*...] [`-t` | `--table` *таблица* [(*столбец* [,...])]] ... [*dbname* | `-a` | `--all`]

`vacuumdb` [*параметр-подключения*...] [*параметр*...] [`-n` | `--schema` *схема*] ... [*dbname* | `-a` | `--all`]

`vacuumdb` [*параметр-подключения*...] [*параметр*...] [`-N` | `--exclude-schema` *схема*] ... [*dbname* | `-a` | `--all`]

Описание

Утилита vacuumdb предназначена для очистки базы данных Postgres Pro. Кроме того, vacuumdb генерирует внутреннюю статистику, которую использует оптимизатор запросов Postgres Pro.

Утилита vacuumdb представляют собой обёртку SQL-команды `VACUUM`. Выполнение очистки и анализа баз данных с её помощью по сути не отличается от выполнения тех же действий при обращении к серверу другими способами.

Параметры

Утилита vacuumdb принимает следующие аргументы командной строки:

`-a`
`--all`

Очистить все базы данных.

`--buffer-usage-limit` *размер*

Указывает размер кольцевого буфера в *стратегии доступа к буферу* для заданного вызова vacuumdb. Указанный размер используется для вычисления числа общих буферов, повторно используемых в рамках стратегии. См. VACUUM.

`[-d]` *dbname*
`[--dbname=]`*dbname*

Указывает имя базы данных для очистки или анализа, когда не используется параметр `-a`/`--all`. Если это указание отсутствует, имя базы определяется переменной окружения `PGDATABASE`. Если эта переменная не задана, именем базы будет имя пользователя, указанное для подключения. В аргументе *имя_бд* может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке.

`--disable-page-skipping`

Запретить пропуск страниц в зависимости от содержимого карты видимости.

`-e`
`--echo`

Выводить команды, которые vacuumdb генерирует и передаёт серверу.

`-f`
`--full`

Произвести «полную» очистку.

`-F`
`--freeze`

Агрессивно «замораживать» версии строк.

`--force-index-cleanup`

Всегда удалять элементы индекса, указывающие на мёртвые кортежи.

`-j` *`число_заданий`*
`--jobs=`*`число_заданий`*

Выполнять команды очистки и анализа в параллельном режиме, запуская их одновременно в количестве *`число_заданий`*. Это может сократить время обработки, но при этом увеличить нагрузку на сервер.

vacuumdb будет устанавливать несколько подключений к базе данных (в количестве *`число_заданий`*), так что убедитесь в том, что значение max_connections достаточно велико, чтобы все эти подключения были приняты.

Заметьте, что использование этого режима с параметром `-f` (`FULL`) может привести к отказам из-за взаимоблокировок, если параллельно начнут обрабатываться определённые системные каталоги.

`--min-mxid-age` *`возраст_мультитранзакции`*

Выполнять команды очистки и анализа только для таблиц, имеющих не менее чем заданный *`возраст_мультитранзакции`*. Этот параметр полезен для выбора таблиц, первоочередная обработка которых поможет предотвратить зацикливание идентификаторов мультитранзакций (см. Подраздел 23.1.5.1).

Применительно к данному параметру возрастом мультитранзакции для отношения считается наибольший из возрастов основного отношения и связанной с ним таблицы TOAST, если она существует. Так как команды, выполняемые утилитой vacuumdb, будут при необходимости обрабатывать не только отношение, но и таблицу TOAST, связанную с ним, рассматривать их возрасты по отдельности не имеет смысла.

`--min-xid-age` *`возраст_транзакции`*

Выполнять команды очистки и анализа только для таблиц, имеющих не менее чем заданный *`возраст_транзакции`*. Этот параметр полезен для выбора таблиц, обработка которых в первую очередь поможет предотвратить зацикливание идентификаторов транзакций (см. Подраздел 23.1.5).

Применительно к данному параметру возрастом транзакции для отношения считается наибольший из возрастов основного отношения и связанной с ним таблицы TOAST, если она существует. Так как команды, выполняемые утилитой vacuumdb, будут при необходимости обрабатывать не только отношение, но и таблицу TOAST, связанную с ним, рассматривать их возрасты по отдельности не имеет смысла.

`-n` *`схема`*
`--schema=`*`схема`*

Производить очистку или анализ всех таблиц только в одной *`схеме`*. Добавив дополнительные ключи `-n`, можно очистить таблицы в нескольких схемах.

`-N` *`схема`*
`--exclude-schema=`*`схема`*

Не производить очистку или анализ каких-либо таблиц в указанной *`схеме`*. Добавив дополнительные ключи `-N`, можно исключить из очистки таблицы в нескольких схемах.

`--no-index-cleanup`

Не удалять элементы индекса, указывающие на мёртвые кортежи.

`--no-process-main`

Пропускать основное отношение.

`--no-process-toast`

Пропускать TOAST-таблицу, связанную с очищаемой таблицей (при наличии).

`--no-truncate`

Не отсекать пустые страницы в конце таблицы.

`-P` *`параллельные_исполнители`*
`--parallel=`*`параллельные_исполнители`*

Задаёт количество параллельных исполнителей для *параллельной очистки*. Это позволяет в ходе очистки задействовать мощности нескольких процессоров для обработки индексов. См. VACUUM.

`-q`
`--quiet`

Подавлять вывод сообщений о прогрессе выполнения.

`--skip-locked`

Пропускать отношения, которые не удаётся немедленно заблокировать для обработки.

`-t` *`таблица`* `[ (`*`столбец`* `[,...]) ]`
`--table=`*`таблица`* `[ (`*`столбец`* `[,...]) ]`

Производить очистку или анализ только указанной *`таблицы`*. Имена столбцов можно указать только в сочетании с параметрами `--analyze` и `--analyze-only`. Добавив дополнительные ключи `-t`, можно обработать несколько таблиц.

> ### Подсказка
>
> Если вы указываете столбцы, вам, вероятно, придётся экранировать скобки в оболочке. (См. примеры ниже.)

`-v`
`--verbose`

Вывести подробную информацию во время процесса.

`-V`
`--version`

Сообщить версию vacuumdb и завершиться.

`-z`
`--analyze`

Также вычислить статистику для оптимизатора.

`-Z`
`--analyze-only`

Только вычислить статистику для оптимизатора (не производить очистку).

`--analyze-in-stages`

Только вычислить статистику для оптимизатора (без очистки), подобно `--analyze-only`. При этом анализ выполняется в три этапа: на первом этапе для скорейшего получения полезной статистики используется минимально возможный ориентир статистики (см. default_statistics_target), а на последующих этапах вычисляется полная статистика.

Этот параметр полезен, только когда нужно провести анализ базы данных, статистика в которой отсутствует или абсолютно неактуальна, например когда база была только что наполнена данными из копии или в результате процедуры `pg_upgrade`. Имейте в виду, что использование этого параметра для базы, в которой уже есть статистика, может привести к тому, что качество решений оптимизатора может временно ухудшиться из-за небольших ориентиров статистики на ранних этапах.

`-?`
`--help`

Показать справку по аргументам командной строки vacuumdb и завершиться.

Утилита vacuumdb также принимает следующие аргументы командной строки в качестве параметров подключения:

`-h` *сервер*
`--host=`*сервер*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *порт*
`--port=`*порт*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *имя_пользователя*
`--username=`*имя_пользователя*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как vacuumdb запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, vacuumdb лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

`--maintenance-db=`*dbname*

Указывает имя базы данных, к которой будет выполняться подключение для определения подлежащих очистке баз данных, когда используется ключ `-a`/`--all`. Если это имя не указано, будет выбрана база `postgres`, а если она не существует — `template1`. В данном аргументе может задаваться строка подключения. В этом случае параметры в строке подключения переопределяют одноимённые параметры, заданные в командной строке. Кроме того, все параметры в строке подключения, за исключением имени базы, будут использоваться и при подключении к другим базам данных.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Диагностика

В случае возникновения трудностей, обратитесь к описаниям VACUUM и psql, где обсуждаются потенциальные проблемы и сообщения об ошибках. Учтите, что на целевом компьютере должен работать сервер баз данных. При этом применяются все свойства подключения по умолчанию и переменные окружения, которые использует клиентская библиотека libpq.

Примеры

Очистка базы данных `test`:

```
$ vacuumdb test
```

Очистка и анализ для оптимизатора базы данных `bigdb`:

```
$ vacuumdb --analyze bigdb
```

Очистка одной таблицы `foo` в базе данных `xyzzy` и анализ только столбца `bar` таблицы для оптимизатора:

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzzy
```

Очистка всех таблиц в схемах `foo` и `bar` в базе данных `xyzzy`:

```
$ vacuumdb --schema='foo' --schema='bar' xyzzy
```

См. также

VACUUM

Серверные приложения Postgres Pro

В этой части содержится справочная информация о серверных приложениях и вспомогательных утилитах Postgres Pro. Описываемые команды могут быть полезны только на том компьютере, где работает сервер баз данных. Другие утилиты рассмотрены в Справке: «Клиентские приложения Postgres Pro».

initdb

initdb — создать кластер баз данных Postgres Pro

Синтаксис

`initdb` [*параметр*...] [`--pgdata` | `-D`]*каталог*

Описание

Команда `initdb` создаёт новый *кластер баз данных* Postgres Pro.

Создание кластера баз данных заключается в создании *каталогов* для хранения данных кластера, формировании общих системных таблиц (относящихся ко всему кластеру, а не к какой-либо базе) и создании баз данных `postgres`, `template1` и `template0`. База `postgres` используется пользователями, утилитами и сторонними приложениями по умолчанию. Базы `template1` и `template0` предназначены для использования в качестве исходных баз данных, копии которых будут создавать последующие команды `CREATE DATABASE`. Базу `template0` не следует изменять никогда, но вы можете добавлять объекты в `template1`, и они по умолчанию будут копироваться в базы данных, создаваемые позже. За подробностями обратитесь к Разделу 21.3.

При попытке создать каталог для хранения данных `initdb` может столкнуться с нехваткой прав доступа, если этот каталог принадлежит суперпользователю root. В таком случае необходимо назначить пользователя базы данных владельцем этого каталога при помощи `chown`. Затем выполнить `su` для смены пользователя и дальнейшего выполнения `initdb`.

Команда `initdb` должна выполняться от имени пользователя, под которым будет запускаться сервер, так как ему необходим полный доступ к файлам и каталогам, создаваемым `initdb`. Сервер не может запускаться от имени суперпользователя, поэтому выполнение команды `initdb` от его лица будет отклонено.

Из соображений безопасности новый кластер, созданный программой `initdb`, будет доступен только для владельца кластера. Ключ `--allow-group-access` позволяет разрешить чтение файлов в кластере всем пользователям, входящим в группу владельца кластера. Это полезно для выполнения резервного копирования от имени непривилегированного пользователя.

`initdb` инициализирует локали и кодировки баз данных кластера, которые будут использоваться по умолчанию. Они могут устанавливаться раздельно при создании новой базы данных. `initdb` определяет параметры локали для шаблонных баз данных, применяющиеся по умолчанию для новых баз.

По умолчанию `initdb` получает параметры локали из переменных среды и определяет кодировку, исходя из параметров локали. Если провайдер локалей не задан, он выбирается автоматически на основании значения `lc_collate` для кластера: для локалей `C` и `POSIX` выбирается `libc`, а для других локалей — `icu`.

Выбрать другую локаль для кластера позволяет параметр `--locale`. Существуют также отдельные параметры `--lc-*` и `--icu-locale` (см. ниже), позволяющие задать значения для отдельных категорий локали. Учтите, что с неподходящими значениями для разных категорий локали результаты могут быть бессмысленными, поэтому эти значения следует выбирать продуманно.

В качестве библиотеки, реализующей поддержку локалей, `initdb` может также использовать библиотеку ICU, если указать `--locale-provider=icu`. При этом сервер должен быть собран с поддержкой ICU. Выбрать определённую локаль ICU позволяет параметр `--icu-locale`. Заметьте, что даже когда используется провайдер локалей ICU, `initdb` в любом случае выбирает и инициализирует локали libc в связи с особенностями реализации и для поддержки старого кода.

Когда команда `initdb` выполняется, она показывает, какие параметры локали выбраны. Если вы задавали комбинацию параметров или у вас есть особые требования, рекомендуется проверить, соответствует ли результат ожиданиям.

Подробнее о параметрах локали можно узнать в Разделе 22.1.

Для изменения кодировки по умолчанию используется параметр `--encoding`. Узнать об этом больше можно в Разделе 22.3.

Параметры

`-A` *`authmethod`*
`--auth=`*`authmethod`*

Параметр определяет метод аутентификации по умолчанию для локальных пользователей, используемый в файле `pg_hba.conf` (строки `host` и `local`). Допустимые значения описаны в Разделе 19.1.

Программа `initdb` предварительно внесёт указанный метод аутентификации в `pg_hba.conf` в записи как обычных соединений, так и соединений репликации.

Не используйте `trust`, если не можете доверять всем локальным пользователям в вашей системе. Режим `trust` используется по умолчанию для облегчения процесса установки.

`--auth-host=`*`authmethod`*

Параметр указывает метод аутентификации для локальных пользователей, подключающихся по TCP/IP, используемый в `pg_hba.conf` (строки `host`).

`--auth-local=`*`authmethod`*

Параметр выбирает метод аутентификации локальных пользователей, подключающихся через Unix-сокет, используемый в `pg_hba.conf` (строки `local`).

`-D` *`каталог`*
`--pgdata=`*`каталог`*

Параметр указывает каталог хранения данных кластера. Это единственный обязательный параметр для команды `initdb`. При этом его можно указать в переменной окружения `PGDATA`, что будет удобным при дальнейшем использовании (`postgres` обращается к этой же переменной).

`-E` *`кодировка`*
`--encoding=`*`кодировка`*

Устанавливает кодировку по умолчанию для баз-шаблонов. Эта кодировка будет по умолчанию выбираться и для баз данных, создаваемых позже, если вы не переопределите её. Наборы символов, поддерживаемые сервером Postgres Pro, описаны в Подразделе 22.3.1.

По умолчанию кодировка для баз-шаблонов выбирается, исходя из указанной локали. Если задан параметр `--no-locale` (или, что равнозначно, задана локаль `C` или `POSIX`), по умолчанию используется `UTF8` для провайдера ICU и `SQL_ASCII` для провайдера `libc`.

`-g`
`--allow-group-access`

Позволяет пользователям, входящим в группу владельца кластера, читать все файлы кластера, создаваемые программой `initdb`. В Windows этот ключ не работает, так как там не поддерживаются разрешения для группы в стиле POSIX.

`--icu-locale=`*`локаль`*

Указывает локаль ICU, если используется провайдер локалей ICU. Поддержка локали описана в Разделе 22.1. По умолчанию используется значение `--locale`, если оно указано, в противном случае — значение lc_collate.

`--icu-rules=`*`правила`*

Указывает дополнительные правила для настройки поведения сортировки по умолчанию. Параметр поддерживается только для ICU.

`-k`
`--data-checksums`

Применять контрольные суммы на страницах данных для выявления сбоев при вводе-выводе, которые иначе останутся незамеченными. Расчёт контрольных сумм может повлечь заметное снижение производительности. Когда контрольные суммы включены, они рассчитываются для всех объектов и во всех базах данных. Все ошибки контрольных сумм будут видны в представлении `pg_stat_database`. За подробностями обратитесь к Разделу 27.2.

По умолчанию кластеры Postgres Pro инициализируются с включёнными контрольными суммами. Чтобы изменить это поведение, добавьте параметр `--no-data-checksums`.

`--no-data-checksums`

Отключить контрольные суммы в страницах данных.

По умолчанию кластеры Postgres Pro инициализируются с включёнными контрольными суммами.

`--locale=`*`локаль`*

Устанавливает локаль кластера по умолчанию. Если флаг не указан, локаль устанавливается согласно окружению, в котором исполняется команда `initdb`. Поддерживаемые локали описаны в Разделе 22.1.

Если для параметра `--locale-provider` установлено значение `builtin`, то также должен быть задан `--locale` или `--builtin-locale` со значением `C` или `C.UTF-8`.

`--lc-collate=`*`локаль`*
`--lc-ctype=`*`локаль`*
`--lc-messages=`*`локаль`*
`--lc-monetary=`*`локаль`*
`--lc-numeric=`*`локаль`*
`--lc-time=`*`локаль`*

Аналогично `--locale` устанавливает локаль, но только для заданной категории. Параметр `--lc-collate` переопределяет параметр `--locale` независимо от того, указан ли провайдер правил сортировки.

`--no-locale`

Аналогично флагу `--locale=C`.

`--builtin-locale=`*`локаль`*

Указывает имя локали при использовании провайдера `builtin`. Поддержка локалей описана в Разделе 22.1.

`--locale-provider={builtin|libc|icu}`

Этот параметр выбирает провайдера локалей для баз данных, создаваемых в новом кластере. Его можно переопределить для создаваемых впоследствии баз в команде `CREATE DATABASE`. См. Подраздел 22.1.4.

`--pwfile=`*`имя_файла`*

Принуждает `initdb` читать пароль начального суперпользователя из файла, первая строка которого используется в качестве пароля.

`--no-tune`

По умолчанию использование утилиты pgpro_tune в `initdb` включено. Чтобы её отключить, используйте этот параметр.

`--tune=`*`options`*

Устанавливает параметры для pgpro_tune. За дополнительной информацией обратитесь к их описанию.

`-T` *`конфигурация`*
`--text-search-config=`*`конфигурация`*

Устанавливает конфигурацию текстового поиска по умолчанию. За дополнительными сведениями обратитесь к default_text_search_config.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Задаёт имя пользователя для *начального суперпользователя*. По умолчанию это имя пользователя ОС, запускающего `initdb`.

`-W`
`--pwprompt`

Указывает `initdb` запросить пароль, который будет назначен начальному суперпользователю. Это не важно, если не планируется использовать аутентификацию по паролю. В ином случае этот режим аутентификации оказывается неприменимым, пока пароль не задан.

`-X` *`каталог`*
`--waldir=`*`каталог`*

Этот параметр указывает каталог для хранения журнала предзаписи.

`--wal-segsize=`*`размер`*

Задаёт *размер сегмента WAL*, в мегабайтах. Такой размер будет иметь каждый отдельный файл в журнале WAL. По умолчанию размер равен 16 мегабайтам. Значение должно задаваться степенью 2 от 1 до 1024 (в мегабайтах). Этот параметр можно установить только во время инициализации и нельзя изменить позже.

Этот размер бывает полезно поменять при тонкой настройке трансляции или архивации WAL. Кроме того, в базах данных с WAL большого объёма огромное количество файлов WAL в каталоге может стать проблемой с точки зрения производительности и администрирования. Увеличение размера файлов WAL приводит к уменьшению числа этих файлов.

Другие реже используемые параметры описаны здесь:

`-c` *`имя=значение`*
`--set` *`имя=значение`*

Принудительно устанавливает заданное *`значение`* для серверного параметра с указанным *`именем`* во время `initdb`, а также переопределяет этот параметр в файле `postgresql.conf`, чтобы использовать его при последующих запусках сервера. Этот параметр `initdb` можно задавать несколько раз, чтобы переопределить несколько серверных параметров. В основном он используется в случаях, когда сервер не может запуститься со значениями параметров по умолчанию.

`-d`
`--debug`

Выводит отладочные сообщения загрузчика и ряд других сообщений, не очень интересных широкой публике. Загрузчик — это приложение `initdb`, используемое для создания каталога таблиц. С этим параметром выдаётся очень много крайне скучных сообщений.

`--discard-caches`

Запустить загрузчик с параметром `debug_discard_caches=1`. Это очень сильно замедлит выполнение и полезно только для низкоуровневой отладки.

`-L` *`каталог`*

Указывает `initdb`, где необходимо искать входные файлы для развёртывания кластера. Обычно это не требуется. Приложение само запросит эти данные, если будет необходимо.

`-n`
`--no-clean`

По умолчанию, при выявлении ошибки на этапе развёртывания кластера, `initdb` удаляет все файлы, которые к тому моменту были созданы. Параметр предотвращает очистку файлов для целей отладки.

`-N`
`--no-sync`

По умолчанию `initdb` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `initdb` завершается быстрее, без ожидания, но в случае неожиданного сбоя операционной системы каталог данных может оказаться испорченным. Этот параметр может быть полезен при тестировании; в производственной среде применять его не следует.

`--no-instructions`

По умолчанию `initdb` после всех сообщений выводит инструкции по запуску кластера. С этим ключом данные инструкции выводиться не будут. Этот ключ предназначен в первую очередь для применения на платформах, где `initdb` используется в специальных высокоуровневых процедурах, в рамках которых данные инструкции оказываются некорректными.

`-s`
`--show`

Вывести внутренние параметры и завершить команду. Может использоваться для отладки во время установки initdb.

`--sync-method=`*`метод`*

Со значением `fsync` (это значение по умолчанию), `initdb` будет рекурсивно открывать и синхронизировать все файлы в каталоге данных. Поиск файлов будет осуществляться по символическим ссылкам для каталога WAL и каждого настроенного табличного пространства.

В Linux возможен вариант `syncfs`, когда от ОС требуется синхронизировать целиком каждую из файловых систем, содержащих каталог данных, файлы WAL и табличные пространства. Более подробно особенности использования `syncfs` описаны в recovery_init_sync_method.

В режиме `--no-sync` этот параметр не оказывает никакого влияния.

`-S`
`--sync-only`

Безопасно записывает все файлы базы на диск и останавливается. Другие операции initdb при этом не выполняются. В целом, этот параметр полезен для обеспечения надёжного восстановления после изменения значения fsync с `off` на `on`.

Другие параметры для тестирования 64-битных идентификаторов транзакций:

`-m` *`START_MX_ID`*
`--multixact-id=`*`START_MX_ID`*

Задаёт для нового экземпляра БД начальный идентификатор мультитранзакций в десятичном виде для тестирования обновлений БД, значение по умолчанию — `0`.

`-o` *`START_MX_OFFSET`*
`--multixact-offset=`*`START_MX_OFFSET`*

Задаёт для нового экземпляра БД начальное смещение мультитранзакций в десятичном виде для тестирования обновлений БД, значение по умолчанию — `0`.

`-x` *`START_XID`*
`--xid=`*`START_XID`*

Задаёт для нового экземпляра БД начальный идентификатор транзакций в десятичном виде для тестирования обновлений БД, значение по умолчанию — `0`.

Прочие параметры:

`-V`
`--version`

Выводит версию initdb и останавливается.

`-?`
`--help`

Показывает помощь по аргументам команды initdb и останавливается.

Переменные окружения

`PGDATA`

Указывает каталог хранения данных кластера, можно изменить параметром `-D`.

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

`PGPRO_TUNE`

Указывает, нужно ли использовать утилиту pgpro_tune без изменения параметров в командной строке. Если установлено значение `disable`, действие этой переменной окружения аналогично установлению параметра `--no-tune`, то есть отключает pgpro_tune. Если задано другое значение и параметр `--tune` или `--no-tune` в командной строке явным образом не указан, поведение будет таким же, как если бы для параметра `--tune` было задано значение `PGPRO_TUNE`.

`TZ`

Указывает часовой пояс кластера по умолчанию. Значение — это полное имя часового пояса (см. Подраздел 8.5.3).

Примечания

`initdb` можно выполнить командой `pg_ctl initdb`.

См. также

pg_ctl, postgres, Раздел 19.1

pg_archivecleanup

pg_archivecleanup — вычистить файлы архивов WAL Postgres Pro

Синтаксис

`pg_archivecleanup` [*параметр*...] *расположение_архива* *старейший_сохраняемый_файл*

Описание

Утилита pg_archivecleanup предназначена для использования в качестве `archive_cleanup_command` для удаления старых файлов WAL на резервном сервере (см. Раздел 25.2). pg_archivecleanup можно использовать и как отдельную программу для выполнения тех же действий.

Чтобы использовать pg_archivecleanup на резервном сервере, добавьте эту строку в файл конфигурации `postgresql.conf`:

```
archive_cleanup_command = 'pg_archivecleanup расположение_архива %r'
```

Здесь *расположение_архива* определяет каталог, из которого должны удаляться файлы сегментов WAL.

Вызываемая в качестве archive_cleanup_command, эта программа просматривает *расположение_архива* и удаляет все файлы WAL, логически предшествующие значению аргумента `%r`. Целью этой операции является сокращение числа сохраняемых файлов без потери возможности восстановления при перезапуске. Такой вариант использования уместен, когда *расположение_архива* указывает на область рабочих файлов конкретного резервного сервера, но *не* когда *расположение_архива* — каталог с архивом WAL для долговременного хранения, или когда несколько резервных серверов восстанавливают записи WAL из одного расположения.

При отдельном использовании этой программы из каталога *расположение_архива* будут удалены все файлы WAL, логически предшествующие файлу *старейший_сохраняемый_файл*. В этом режиме, если вы укажете имя файла с расширением `.partial` или `.backup`, *старейший_сохраняемый_файл* будет определяться по имени без расширения. Благодаря такой интерпретации расширения `.backup` будут корректно удалены все файлы WAL, заархивированные до определённой базовой копии. Например, следующая команда удалит все файлы старее файла WAL с именем `000000010000003700000010`:

```
pg_archivecleanup -d archive 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file "archive/00000001000000370000000F"
pg_archivecleanup:  removing file "archive/00000001000000370000000E"
```

pg_archivecleanup рассчитывает на то, что *расположение_архива* доступно для чтения и записи пользователю, владеющему серверным процессом.

Параметры

pg_archivecleanup принимает следующие аргументы командной строки:

`-b`
`--clean-backup-history`

Удалить также файлы истории резервного копирования. За более подробной информацией о файлах истории резервного копирования обратитесь к разделу Подраздел 24.3.2.

`-d`
`--debug`

Выводить подробные отладочные сообщения в `stderr`.

`-n`
`--dry-run`

Вывести имена файлов, которые должны быть удалены, в `stdout` (не выполняя удаление).

`-V`
`--version`

Вывести версию pg_archivecleanup и завершиться.

`-x` *`расширение`*
`--strip-extension=`*`расширение`*

Установить расширение, которое будет убрано из имён файлов до принятия решения об удалении определённых файлов. Это обычно полезно для очистки файлов архивов, которые сжимаются для хранения и получают расширение, задаваемое программой сжатия. Например: `-x .gz`.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_archivecleanup и завершиться.

Переменные окружения

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Программа pg_archivecleanup рассчитана на работу с PostgreSQL 8.0 и новее как отдельная утилита, а также с PostgreSQL 9.0 и новее как команда очистки архива.

Программа pg_archivecleanup написана на C; её исходный код легко поддаётся модификации (он содержит секции, предназначенные для изменения при надобности)

Примеры

В системах Linux или Unix можно использовать команду:

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r 2>>cleanup.log'
```

Предполагается, что каталог архива физически располагается на резервном сервере, так что команда `archive_command` обращается к нему по NFS, но для резервного сервера эти файлы локальные. Эта команда будет:

- выводить отладочную информацию в `cleanup.log`
- удалять ставшие ненужными файлы из каталога архива

pg_checksums

pg_checksums — включить, отключить или проверить контрольные суммы данных в кластере Postgres Pro

Синтаксис

`pg_checksums` [*параметр*...] [[`-D` | `--pgdata`]*каталог_данных*]

Описание

Утилита pg_checksums позволяет проверить, включить или отключить контрольные суммы данных в кластере Postgres Pro. Перед запуском pg_checksums сервер должен быть остановлен в штатном режиме. При проверке контрольных сумм она возвращает нулевой код состояния, если ошибок не найдено, либо ненулевой код, если обнаружится хотя бы одна ошибка. При включении или отключении контрольных сумм ненулевой код завершения показывает, что выполнить операцию не удалось.

В процессе проверки контрольных сумм проверяется каждый файл в кластере. При включении контрольных сумм для каждого блока файлов отношений рассчитывается контрольная сумма и он перезаписывается на месте, а при отключении изменяется только файл `pg_control`.

Параметры

Принимаются следующие параметры командной строки:

`-D` *каталог*
`--pgdata=`*каталог*

Указывает каталог, в котором располагается кластер баз данных.

`-c`
`--check`

Запускает проверку контрольных сумм. Это режим по умолчанию, который выбирается, когда не указан никакой другой.

`-d`
`--disable`

Отключает контрольные суммы.

`-e`
`--enable`

Включает контрольные суммы.

`-f` *файловый_узел*
`--filenode=`*файловый_узел*

Проверять контрольные суммы только в отношении, которому соответствует указанный *файловый_узел*.

`-N`
`--no-sync`

По умолчанию `pg_checksums` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_checksums` завершается быстрее, без ожидания, но в случае неожиданного сбоя операционной системы каталог с изменёнными файлами может повредиться. Этот параметр может быть полезен при тестировании; в производственной среде применять его не следует. В режиме `--check` он не оказывает никакого влияния.

`-P`
`--progress`

Включает вывод сообщений о прогрессе. Эти сообщения будут выводиться при проверке или включении контрольных сумм.

`--sync-method=`*`метод`*

Со значением `fsync` (это значение по умолчанию), `pg_checksums` будет рекурсивно открывать и синхронизировать все файлы в каталоге данных. Поиск файлов будет осуществляться по символическим ссылкам для каталога WAL и каждого настроенного табличного пространства.

В Linux возможен вариант `syncfs`, когда от ОС требуется синхронизировать целиком каждую из файловых систем, содержащих каталог данных, файлы WAL и табличные пространства. Более подробно особенности использования `syncfs` описаны в recovery_init_sync_method.

В режиме `--no-sync` этот параметр не оказывает никакого влияния.

`-v`
`--verbose`

Выводить подробные сообщения, в частности список всех проверенных файлов.

`-V`
`--version`

Выводит версию pg_checksums и завершает работу.

`-?`
`--help`

Показывает справку по аргументам командной строки pg_checksums и завершает работу.

Переменные окружения

`PGDATA`

Указывает каталог, в котором располагается кластер баз данных; может переопределяться параметром `-D`.

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Включение контрольных сумм в большом кластере может занять продолжительное время. Пока эта операция не закончится, нельзя запускать сервер или другие программы, которые могут произвести запись в каталог данных, иначе возможна потеря данных.

В конфигурациях, где организована репликация, и она осуществляется путём непосредственного копирования блоков отношений на уровне файлов (например, с помощью pg_rewind), включение или отключение контрольных сумм может привести к повреждению страниц (а именно, расхождению контрольных сумм), если эта операция не будет выполнена согласованно на всех узлах. Поэтому в подобных конфигурациях рекомендуется остановить все кластеры, с тем чтобы одновременно переключить их в другой режим. Ещё один безопасный вариант — ликвидировать все ведомые серверы, произвести нужную операцию на ведущем, а затем создать ведомые серверы заново.

Если pg_checksums прерывается или работающий процесс уничтожается при включении или отключении контрольных сумм, конфигурация контрольных сумм в кластере остаётся неизменной, и pg_checksums можно перезапустить ещё раз для повторения невыполненной операции.

pg_controldata

pg_controldata — вывести управляющую информацию кластера баз данных Postgres Pro

Синтаксис

`pg_controldata` [*параметр*] [[`-D` | `--pgdata`]*каталог_данных*]

Описание

`pg_controldata` показывает свойства, установленные командой `initdb`, например, версию каталога. Она также выводит сведения о журнале предзаписи, включая данные контрольной точки. Эта информация относится ко всему кластеру, а не к отдельной базе данных.

Утилита запускается от лица пользователя, создавшего кластер, так как требует права на чтение в каталоге хранения данных. Можно указать путь к каталогу из командной строки, либо использовать значение переменной окружения `PGDATA`. Также поддерживаются флаги `-V` и `--version`, которые выводят версию pg_controldata и прерывают выполнение. Флаги `-?` и `--help` отображают помощь по поддерживаемым командой аргументам.

Примечание

Вместо этой утилиты рекомендуется использовать pgpro_controldata, способную читать управляющую информацию баз данных PostgreSQL и Postgres Pro разных версий, в том числе более новых.

Переменные окружения

`PGDATA`

Каталог размещения данных кластера по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

См. также

pgpro_controldata

pg_createsubscriber

pg_createsubscriber — преобразовать физическую реплику в новую логическую реплику

Синтаксис

`pg_createsubscriber` [*параметр*...] { `-d` | `--database` }*имя_бд* { `-D` | `--pgdata` }*каталог_данных* { `-P` | `--publisher-server` }*строка_подключения*

Описание

pg_createsubscriber создаёт новую логическую реплику из резервного сервера физической репликации. Все таблицы указанной базы данных включаются в схему логической репликации. Для каждой базы данных создаётся пара объектов публикации и подписки. Утилиту нужно запускать на целевом сервере.

После успешного запуска состояние целевого сервера не отличается от новой схемы логической репликации. Единственное отличие логической репликации от механизма pg_createsubscriber в том, как выполняется синхронизация данных. Утилита не копирует исходные данные таблицы, а лишь выполняет синхронизацию, что позволяет привести каждую таблицу в синхронизированное состояние.

Утилита pg_createsubscriber предназначена для крупных баз данных, поскольку при логической репликации больше всего времени уходит именно на начальное копирование данных. Кроме того, в результате длительной синхронизации обычно приходится применять много изменений (которые возникли при начальном копировании), из-за чего логическая реплика ещё дольше остаётся недоступной. Для небольших баз данных рекомендуется настраивать логическую репликацию с начальной синхронизацией данных. За подробностями обратитесь к параметру `CREATE SUBSCRIPTION` `copy_data`.

Параметры

pg_createsubscriber принимает следующие аргументы командной строки:

`-d` *`имя_бд`*
`--database=`*`имя_бд`*

Имя базы данных, в которой необходимо создать подписку. Укажите несколько параметров `-d`, чтобы выбрать несколько баз данных.

`-D` *`каталог`*
`--pgdata=`*`каталог`*

Целевой каталог, содержащий каталог кластера физической реплики.

`-n`
`--dry-run`

Делать всё, кроме внесения изменений в целевой каталог.

`-p` *`порт`*
`--subscriber-port=`*`порт`*

Номер порта, через который целевой сервер принимает подключения. По умолчанию целевой сервер запускается с портом 50432, чтобы не допустить нежелательных клиентских подключений.

`-P` *`строка_подключения`*
`--publisher-server=`*`строка_подключения`*

Строка подключения к публикующему серверу. Подробности описаны в Подразделе 32.1.1.

`-s` *`каталог`*
`--socketdir=`*`каталог`*

Каталог, в котором процесс postmaster будет создавать сокеты на целевом сервере. По умолчанию — текущий каталог.

`-t` *`секунды`*
`--recovery-timeout=`*`секунды`*

Максимальное время ожидания завершения восстановления (в секундах). Значение 0 (по умолчанию) отключает этот тайм-аут.

`-U` *`имя_пользователя`*
`--subscriber-username=`*`имя_пользователя`*

Имя пользователя для подключения к целевому серверу. По умолчанию — имя текущего пользователя операционной системы.

`-v`
`--verbose`

Включает режим подробных сообщений. При включённом параметре pg_createsubscriber будет выводить сообщения о процессе выполнения и подробную информацию о каждом шаге в стандартный поток ошибок. Если указать этот параметр повторно, в стандартный поток ошибок будут выдаваться дополнительные отладочные сообщения.

`--config-file=`*`имя_файла`*

Использовать указанный основной файл конфигурации сервера для целевого каталога данных. Утилита pg_createsubscriber использует команду `pg_ctl` для запуска и остановки целевого сервера. Этот параметр позволяет указывать фактический файл конфигурации `postgresql.conf`, если он хранится за пределами каталога данных.

`--publication=`*`имя`*

Имя публикации для настройки логической репликации. Используйте несколько параметров `--publication`, чтобы указать несколько публикаций. Число публикаций должно совпадать с числом указанных баз данных, в противном случае выводится сообщение об ошибке. Публикации необходимо указывать в том же порядке, что и базы данных в соответствующем параметре. Если не указать этот параметр, публикации присваивается сгенерированное имя.

`--replication-slot=`*`имя`*

Имя слота репликации для настройки логической репликации. Используйте несколько параметров `--replication-slot`, чтобы указать несколько слотов репликации. Число слотов должно совпадать с числом указанных баз данных, в противном случае выводится ошибка. Слоты репликации необходимо указывать в том же порядке, что и базы данных в соответствующем параметре. Если не указать этот параметр, слоту репликации присваивается имя подписки.

`--subscription=`*`имя`*

Имя подписки для настройки логической репликации. Используйте несколько параметров `--subscription`, чтобы указать несколько подписок. Число подписок должно совпадать с числом указанных баз данных, в противном случае выводится ошибка. Подписки необходимо указывать в том же порядке, что и базы данных в соответствующем параметре. Если не указать этот параметр, подписке присваивается сгенерированное имя.

`-V`
`--version`

Вывести версию pg_createsubscriber и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_createsubscriber и завершиться.

Примечания

Предварительные требования

Существуют определённые требования, которые должны выполнятся, чтобы утилита pg_createsubscriber могла преобразовать целевой сервер в логическую реплику. При их невыполнении выводится сообщение об ошибке. Для корректного использования pg_createsubscriber основные версии исходного и целевого сервера должны совпадать. У заданного целевого каталога данных должен быть такой же системный идентификатор, как у исходного. Пользователь баз данных, указанный для целевого каталога данных, должен иметь права на создание подписок и использование функции `pg_replication_origin_advance()`.

Целевой сервер должен использоваться как резервный сервер физической репликации. На целевом сервере значение параметров конфигурации max_replication_slots и max_logical_replication_workers должно быть больше или равно числу указанных баз данных, а значение параметра max_worker_processes должно быть больше этого числа. Целевой сервер должен принимать локальные подключения.

Исходный сервер должен принимать подключения от целевого сервера. Исходный сервер не должен находиться в состоянии восстановления. На исходном сервере параметр конфигурации wal_level должен иметь значение `logical`, значение параметра max_replication_slots должно быть больше или равно сумме указанных баз данных и существующих слотов репликации, а значение параметра max_wal_senders должно быть больше или равно сумме указанных баз данных и существующих процессов передачи WAL.

Предупреждения

При нарушении работы pg_createsubscriber после повышения целевого сервера, вероятнее всего, каталог данных нельзя будет восстановить. Если это произошло, рекомендуется создать новый резервный сервер.

При преобразовании реплик pg_createsubscriber обычно запускает целевой сервер с другими параметрами подключения, поэтому в этот момент подключиться к нему не получится.

DDL-команды не участвуют в логической репликации, поэтому при работе с pg_createsubscriber необходимо избегать выполнения команд, изменяющих схему баз данных. Если целевой сервер уже преобразован в логическую реплику, DDL-команды, вероятно, не будут реплицированы, что может вызвать ошибку.

При нарушении работы pg_createsubscriber во время обработки данных удаляются объекты (публикации, слоты репликации), созданные на исходном сервере. Если целевому серверу не удаётся подключиться к исходному серверу, удаление может завершиться ошибкой. В этом случае предупреждение покажет объекты, которые не были удалены. Запущенный целевой сервер будет остановлен.

Если для репликации используется параметр primary_slot_name, этот слот будет удалён с исходного сервера после настройки логической репликации.

Если целевой сервер является синхронной репликой, при работе pg_createsubscriber транзакции на ведущем сервере могут ожидать завершения репликации.

Утилита pg_createsubscriber настраивает логическую репликацию с отключённой двухфазной фиксацией. Это означает, что любые подготовленные транзакции будут реплицироваться во время выполнения оператора `COMMIT PREPARED` без предварительной подготовки. После завершения настройки можно вручную удалять и создавать подписки заново с включённым параметром `two_phase`.

pg_createsubscriber изменяет системный идентификатор при помощи утилиты pg_resetwal, которая позволяет избегать ситуаций, когда целевой сервер может использовать файлы WAL исходного сервера. Если у целевого сервера есть резервный сервер, репликация прервётся и нужно будет создать новый резервный сервер.

Как это работает

Основная идея состоит в том, чтобы иметь начальную точку репликации с исходного сервера и настроить начало логической репликации с этой точки:

1. Запустить целевой сервер с указанными параметрами командной строки. Если он уже запущен, pg_createsubscriber завершится с ошибкой.
2. Проверить, можно ли преобразовать целевой сервер. Также выполняется несколько проверок исходного сервера. Если какое-либо из предварительных требований не выполнено, pg_createsubscriber завершится ошибкой.
3. Создать публикацию и слот репликации для каждой указанной базы данных на исходном сервере. Каждая публикация создаётся с использованием параметра `FOR ALL TABLES`. Если параметр `--publication` не указан, имя публикации задаётся по следующему шаблону: «`pg_createsubscriber_%u_%x`» (параметры: *`oid`* базы данных, случайное целое число *`int`*). Если параметр `--replication-slot` не указан, имя слота репликации задаётся по следующему шаблону: «`pg_createsubscriber_%u_%x`» (параметры: *`oid`* базы данных, случайное целое число *`int`*). Эти слоты репликации будут использоваться подписками на следующем шаге. Последний LSN для слота репликации используется в качестве точки остановки в параметре recovery_target_lsn, а подписками используется в качестве начальной точки репликации. Таким образом гарантируется, что ни одна транзакция не будет потеряна.
4. Записать параметры восстановления в целевой каталог данных и перезапустить целевой сервер. Указывает LSN (recovery_target_lsn) позиции в журнале предзаписи, до которой должно выполняться восстановление. Также указывает `promote` в качестве действия, которое должен выполнить сервер после достижения точки восстановления. Добавляются дополнительные параметры восстановления, чтобы избежать непредвиденного поведения во время процесса восстановления, например завершения восстановления при достижении согласованного состояния (WAL должен применяться до точки начала репликации) или появления множественных точек восстановления, которые могут привести к сбою. Этот шаг завершается после выхода сервера из режима резерва и начала приёма транзакций чтения-записи. При заданном параметре `--recovery-timeout` утилита pg_createsubscriber завершает работу, если восстановление не заканчивается до истечения указанного количества секунд.
5. Создать подписку для каждой указанной базы данных на целевом сервере. Если параметр `--subscription` не указан, имя подписки задаётся по следующему шаблону: «`pg_createsubscriber_%u_%x`» (параметры: *`oid`* базы данных, случайное целое число *`int`*). Подписка не копирует существующие данные с исходного сервера и не создаёт слот репликации. Она использует слот репликации, который был создан на предыдущем этапе. Подписка создана, но ещё не активирована, потому что, прежде чем начать репликацию, её положение нужно установить на начальную точку.
6. Удалить на целевом сервере публикации, изменения которых были реплицированы, так как они были созданы до точки начала репликации. Подписчик не сможет их использовать.
7. Установить положение репликации в начальную точку репликации для всех подписок. Когда целевой сервер начинает восстановление, он нагоняет начальную точку репликации. Именно этот LSN должен использоваться в качестве исходного положения репликации для всех подписок. Имя источника репликации поступает из-за создания подписки. Имя источника репликации и начальная точка репликации используются функцией `pg_replication_origin_advance()`, чтобы установить исходное положение репликации.
8. Активировать подписку для каждой указанной базы данных на целевом сервере. Подписка начинает применять транзакции с начальной точки репликации.
9. Если на резервном сервере был установлен параметр primary_slot_name, теперь его можно удалить за ненадобностью.
10. Если на резервном сервере есть слоты репликации для отработки отказа, их можно удалить из-за невозможности дальнейшей синхронизации.

11. Обновить системный идентификатор на целевом сервере. Для его изменения запускается утилита pg_resetwal. Целевой сервер останавливается — это обязательное условие для выполнения команды `pg_resetwal`.

Примеры

Используйте следующую команду, чтобы создать логическую реплику для баз данных `hr` и `finance` из физической реплики, подключившись к публикующему серверу `foo`:

```
$ pg_createsubscriber -D /usr/local/pgsql/data -P "host=foo" -d hr -d finance
```

См. также

pg_basebackup

pg_ctl

pg_ctl — инициализировать, запустить, остановить или управлять сервером Postgres Pro

Синтаксис

`pg_ctl` `init[db]` [`-D` *каталог_данных*] [`-s`] [`-o` *параметры-initdb*]

`pg_ctl` `start` [`-D` *каталог_данных*] [`-l` *имя_файла*] [`-W`] [`-t` *секунды*] [`-s`] [`-o` *параметры*] [`-p` *путь*] [`-c`]

`pg_ctl` `stop` [`-D` *каталог_данных*] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t` *секунды*] [`-s`]

`pg_ctl` `restart` [`-D` *каталог_данных*] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t` *секунды*] [`-s`] [`-o` *параметры*] [`-c`]

`pg_ctl` `reload` [`-D` *каталог_данных*] [`-s`]

`pg_ctl` `status` [`-D` *каталог_данных*]

`pg_ctl` `promote` [`-D` *каталог_данных*] [`-W`] [`-t` *секунды*] [`-s`]

`pg_ctl` `logrotate` [`-D` *каталог_данных*] [`-s`]

`pg_ctl` `kill` *имя_сигнала* *ид_процесса*

В системах Microsoft Windows также:

`pg_ctl` `register` [`-D` *каталог_данных*] [`-N` *имя_службы*] [`-U` *имя_пользователя*] [`-P` *пароль*] [`-S` `a[uto]` | `d[emand]`] [`-e` *источник*] [`-W`] [`-t` *секунды*] [`-s`] [`-o` *параметры*]

`pg_ctl` `unregister` [`-N` *имя_службы*]

Описание

pg_ctl — это утилита для начальной инициализации, запуска, остановки, повторного запуска и управления кластером баз данных Postgres Pro (postgres). Сервер можно стартовать в ручном режиме, но pg_ctl реализует задачи направления вывода в журнал и отсоединения от терминала и группы процессов, а также предоставляет удобный интерфейс остановки кластера.

Команда `init` (`initdb`) создаёт кластер баз данных Postgres Pro, то есть коллекцию баз данных, которой будет управлять один экземпляр сервера. Эта команда вызывает программу `initdb`. За подробностями обратитесь к initdb.

Команда `start` запускает сервер. Процесс запускается в фоне, а стандартный ввод связывается с `/dev/null` (или `nul` в Windows). По умолчанию в Unix-подобных системах вывод и ошибки сервера пишутся в устройство стандартного вывода (не ошибок) pg_ctl. Вывод pg_ctl следует перенаправить в файл или процесс, например, приложение ротации журналов rotatelogs; иначе `postgres` будет писать вывод в управляющий терминал (в фоновом режиме) и останется в группе процессов оболочки. В Windows сообщения и ошибки сервера по умолчанию перенаправляются в терминал. Это поведение по умолчанию можно изменить и направить вывод сервера в файл, добавив ключ `-l`. Предпочтительными вариантами является использование `-l` или перенаправление вывода.

Команда `stop` останавливает сервер, работающий с указанным каталогом данных. Параметр `-m` позволяет выбрать один из трёх режимов остановки. Режим «Smart» запрещает новые подключения, а затем ожидает отключения всех существующих клиентов. Если сервер работает в режиме горячего резерва, восстановление и потоковая репликация будут прерваны, как только отключатся все клиенты. Режим «Fast» (выбираемый по умолчанию) не ожидает отключения клиентов. Все активные транзакции откатываются, а клиенты принудительно отключаются, после чего сервер останавливается. Режим «Immediate» незамедлительно прерывает все серверные процессы, не выполняя процедуру штатной остановки. Этот вариант влечёт необходимость выполнить восстановление после сбоя при следующем запуске сервера.

Команда `restart` по сути производит остановку и последующий запуск сервера. Это позволяет изменить параметры командной строки `postgres` либо применить изменения в файле конфигурации, не вступающие в силу без перезапуска сервера. Если в командной строке при запуске сервера указывались относительные пути, команда `restart` может не выполниться, если вызвать pg_ctl не в том каталоге, где производился предыдущий запуск.

Команда `reload` просто посылает процессу сервера `postgres` сигнал SIGHUP, получив который он перечитывает свои файлы конфигурации (`postgresql.conf`, `pg_hba.conf` и т. д.). Это позволяет применить изменения параметров в файле конфигурации, не требующие полного перезапуска сервера.

Команда `status` проверяет, работает ли сервер в указанном каталоге данных. Если да, она выдаёт PID сервера и параметры командной строки, с которыми он был запущен. Если сервер не работает, pg_ctl возвращает код завершения 3. Если в параметрах не указан доступный каталог данных, pg_ctl возвращает код завершения 4.

Команда `promote` указывает серверу, работающему в режиме резерва с указанным каталогом данных, выйти из этого режима и начать операции чтения/записи.

Команда `logrotate` прокручивает файл журнала сервера. Подробнее о том, как использовать это с внешними средствами прокрутки журнала, рассказывается в Разделе 23.3.

Команда `kill` передаёт сигнал заданному процессу. Прежде все это полезно в Microsoft Windows, где отсутствует встроенная команда kill. Для получения списка имён поддерживаемых сигналов воспользуйтесь ключом `--help`.

Команда `register` регистрирует сервер Postgres Pro в качестве системной службы в Microsoft Windows. Параметр `-S` позволяет выбрать тип запуска службы: «auto» (запускать службу автоматически при загрузке системы) или «demand» (запускать службу по требованию).

Режим `unregister` разрегистрирует системную службу в Microsoft Windows. Эта операция отменяет действие команды `register`.

Параметры

`-c`
`--core-files`

Способствует сбросу дампа памяти процесса при крахе сервера на платформах, где это возможно, поднимая мягкие ограничения, задаваемые для файлов дампа. Это полезно при отладке и диагностике проблем, так как позволяет получить трассировку стека отказавшего процесса сервера.

`-D` *каталог_данных*
`--pgdata=`*каталог_данных*

Указывает размещение конфигурационных файлов кластера. Если этот ключ опущен, используется значение переменной окружения `PGDATA`.

`-l` *имя_файла*
`--log=`*имя_файла*

Направляет вывод сообщений сервера в файл *имя_файла*. Файл создаётся, если он ещё не существует. При этом устанавливается umask 077, что предотвращает доступ других пользователей к этому файлу.

`-m` *`режим`*
`--mode=`*`режим`*

Задаёт режим остановки кластера. Значением *`режим`* может быть `smart`, `fast` или `immediate`, либо первая буква этих вариантов. Если этот ключ опущен, по умолчанию выбирается режим `fast`.

`-o` *`параметры`*
`--options=`*`параметры`*

Указывает параметры, которые будут передаваться непосредственно программе `postgres`. Ключ `-o` можно указывать несколько раз, при этом ей будут переданы параметры из всех ключей.

Задаваемые *`параметры`* обычно следует обрамлять одинарными или двойными кавычками, чтобы они передавались одной группой.

`-o` *`параметры-initdb`*
`--options=`*`параметры-initdb`*

Указывает параметры, которые будут передаваться непосредственно программе `initdb`. Ключ `-o` можно указывать несколько раз, при этом ей будут переданы параметры из всех ключей.

Задаваемые *`параметры-initdb`* обычно следует обрамлять одинарными или двойными кавычками, чтобы они передавались вместе одной группой.

`-p` *`путь`*

Указывает размещение исполняемого файла `postgres`. По умолчанию задействуется исполняемый файл `postgres` из того же каталога, из которого запускался `pg_ctl`, а если это невозможно, из жёстко заданного каталога инсталляции. Применять этот параметр может понадобиться, только если вы делаете что-то необычное или получаете сообщения, что найти исполняемый файл `postgres` не удаётся.

В режиме `init` этот параметр аналогичным образом задаёт размещение исполняемого файла `initdb`.

`-s`
`--silent`

Выводить лишь ошибки, без сообщений информационного характера.

`-t` *`секунды`*
`--timeout=`*`секунды`*

Задаёт максимальное время (в секундах) ожидания завершения операции (см. параметр `-w`). По умолчанию действует значение переменной среды `PGCTLTIMEOUT` или, если оно не задано, 60 секунд.

`-V`
`--version`

Выводит версию pg_ctl и прерывает выполнение.

`-w`
`--wait`

Ждать завершения операции. Этот режим поддерживается (и действует по умолчанию) для команд `start`, `stop`, `restart`, `promote` и `register`.

В процессе ожидания `pg_ctl` постоянно проверяет PID-файл сервера, приостанавливаясь на короткое время между проверками. Запуск считается завершённым, когда PID-файл указывает на то, что сервер готов принимать подключения. Остановка считается завершённой, когда сервер удаляет свой PID-файл. Программа `pg_ctl` возвращает код завершения в зависимости от успеха запуска или остановки.

Если операция не заканчивается за отведённое время (см. параметр `-t`), программа `pg_ctl` завершается с ненулевым кодом выхода. Но заметьте, что при этом выполнение операции может продолжиться и в конце концов увенчаться успехом.

`-W`
`--no-wait`

Не ждать завершения операции. Этот режим противоположен режиму `-w`.

Если ожидание отключено, запрошенное действие вызывается, но о его результате ничего не известно. В этом случае для проверки текущего состояния и результата операции потребуется обратиться к файлу журнала сервера или воспользоваться внешней системой мониторинга.

В предыдущих выпусках Postgres Pro этот режим действовал по умолчанию (кроме команды `stop`).

`-?`
`--help`

Вывести справку по команде pg_ctl и прервать выполнение.

Если некоторый параметр является допустимым, но не применим к выбранному режиму работы, pg_ctl игнорирует его.

Параметры, специфичные для Windows

`-e` *`источник`*

Имя источника событий, с которым pg_ctl будет записывать в системный журнал события при запуске в виде службы Windows. Имя по умолчанию — `Postgres Pro`. Заметьте, что это влияет только на сообщения, которые выдаёт сам pg_ctl; как только сервер запустится, он будет использовать источник событий, заданный в event_source. Если произойдёт ошибка при запуске сервера на ранней стадии, прежде чем будет считан этот параметр, он может также выдавать сообщения с источником по умолчанию `Postgres Pro`.

`-N` *`имя_службы`*

Имя регистрируемой системной службы. Оно станет и собственно именем службы, и отображаемым именем. По умолчанию — `Postgres Pro`.

`-P` *`пароль`*

Пароль для пользователя, запускающего службу.

`-S` *`тип-запуска`*

Тип запуска системной службы. В качестве значения *`тип-запуска`* можно задать `auto`, `demand` или первую букву этих слов. По умолчанию выбирается тип `auto`.

`-U` *`имя_пользователя`*

Имя пользователя, от имени которого будут запущена служба. Для доменных пользователей используйте формат `DOMAIN\username`.

Переменные окружения

`PGCTLTIMEOUT`

Значение по умолчанию для максимального времени ожидания запуска или остановки сервера (в секундах). По умолчанию это время составляет 60 секунд.

`PGDATA`

Размещение каталога хранения данных по умолчанию.

Для большинства режимов `pg_ctl` требуется знать расположение каталога данных; поэтому если не задана переменная `PGDATA`, параметр `-D` является обязательным.

Список дополнительных переменных, влияющих на работу сервера, можно найти в postgres.

Файлы

`postmaster.pid`

Проверяя этот файл в каталоге данных, pg_ctl определяет, работает ли сервер в настоящий момент.

`postmaster.opts`

Если файл существует в каталоге хранения данных, то pg_ctl (при `restart`) передаст его содержимое в качестве аргументов postgres, если не указаны иные значения в `-o`. Содержимое файла также отображается при вызове в режиме `status`.

Примеры

Запуск сервера

Запуск сервера и ожидание момента, когда он начнёт принимать подключения:

```
$ pg_ctl start
```

Чтобы запустить сервер с использованием порта 5433 и без `fsync`, выполните:

```
$ pg_ctl -o "-F -p 5433" start
```

Остановка сервера

Чтобы остановить сервер, выполните:

```
$ pg_ctl stop
```

Ключ `-m` позволяет управлять тем, *как* сервер будет остановлен:

```
$ pg_ctl stop -m smart
```

Повторный запуск сервера

Перезапуск сервера почти равнозначен остановке и запуску сервера за исключением того, что по умолчанию `pg_ctl` сохраняет параметры командной строки, которые были переданы ранее запущенному экземпляру. Таким образом, чтобы перезапустить сервер с теми же параметрами, с какими он был запущен, выполните:

```
$ pg_ctl restart
```

Но если добавляется ключ `-o`, он заменяет все предыдущие параметры. Эта команда осуществит перезапуск с использованием порта 5433 и без `fsync`:

```
$ pg_ctl -o "-F -p 5433" restart
```

Вывод состояния сервера

Ниже представлен примерный вывод pg_ctl:

```
$ pg_ctl status

pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

Во второй строке показывается команда, которая будет выполнена в режиме перезапуска.

См. также

initdb, postgres

pgpro_tune

pgpro_tune — утилита командной строки для автоматической настройки

Синтаксис

`pgpro_tune` [*параметр*...] [`-D` | `--pgdata`]*каталогимя_пресета*

Описание

pgpro_tune — это утилита командной строки для автоматической настройки. Оптимальные значения различных параметров конфигурации Postgres Pro зависят от оборудования. Утилита pgpro_tune собирает информацию о системе и преобразует её в набор параметров, записываемых в файл конфигурации.

Использование

Использование pgpro_tune включено в `initdb` по умолчанию. Для этой утилиты предусмотрено два параметра `initdb`: `--no-tune` отключает её использование, `--tune=OPTIONS` позволяет передавать дополнительные параметры для запуска pgpro_tune.

Сначала pgpro_tune запускает стандартный пресет. Дополнительные пресеты могут быть заданы с помощью параметра `-P` или `--preset=NAME`, или путём указания имени пресета в качестве последнего аргумента командной строки. Чтобы отключить стандартный пресет, используйте параметр `--no-default`.

В данный момент доступны следующие пресеты по умолчанию:

- `default.tune`: Основной на данный момент пресет, который обеспечивает различные конфигурации в зависимости от редакции и версии сервера.
- `regress.tune`: Этот пресет применяется для отладки. Он отменяет изменения, произведённые конфигурационными параметрами пресета `default.tune`. Таким образом обеспечивается успешное выполнение регрессионных тестов, результаты которых могут различаться из-за аппаратно-зависимых настроек.
- `1c.tune`: Этот пресет содержит параметры, специфичные для 1С, некоторые из которых требуют установки дополнительных модулей (таких как online_analyze и plantuner). Подробную информацию о настройках Postgres Pro для решений 1С можно найти в Приложении K.
- `extensions.tune`: Этот пресет содержит дополнительные параметры для расширений. Эти параметры требуют установки других модулей (таких как pg_stat_statements).

Дополнительные параметры можно указать с помощью `-O` или `--options=OPTIONS` для любого пресета, но для пресета по умолчанию следует использовать `--default-options=OPTIONS`.

Дополнительные переменные окружения для пресета можно установить в параметре `--set NAME=VALUE`. Эти значения будут одинаковыми как для пресета по умолчанию, так и для всех пользовательских.

По умолчанию пресеты размещаются в каталоге `share`, но можно указать их расположение вручную в параметре `--preset-dir=NAME`.

По умолчанию pgpro_tune работает с файлом конфигурации `postgresql.conf`, расположенным в каталоге данных. Каталог данных можно указать с помощью `-D` или `--pgdata=DATADIR`, либо задать с помощью переменной окружения `PGDATA`. При запуске pgpro_tune в `initdb` утилита будет использовать каталог данных `initdb`.

Файл конфигурации также можно указать с помощью `--config-file=FILENAME`.

Утилита pgpro_tune записывает свои изменения в отдельный блок в конце файла конфигурации с комментарием, что он был добавлен pgpro_tune. За подробностями обратитесь к примеру пресета.

Параметры

`--config-file=`*`имя_файла`*

Указывает имя основного файла конфигурации.

`-D` *`каталог_данных`*
`--pgdata=`*`каталог_данных`*

Указывает каталог, где будет храниться кластер баз данных.

`--default-options=`*`параметры`*

Задаёт параметры для стандартного пресета.

`--no-default`

Отключает стандартный пресет. Обратите внимание, что в этом случае обязательно должен быть указан другой пресет либо с помощью параметра `-P`, либо в качестве последнего параметра в виде имени этого пресета.

`-O` *`параметры-пресета`*
`--options=`*`параметры-пресета`*

Указывает параметры, которые необходимо передать непосредственно пресету, заданному ранее в параметре `-P`, или пресету, указанному в качестве последнего параметра командной строки, если `-P` не использовался.

`-P` *`имя`*
`--preset=`*`имя`*

Указывает имя пресета. Если указано несколько имён, пресеты выполняются по очереди с учётом вывода предыдущего пресета.

> **Примечание**
>
> Имя пресета может быть указано либо в этом параметре, либо в качестве последнего параметра. Использование обоих вариантов одновременно приведёт к ошибке.

`--preset-dir=`*`имя`*

Указывает каталог, содержащий пресеты. По умолчанию пресеты находятся в каталоге `share`.

`--set` *`имя`*`=`*`значение`*

Устанавливает переменную окружения для пресетов.

`--show`

Показывает все доступные пресеты из каталога, указанного в параметре `--preset-dir`, если он задан, или из каталога по умолчанию в противном случае.

`-V`
`--version`

Вывести версию pgpro_tune и завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки pgpro_tune и завершиться.

Переменные окружения

pgpro_tune устанавливает следующие переменные окружения перед запуском пресетов.

- `EDITION` — редакция Postgres Pro.
- `ENABLE_CRASH_INFO` — если установлена, Postgres Pro был собран с параметром `--enable-crash-info`.
- `ENABLE_NLS` — если установлена, Postgres Pro был собран с параметром `--enable-nls`.
- `ENABLE_PGPRO_TUNE` — если установлена, Postgres Pro был собран с параметром `--enable-pgpro-tune`.
- `MEMMB` — размер оперативной памяти в мегабайтах.
- `MVER` — основная версия Postgres Pro.
- `NCPU` — количество процессоров.
- `USE_BONJOUR` — если установлена, Postgres Pro был собран с параметром `--with-bonjour`.
- `USE_BSD_AUTH` — если установлена, Postgres Pro был собран с параметром `--with-bsd-auth`.
- `USE_ICU` — если установлена, Postgres Pro был собран с параметром `--with-icu`.
- `USE_LDAP` — если установлена, Postgres Pro был собран с параметром `--with-ldap`.
- `USE_LIBUNWIND` — если установлена, Postgres Pro был собран с параметром `--with-libunwind`.
- `USE_LIBXML` — если установлена, Postgres Pro был собран с параметром `--with-libxml`.
- `USE_LIBXSLT` — если установлена, Postgres Pro был собран с параметром `--with-libxslt`.
- `USE_LLVM` — если установлена, Postgres Pro был собран с параметром `--with-llvm`.
- `USE_LZ4` — если установлена, Postgres Pro был собран с параметром `--with-lz4`.
- `USE_OPENSSL` — если установлена, Postgres Pro был собран с параметром `--with-openssl`.
- `USE_PAM` — если установлена, Postgres Pro был собран с параметром `--with-pam`.
- `USE_SYSTEMD` — если установлена, Postgres Pro был собран с параметром `--with-systemd`.
- `USE_ZSTD` — если установлена, Postgres Pro был собран с параметром `--with-zstd`.

Создание пресетов

Пресет — это исполняемый файл, обычно скрипт оболочки, который преобразует информацию, передаваемую через переменные окружения, в набор параметров конфигурации.

Ожидаемый вывод пресета — набор специально отформатированных строк. Первый непробельный символ — индикатор интерпретации содержимого строки для pgpro_tune. Индикатор должен быть одним из следующих символов:

- `#` для комментирования, остальная часть выходной строки будет записана в файл конфигурации после «#».
- `=` для замены, остальная часть выходной строки должна содержать пару параметров `NAME` и `VALUE` (приемлемы оба варианта: `NAME=VALUE` и `NAME VALUE`). Строка `NAME = VALUE` будет записана в файл конфигурации. Текст после `VALUE` будет добавлен как комментарий.
- `=+` или `+=` для добавления, остальная часть выходной строки должна содержать пару параметров `NAME` и `VALUE`. `VALUE` будет добавлен к предыдущему действующему значению параметра `NAME` сзади (`=+`) или спереди (`+=`). Если действующего значения нет, `ADD` работает как замена. Текст после `VALUE` будет добавлен как комментарий.
- `-=` для отмены, остальная часть выходной строки должна содержать пару параметров `NAME` и `VALUE`. Если параметр `NAME` имеет действующее значение, содержащее `VALUE`, оно будет удалено, а остальная часть действующего значения останется прежней. Текст после `VALUE` будет добавлен в качестве комментария. Если действующего значения нет, ничего не меняется.

Пример пресета

Предположим, есть следующий файл конфигурации с именем `test.conf`:

```
work_mem = 4MB # Default value
shared_preload_libraries = 'plantuner'
search_path = '"$user",wrong_schema,public'
```

И следующий пресет с именем `test.tune`:

```
echo "# Adding new configuration parameters."
#Replace configuration parameter value by a new one
echo "work_mem = 8MB"
#Append to the start of existing value
echo "shared_preload_libraries += pg_stat_statements"
#Append to the end of existing value
echo "shared_preload_libraries =+ pg_prewarm"
#Withdraw from existing value
echo "search_path -= 'wrong_schema'"
```

Чтобы использовать этот пресет, выполните следующую команду:

```
pgpro_tune --config-file=/path/to/test.conf -P/path/to/test.tune --no-default
```

Эта команда приведёт к следующим изменениям файла конфигурации:

```
#------------------------------------------------------------------------------
# The following settings were added by pgpro_tune.
# pgpro_tune was run with the following options:
# --no-default --config-file=/path/to/test/conf -P/path/to/test/tune --no-default
# At YYYY-MM-DD HH:MM:SS
#------------------------------------------------------------------------------
# Adding new configuration parameters.
work_mem = 8MB
shared_preload_libraries = 'pg_stat_statements, plantuner'
shared_preload_libraries = 'pg_stat_statements, plantuner, pg_prewarm'
search_path = '"$user", public'
#------------------------------------------------------------------------------
# End of settings added by pgpro_tune at YYYY-MM-DD HH-MM-SS
#------------------------------------------------------------------------------
```

pg_resetwal

pg_resetwal — очистка журнала предзаписи и другой управляющей информации кластера Postgres Pro

Синтаксис

`pg_resetwal` [`-f` | `--force`] [`-n` | `--dry-run`] [*параметр*...] [`-D` | `--pgdata`]*каталог_данных*

Описание

`pg_resetwal` очищает журнал предзаписи (WAL) и может сбросить некоторую другую управляющую информацию, хранящуюся в файле `pg_control`. Данная функция может быть востребована при повреждении этих файлов. Использовать её нужно только как крайнюю меру, когда запуск сервера оказывается невозможен из-за этого повреждения.

Некоторые параметры, такие как `--wal-segsize` (см. ниже), могут также использоваться для изменения некоторых глобальных параметров кластера баз данных без необходимости повторного запуска `initdb`. Это можно сделать безопасно в работающем кластере баз данных, если не используются опасные режимы, упомянутые ниже.

Если `pg_resetwal` используется для каталога данных, где сервер был остановлен в штатном режиме и управляющий файл в порядке, то это не повлияет на содержимое системы баз данных, за исключением того, что неиспользуемые файлы WAL будут удалены. Любое другое использование может быть потенциально опасным и должно осуществляться с большой осторожностью. Для каталога после аварийного отключения сервера или повреждения управляющего файла `pg_resetwal` потребует указать параметр `-f` (принудительное выполнение).

После выполнения этой команды в каталоге данных с повреждёнными файлами WAL или управляющим файлом, запуск сервера, скорее всего, будет возможен, однако стоит учитывать, что база данных может содержать несогласованные данные из-за транзакций, зафиксированных частично. Необходимо немедленно выгрузить данные, выполнить `initdb`, а затем восстановить данные. После этого проверьте целостность базы и внесите необходимые коррективы.

Если `pg_resetwal` сообщает о невозможности определить данные из `pg_control`, команду можно запустить принудительно, указав `-f`. В этом случае будут использованы наиболее вероятные значения. Они должны подходить для большинства полей, но для некоторых может потребоваться задать нужные значения явно: следующее значение OID, ID и эпоха следующей транзакции, ID мультитранзакции и смещение, начальная позиция WAL. Эти значения можно указать с помощью описанных далее параметров. Если их невозможно определить, то флаг `f` позволяет это обойти. Однако достоверность данных восстановленной базы не гарантируется: крайне необходимо незамедлительно выгрузить и затем восстановить данные. *Не* выполняйте никаких операций модификации до создания дампа данных, так как это может привести к ещё более печальным последствиям.

Эту утилиту может запускать только пользователь, установивший сервер, так как ей нужны права записи/чтения в каталоге хранения данных кластера.

Параметры

каталог_данных
`-D` *каталог_данных*
`--pgdata=`*каталог_данных*

Указывает расположение каталога базы данных. В целях безопасности каталог данных необходимо указывать в командной строке. `pg_resetwal` не поддерживает переменную окружения `PGDATA`.

`-f`
`--force`

Принудить `pg_resetwal` продолжить работу даже в опасных ситуациях, описанных выше. В частности, этот параметр необходим для продолжения работы, если сервер был аварийно остановлен или если `pg_resetwal` не может определить корректные данные для файла `pg_control`.

`-n`
`--dry-run`

С ключом `-n`/`--dry-run` команда `pg_resetwal` отображает значения, извлечённые из `pg_control`, а также значения, которые планируется изменить, и завершается, не внося никаких изменений. Это, прежде всего, средство отладки, хотя оно может быть полезно и для того, чтобы проверить корректность параметров, прежде чем `pg_resetwal` начнёт что-либо делать.

`-V`
`--version`

Показать версию, а затем завершиться.

`-?`
`--help`

Показать справку, а затем завершиться.

Следующие параметры необходимы, только когда `pg_resetwal` не может определить подходящие значения, прочитав `pg_control`. Безопасные значения можно определить, как описано ниже. Для значений, принимающих числовые аргументы, можно задать шестнадцатеричные значения, добавив префикс `0x`. Обратите внимание, что эти инструкции относятся только к стандартному блоку размером 8 КБ.

`-c` *`xid`*`,`*`xid`*
`--commit-timestamp-ids=`*`xid`*`,`*`xid`*

Вручную задать идентификаторы старейшей и новейшей транзакций, для которых можно получить время фиксации.

Безопасное значение идентификатора старейшей транзакции, для которой можно получить время фиксации (первый компонент), можно определить, найдя наименьшее в числовом виде имя файла в каталоге `pg_commit_ts` внутри каталога данных. Безопасное значение идентификатора новейшей транзакции, для которой можно получить время фиксации (второй компонент), можно определить, найдя, напротив, наибольшее в числовом виде имя файла в том же каталоге. Числа в именах этих файлов представлены в шестнадцатеричном формате.

`-e` *`эпоха_xid`*
`--epoch=`*`эпоха_xid`*

Вручную задать эпоху в ID следующей транзакции.

Эпоха идентификаторов транзакции не хранится в базе данных нигде, кроме поля, устанавливаемого командой `pg_resetwal`, поэтому если рассматривать собственно базу, допустимым будет любое значение. Это значение, возможно, понадобится скорректировать для обеспечения правильной работы системы репликации, например, Slony-I и Skytools. В этом случае подходящее значение следует получить из состояния нижележащей реплицированной базы данных.

`-l` *`walfile`*
`--next-wal-file=`*`walfile`*

Вручную задать начальную позицию WAL, указав имя файла следующего сегмента WAL.

Имя файла следующего сегмента WAL должно превышать имена любых других файлов сегментов WAL, в настоящее время находящихся в подкаталоге `pg_wal` каталога данных. Эти имена тоже представлены в шестнадцатеричном виде и состоят из трёх частей. Первая из них — «ID линии времени» и её обычно не следует менять. Например, если `00000001000000320000004A` — наибольшее значение в `pg_wal`, нужно указать `-l 00000001000000320000004B` или большее число.

Заметьте, что при использовании нестандартных размеров сегментов WAL числа в именах файлов WAL отличаются от LSN, выдаваемых системными функциями и представлениями. Этот параметр принимает имя файла WAL, а не LSN.

Примечание

`pg_resetwal` ищет среди файлов каталога `pg_wal`, и по умолчанию выбирает значение для флага `-l`, идущее следующим после найденного. Таким образом, вручную задавать параметр `-l` нужно лишь если известно о существовании сегментов WAL, отсутствующих в настоящий момент в каталоге `pg_wal` (например, они могут находится в отдельном архиве); либо если содержимое `pg_wal` было полностью утеряно.

`-m` *`mxid`*`,`*`mxid`*
`--multixact-ids=`*`mxid`*`,`*`mxid`*

Вручную задать ID следующей и старейшей мультитранзакции.

Безопасное значение следующего идентификатора мультитранзакции (первый компонент) можно вычислить, найдя наибольшее числовое значение среди имён файлов, расположенных в каталоге `pg_multixact/offsets`. К найденному значению необходимо прибавить один, затем умножить на 65536 (0x10000). Для вычисления безопасного значения ID старейшей мультитранзакции (второй компонент `-m`) необходимо найти наименьшее числовое значение среди тех же файлов, и умножить его на 65536. Имена файлов представляются в шестнадцатеричном формате, так что значения проще указывать в нём же, добавив в конце четыре нуля.

`-o` *`oid`*
`--next-oid=`*`oid`*

Вручную задать следующий OID.

Не существует относительно простого способа вычисления следующего за наибольшим из существующих значением OID, однако это некритично.

`-O` *`mxoff`*
`--multixact-offset=`*`mxoff`*

Вручную задать смещение следующей мультитранзакции.

Безопасное значение можно определить, найдя наибольшее числовое значение в имени файла в каталоге `pg_multixact/members`, прибавив один и умножив результат на 52352 (0xCC80). Имена файлов представлены в шестнадцатеричном формате. Простого рецепта с прибавлением нулей в конце, как для других параметров, в данном случае нет.

`--wal-segsize=`*`размер_сегмента_wal`*

Задать другой размер сегмента WAL, в мегабайтах. Значение параметра должно быть степенью 2 от 1 до 1024 (в мегабайтах). Подробнее о нём можно узнать в описании такого же параметра initdb.

Этот параметр можно также использовать для изменения размера сегментов WAL существующего кластера баз данных, обходясь без повторного использования `initdb`.

Примечание

Хотя `pg_resetwal` выбирает стартовый адрес WAL, превышающий номер последнего существующего файла сегмента WAL, при некоторых изменениях размера предыдущие имена

> файлов WAL могут использоваться повторно. Если наложение имён файлов WAL приведёт к проблемам с вашей стратегией архивации, рекомендуется использовать `-l` вместе с этим ключом, чтобы вручную задать начальный адрес WAL.

`-u` *`xid`*
`--oldest-transaction-id=`*`xid`*

Вручную задать ID старейшей незамороженной транзакции.

Безопасное значение можно определить, найдя наименьшее числовое значение в имени файла в подкаталоге `pg_xact` каталога данных, а затем умножив результат на 1048576 (0x100000). Заметьте, что имена файлов представлены в шестнадцатеричном формате. Значение этого параметра обычно также проще задавать в шестнадцатеричном виде. Например, если `0007` — наименьшее значение в `pg_xact`, подходящим значением будет `-u 0x700000` (нужный множитель дают пять последних нулей).

`-x` *`xid`*
`--next-transaction-id=`*`xid`*

Вручную задать ID следующей транзакции.

Безопасное значение можно определить, найдя наибольшее числовое значение в имени файла в подкаталоге `pg_xact` каталога данных, прибавив один и умножив результат на 1048576 (0x100000). Заметьте, что имена файлов представлены в шестнадцатеричном формате. Значение этого параметра обычно также проще задавать в шестнадцатеричном виде. Например, если `0011` — наибольшее значение в `pg_xact`, подходящим значением будет `-x 0x1200000` (нужный множитель дают пять последних нулей).

Переменные окружения

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Эту команду нельзя выполнять на работающем сервере. `pg_resetwal` отклонит выполнение при обнаруженном блокирующем файле в каталоге хранения данных. Иногда при аварии сервера блокирующий файл может остаться в системе. В этом случае необходимо самостоятельно удалить его, чтобы дать возможность `pg_resetwal` отработать. Перед выполнением операции дважды проверьте, что сервер остановлен.

`pg_resetwal` работает только с серверами той же основной версии.

См. также

pg_controldata

pg_rewind

pg_rewind — синхронизировать каталог данных Postgres Pro с другим каталогом, ответвлённым от него

Синтаксис

`pg_rewind` [*параметр*...] { `-D` | `--target-pgdata` }*каталог* { `--source-pgdata=`*каталог* | `--source-server=`*строка_подключения* }

Описание

Утилита pg_rewind представляет собой средство синхронизации кластера Postgres Pro с другой копией того же кластера после расхождения линий времени этих кластеров. Обычный сценарий её использования — вернуть в работу старый ведущий сервер после переключения на ведомый, в качестве ведомого для сервера, ставшего ведущим.

После успешной синхронизации состояние целевого каталога данных будет аналогично состоянию базовой копии исходного каталога. В отличие от создания новой копии или использования такого средства, как rsync, синхронизация pg_rewind не требует сравнения или копирования неизменённых блоков в кластере. Из существующих файлов отношений копируются только изменённые блоки, а все остальные файлы, включая новые файлы отношений, файлы конфигурации и сегменты WAL переносятся полностью. Благодаря этому такая синхронизация гораздо быстрее других методов, когда база данных большая, а различия между кластерами ограничиваются лишь небольшим количеством блоков.

Утилита pg_rewind изучает истории линий времени исходного и целевого кластеров с целью найти точку, в которой они разошлись, и ожидает найти журналы WAL в каталоге `pg_wal` целевого кластера вплоть до точки расхождения. Точка расхождения может быть найдена на целевой или исходной линии времени либо в их общем предке. В типичном сценарии отработки отказа, когда целевой кластер отключается вскоре после расхождения, это не проблема, но если целевой кластер проработал долгое время после расхождения, старые файлы WAL могут быть уже удалены. В этом случае их можно вручную скопировать из архива WAL в каталог `pg_wal` или запустить pg_rewind с ключом `-c`, чтобы они были автоматически получены из архива WAL. Варианты использования pg_rewind не ограничиваются отработкой отказа; например, резервный сервер может быть повышен, выполнить несколько пишущих транзакций, а затем, после восстановления синхронизации, вновь стать резервным.

После выполнения pg_rewind для приведения каталога данных в согласованное состояние должна завершиться процедура воспроизведения WAL. Когда целевой сервер запускается, он переходит в режим восстановления из архива и воспроизводит все изменения из WAL с исходного сервера, накопившиеся после последней контрольной точки с момента расхождения. Если какие-то сегменты WAL оказались недоступны на исходном сервере, когда выполнялась pg_rewind, и поэтому не могли быть скопированы в ходе работы pg_rewind, их необходимо предоставить, когда сервер будет запускаться. Это можно сделать, создав файл `recovery.signal` в целевом каталоге данных и определив подходящую команду restore_command в `postgresql.conf`.

Утилита pg_rewind требует, чтобы на целевом сервере был либо включён режим wal_log_hints в `postgresql.conf`, либо включены контрольные суммы, когда кластер был инициализирован командой initdb. По умолчанию режим `wal_log_hints` отключён, а контрольные суммы включены. Также должен быть включён режим full_page_writes (по умолчанию он включён).

Предупреждение: ошибки при синхронизации

Если во время работы pg_rewind происходит ошибка, вероятнее всего, целевой каталог данных будет в состоянии, не подходящем для восстановления. В этом случае рекомендуется сделать новую резервную копию.

> Так как pg_rewind копирует файлы конфигурации с исходного сервера без изменений, в полученную конфигурацию может потребоваться внести коррективы до перезапуска целевого сервера, особенно если целевой сервер становится ведомым для исходного. Если вы перезапустите сервер после окончания операции синхронизации, но не настроите команду восстановления WAL, целевой сервер может снова разойтись с ведущим.
>
> Программа pg_rewind немедленно прекращает работу, если обнаруживает файлы, непосредственная запись в которые невозможна. Это может иметь место, например, когда на исходном и целевом серверах совпадают пути файлов сертификатов и ключей SSL, доступных только для чтения. Если такие файлы существуют на целевом сервере, их рекомендуется удалить до запуска pg_rewind. После выполнения синхронизации некоторые из таких файлов могут быть скопированы из источника и тогда может потребоваться удалить скопированные данные и восстановить ссылки/файлы, существовавшие до синхронизации.

Параметры

pg_rewind принимает следующие аргументы командной строки:

`-D` *`каталог`*
`--target-pgdata=`*`каталог`*

Этот параметр задаёт целевой каталог данных, который будет синхронизирован с источником. Целевой сервер должен быть отключён штатным образом до запуска pg_rewind

`--source-pgdata=`*`каталог`*

Задаёт путь в файловой системе к каталогу данных исходного сервера, с которым будет синхронизироваться целевой. Для применения этого ключа исходный сервер должен быть остановлен штатным образом.

`--source-server=`*`строка_подключения`*

Задаёт строку подключения libpq для подключения к исходному серверу Postgres Pro, с которым будет синхронизирован целевой. Подключение должно устанавливаться как обычное (не реплицирующее) от имени роли, имеющей необходимые права для выполнения функций pg_rewind на исходном сервере (подробнее об этом говорится в Замечаниях), или от имени суперпользователя. Для применения этого параметра исходный сервер должен быть запущен и принимать подключения.

`-R`
`--write-recovery-conf`

Создать `standby.signal` и добавить параметры подключения в `postgresql.auto.conf` в выходном каталоге. Этот ключ требует указания `--source-server`.

`-n`
`--dry-run`

Делать всё, кроме внесения изменений в целевой каталог.

`-N`
`--no-sync`

По умолчанию `pg_rewind` ждёт, пока все файлы не будут надёжно записаны на диск. С данным параметром `pg_rewind` завершается быстрее, без ожидания, но в случае неожиданного сбоя операционной системы целевой каталог данных может оказаться испорченным. Этот параметр может быть полезен при тестировании; в производственной среде применять его не следует.

`-P`
`--progress`

Включает вывод сообщений о прогрессе. При этом в процессе копирования данных из исходного кластера будет выдаваться приблизительный процент выполнения.

`-c`
`--restore-target-wal`

Использовать команду `restore_command`, определённую в конфигурации целевого кластера, для получения файлов WAL из архива в случае отсутствия их в каталоге `pg_wal`.

`--config-file=`*`имя_файла`*

Использовать указанный основной файл конфигурации сервера для целевого кластера. Это влияет на поведение программы postgres, которую вызывает pg_rewind в процессе операции синхронизации в данном кластере (для получения `restore_command` при вызове с параметром `-c`/`--restore-target-wal` и для выполнения процедуры восстановления после сбоя).

`--debug`

Выводить подробные отладочные сообщения, полезные в основном для разработчиков, отлаживающих pg_rewind.

`--no-ensure-shutdown`

Для pg_rewind необходимо, чтобы целевой сервер был остановлен штатным образом до синхронизации. По умолчанию, если целевой сервер не был остановлен штатно, pg_rewind запускает его в однопользовательском режиме, чтобы он выполнил процедуру восстановления после сбоя, а затем останавливает его. Когда передаётся данный ключ, pg_rewind не делает этого, а завершается с ошибкой немедленно, если сервер не был остановлен корректно. Ожидается, что в этом случае пользователи сами исправят сложившуюся ситуацию.

`--sync-method=`*`метод`*

Со значением `fsync` (по умолчанию) `pg_rewind` будет рекурсивно открывать и синхронизировать все файлы в каталоге данных. Поиск файлов будет осуществляться по символическим ссылкам для каталога WAL и каждого настроенного табличного пространства.

В Linux возможен вариант `syncfs`, когда от ОС требуется синхронизировать целиком каждую из файловых систем, содержащих каталог данных, файлы WAL и табличные пространства. Более подробно особенности использования `syncfs` описаны в recovery_init_sync_method.

В режиме `--no-sync` этот параметр не оказывает никакого влияния.

`-V`
`--version`

Показать версию, а затем завершиться.

`-?`
`--help`

Показать справку, а затем завершиться.

Переменные окружения

Когда используется `--source-server`, pg_rewind также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Когда исходным кластером для pg_rewind является работающий сервер, вместо суперпользователя может применяться роль, имеющая в этом кластере достаточные права для выполнения функций, которые использует pg_rewind. Такую роль (`rewind_user`) можно создать так:

```
CREATE USER rewind_user LOGIN;
```

```
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text, bigint, bigint, boolean)
 TO rewind_user;
```

Как это работает

Основная идея состоит в том, чтобы перенести все изменения на уровне файловой системы из исходного кластера в целевой:

1. Просканировать журнал WAL в целевом кластере, начиная с последней контрольной точки перед моментом, когда история линии времени исходного кластера разошлась с целевым кластером. Для каждой записи WAL отметить, какие блоки данных были затронуты. В результате будет получен список всех блоков данных, которые были изменены в целевом кластере после отделения исходного. Если какие-либо файлы WAL уже удалены, можно запустить pg_rewind с ключом `-c`, чтобы целевой сервер обратился за ними к архиву WAL.
2. Скопировать все эти изменённые блоки из исходного кластера в целевой либо на уровне файловой системы (`--source-pgdata`), либо на уровне SQL (`--source-server`). После этого файлы отношений оказываются в том же состоянии, в котором они были в момент последней контрольной точки, предшествующей моменту расхождения линий времени исходного и целевого кластера, с добавлением текущего состояния, полученного из исходного кластера, тех блоков, которые были изменены в целевом кластере после расхождения.
3. Скопировать все остальные файлы, включая новые файлы отношений, сегменты WAL, `pg_xact` и файлы конфигурации, из исходного кластера в целевой. Как и при базовом копировании, содержимое каталогов `pg_dynshmem/`, `pg_notify/`, `pg_replslot/`, `pg_serial/`, `pg_snapshots/`, `pg_stat_tmp/` и `pg_subtrans/` исключается из данных, копируемых из исходного кластера. Кроме того, исключаются файлы или каталоги с именами, начинающимися с `pgsql_tmp`, а также файлы `backup_label`, `tablespace_map`, `pg_internal.init`, `postmaster.opts`, `postmaster.pid` и `.DS_Store`.
4. Создать файл `backup_label` для перехода к воспроизведению WAL в контрольной точке, произведённой при переключении, и установить в файле `pg_control` минимальный LSN согласованного состояния, определённый в результате вызова `pg_current_wal_insert_lsn()` при синхронизации с работающим источником или равный LSN последней контрольной точке при синхронизации с остановленным.
5. При запуске целевого сервера Postgres Pro воспроизводятся все требуемые записи WAL, в результате чего каталог данных приводится в согласованное состояние.

pg-setup

pg-setup — создать кластер баз данных Postgres Pro и управлять соответствующей службой

Синтаксис

`pg-setup initdb` [*`параметры_initdb`*]

`pg-setup find-free-port`

`pg-setup set-server-port` *`порт`*

`pg-setup set` *`параметр`* *`значение`*

`pg-setup service` *`параметр_service`*

Описание

Программа pg-setup представляет собой скрипт оболочки, включённый в дистрибутив Postgres Pro и предназначенный для автоматизации установки кластера баз данных в системах Linux. Этот скрипт входит в состав пакета `postgrespro-std-17-server`. После установки Postgres Pro вы можете найти pg-setup в каталоге *`каталог-инсталляции`*`/bin`, где *`каталог-инсталляции`* обозначает путь `/opt/pgpro/std-17`.

Программу pg-setup нужно запускать от имени root, но она выполняет административные операции с базой данных под именем пользователя `postgres`. Запуская её с разными параметрами, вы можете:

- инициализировать кластер баз данных
- настроить кластер БД для определённого продукта Postgres Pro
- проверить доступные порты и изменить порт, который будет использоваться сервером Postgres Pro
- включить/отключить автоматический запуск службы Postgres Pro
- запустить, остановить или перезапустить службу Postgres Pro

Параметры

pg-setup принимает следующие аргументы командной строки:

`initdb [`*`параметры_initdb`*`]`

Инициализировать кластер баз данных от имени пользователя `postgres`.

По умолчанию кластер баз данных инициализируется в каталоге `/var/lib/pgpro/std-17/data` и настраивается для продукта Postgres Pro; в этом кластере будут включены контрольные суммы, параметр `auth-local` будет иметь значение `peer`, а `auth-host` — значение `md5`. Параметры локализации наследуются из переменной окружения `LANG`, определённой в текущем сеансе. Все переменные окружения `LC_*` при этом игнорируются. Вы можете также передать параметры initdb для дополнительной настройки инсталляции.

Если база данных по умолчанию создаётся с помощью pg-setup, путь к каталогу данных сохраняется в файле `/etc/default/postgrespro-standard-17`, чтобы все последующие команды pg-setup, а также все команды, управляющие службой Postgres Pro, воздействовали именно на эту базу данных. Управлять несколькими базами данных с помощью pg-setup нельзя.

В случае, если в вашей системе будут работать несколько серверов БД и/или приложений, может потребоваться изменить настроенную конфигурацию, так как pg-setup выбирает параметры конфигурации в зависимости от характеристик оборудования, полагая, что система будет использоваться только одним сервером БД.

`find-free-port`

Найти свободный порт в системе. Это может быть полезно, если вы планируете устанавливать несколько экземпляров серверов, либо стандартный порт 5432 уже используется другой программой.

`set-server-port` *`порт`*

Задать номер порта, на котором сервер будет принимать соединения. Воспользуйтесь этой командой, чтобы избежать конфликтов, если вы устанавливаете в одной системе более одного сервера.

По умолчанию: 5432

`set` *`имя значение`*

Установить для заданного параметра конфигурации указанное *`значение`* в файле `postgresql.conf`. Если этот параметр ранее был определён командой `ALTER SYSTEM`, его предыдущее значение удаляется из файла `postgresql.auto.conf`.

`service` *`параметр_service`*

Управление службой Postgres Pro с помощью следующих команд:

- `enable` — включить автоматический запуск службы при перезагрузке системы.
- `disable` — отключить автоматический запуск службы при перезагрузке системы.
- `start` — запустить службу.
- `stop` — остановить службу.
- `condrestart` — перезапустить службу, если она работает в момент вызова pg-setup.
- `status` — возвращает статус службы Postgres Pro.

Замечания

Если вы устанавливаете Postgres Pro из пакета `postgrespro-std-17`, pg-setup вызывается автоматически с параметрами по умолчанию. В результате создаётся кластер баз данных и база по умолчанию в каталоге `/var/lib/pgpro/std-17/data`, настраивается автозапуск службы Postgres Pro и служба запускается.

Если вы устанавливаете сервер Postgres Pro непосредственно из пакета `postgrespro-std-17-server`, вы можете запустить этот скрипт вручную для инициализации кластера баз данных или управления службой Postgres Pro.

Установка из двоичных пакетов в Linux подробно описана в Разделе 16.1.

pg_test_fsync

pg_test_fsync — подобрать наилучший вариант `wal_sync_method` для Postgres Pro

Синтаксис

`pg_test_fsync` [*параметр*...]

Описание

Программа pg_test_fsync предназначена для того, чтобы дать вам представление о том, какой из вариантов wal_sync_method оптимален для вашей конкретной системы, а также выдать вспомогательные диагностические сведения в случае проблем со вводом-выводом. Однако отличия, показанные программой pg_test_fsync, могут не оказывать большого влияния на реальную производительность баз данных, в частности потому, что для многих серверов баз данных производительность упирается не в запись журналов предзаписи. pg_test_fsync выводит среднее время операции синхронизации с ФС для каждого метода `wal_sync_method`, что также может быть полезно при поиске оптимального значения commit_delay.

Параметры

pg_test_fsync принимает следующие параметры командной строки:

`-f`
`--filename`

Задаёт имя файла для записи данных тестов. Этот файл должен находиться в той же файловой системе, где размещается или будет размещаться каталог `pg_wal`. (В каталоге `pg_wal` содержатся файлы WAL.) По умолчанию выбирается файл `pg_test_fsync.out` в текущем каталоге.

`-s`
`--secs-per-test`

Задаёт продолжительность каждого теста (в секундах). Чем больше длится тест, тем точнее результат, но тем дольше работает программа. Со значением по умолчанию (5 секунд) программа должна завершиться примерно за 2 минуты.

`-V`
`--version`

Вывести версию pg_test_fsync и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_test_fsync и завершиться.

Переменные окружения

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

См. также

postgres

pg_test_timing

pg_test_timing — определить издержки замера времени

Синтаксис

`pg_test_timing` [*параметр*...]

Описание

Программа pg_test_timing позволяет оценить издержки замера времени в вашей системе и убедиться в том, что системное время никогда не идёт назад. Системы, в которых замер времени является длительной операцией, дают менее точные результаты `EXPLAIN ANALYZE`.

Параметры

pg_test_timing принимает следующие аргументы командной строки:

`-d` *`длительность`*
`--duration=`*`длительность`*

Задаёт продолжительность теста (в секундах). Чем больше эта продолжительность, тем выше точность и больше вероятность обнаружить аномалию с обратным ходом системных часов. По умолчанию время тестирования — 3 секунды.

`-V`
`--version`

Вывести версию pg_test_timing и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки pg_test_timing и завершиться.

Использование

Интерпретация результатов

В благоприятном случае практически все (>90%) отдельные вызовы замеров времени должны выполняться быстрее одной микросекунды. Средние издержки замера на цикл должны быть ещё меньше, в пределах 100 наносекунд. Эта проба, взятая в системе Intel i7-860 через источник времени TSC, показывает отличную производительность:

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 ns
Histogram of timing durations:
  < us   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

Заметьте, что время вызова в цикле и время в гистограмме выражается в разных единицах. Время в цикле может определяться с точностью до наносекунд (ns), а длительность отдельного вызова замера времени — только с точностью до микросекунд (us).

Измерение издержек исполнителя на замер времени

Когда исполнитель запроса выполняет запрос под контролем `EXPLAIN ANALYZE`, замеряется не только общее время, но и время отдельных операций. Каковы издержки этих операций в вашей системе, можно узнать, подсчитав строки тестовой таблицы в программе psql:

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

В системе i7-860 этот запрос выполняется 9.8 мс, а версия с `EXPLAIN ANALYZE` — 16.6 мс, при этом обрабатывается около 100 000 строк. Это различие в 6.8 мс означает, что издержки замера времени для одной строки составляют около 68 нс, примерно вдвое больше, чем предсказала pg_test_timing. Даже с такими относительно небольшими издержками операция COUNT с полным подсчётом времени выполняется почти на 70% дольше. На более сложных запросах издержки замера времени могут быть не так важны.

Смена источника времени

В некоторых современных системах Linux можно в любой момент сменить источник времени, который используется для замера времени. Второй пример иллюстрирует возможное замедление от переключения на более медленный источник времени acpi_pm time в той же системе, в которой были получены показанные выше хорошие результаты:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 ns
Histogram of timing durations:
  < us   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

В этой конфигурации тот же `EXPLAIN ANALYZE` выполняется 115.9 мс. Таким образом издержки составили 1061 нс, что соответствует непосредственному результату этой утилиты с небольшим коэффициентом. Такие большие издержки означают, что сам запрос выполняется лишь небольшой процент всего времени, а основное время уходит на замеры времени. В такой конфигурации временные показатели `EXPLAIN ANALYZE` для запросов со множеством замеряемых операций значительно увеличатся за счёт издержек замера времени.

FreeBSD так же позволяет сменять источник времени «на лету» и выводит информацию о выбранном таймере при загрузке:

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

Другие системы могут допускать смену источника времени только при загрузке. В старых системах Linux это можно было сделать только с помощью параметра ядра «clock». И даже в некоторых самых последних системах можно увидеть только один источник времени — jiffies. Это старая программная реализация часов в Linux, которая может давать хорошее разрешение, когда поддерживается достаточно хорошим оборудованием, как в этом примере:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
```

```
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  < us   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

Аппаратные часы и точность замера времени

Измерение времени обычно осуществляется на компьютерах по аппаратным часам, точность которых может быть разного уровня. С некоторым оборудованием операционные системы могут передавать время системных часов непосредственно программам. Также системное время может поступать с чипа, который просто генерирует прерывания по времени, с заведомо известным периодом. В любом случае ядра операционных систем предоставляют источник времени, который скрывает эти детали. Но точность этого источника и возможная скорость получения результатов от него зависит от нижележащего оборудования.

Неточность в замерах времени может приводить к нестабильности системы. Поэтому стоит очень тщательно протестировать выбранный источник времени. Иногда по умолчанию в ОС выбирается источник не более точный, а более надёжный. И если вы используете виртуальную машину, поинтересуйтесь, какие источники времени рекомендуется использовать с ней. Имитация таймеров на виртуальном оборудовании связана с дополнительными сложностями, и производители средств виртуализации часто рекомендуют определённые параметры для операционных систем.

Источник времени TSC (Time Stamp Counter, Счётчик отметки времени) наиболее точный из всех для процессоров текущего поколения. Его рекомендуется использовать для получения системного времени, когда он поддерживается операционной системой и показания TSC надёжны. Возможны ситуации, когда TSC не является точным источником времени, и таким образом, оказывается ненадёжным. Например, в старых системах показания TSC могут зависеть от температуры процессора, что не годится для точного замера времени. При попытке использовать TSC в некоторых старых многоядерных процессорах можно получить разное время на различных ядрах. В результате может оказаться, что время идёт назад (эту аномалию выявляет данная программа). И даже на самых современных системах не всегда можно получить точное время через TSC в режимах очень агрессивного энергосбережения.

Новые операционные системы могут проверять наличие известных проблем TSC и переключаться на более медленный, но более стабильный источник времени, если они проявляются. Если ваша система поддерживает источник TSC, но не выбирает его по умолчанию, возможно, он отключён обоснованно. С другой стороны, некоторые операционные системы могут не выявлять все возможные проблемы, либо разрешают использовать TSC даже в ситуациях, когда он определённо неточен.

HPET (High Precision Event Timer, Таймер событий высокой точности) рекомендуется использовать в системах, где он имеется, а TSC неточен. Сам этот чип можно запрограммировать для получения точности до 100 наносекунд, но системное время с такой точностью вы не получите.

ACPI (Advanced Configuration and Power Interface, Расширенный интерфейс конфигурации и питания) обеспечивает таймер PM (Управления питанием), который в Linux называется acpi_pm. Время, поступающее из acpi_pm, в лучшем случае будет иметь разрешение 300 наносекунд.

На старом оборудовании PC применялись таймеры 8254 PIT (Programmable Interval Timer, Программируемый интервальный таймер), RTC (Real-Time Clock, Часы реального времени), таймер APIC (Advanced Programmable Interrupt Controller, Расширенный программируемый контроллер прерываний) и Cyclone. Все эти таймеры обеспечивали точность до миллисекунд.

См. также

EXPLAIN

pg_upgrade

pg_upgrade — обновить экземпляр сервера Postgres Pro

Синтаксис

`pg_upgrade` `-b` *`старый_каталог_bin`* [`-B` *`новый_каталог_bin`*] `-d` *`старый_каталог_конфигурации`* `-D` *`новый_каталог_конфигурации`* [*`параметр`*...]

Описание

Программа pg_upgrade (ранее называвшаяся pg_migrator) позволяет обновить данные в каталоге базы PostgreSQL или Postgres Pro до последней основной версии Postgres Pro без операции выгрузки/восстановления данных, обычно необходимой при обновлениях основной версии, например, при переходе от 12.14 к 13.10 или от 14.9 к 15.5. Эти действия не требуются при установке корректирующей версии, например, при переходе от 12.7 к 12.8 или от 14.1 к 14.5.

С выходом новых основных версий в Postgres Pro регулярно добавляются новые возможности, которые часто меняют структуру системных таблиц, но внутренний формат хранения меняется редко. Учитывая этот факт, pg_upgrade позволяет выполнить быстрое обновление, создавая системные таблицы заново, но сохраняя старые файлы данных. Если при обновлении основной версии формат хранения данных изменится так, что данные в старом формате окажутся нечитаемыми, pg_upgrade не сможет произвести такое обновление. (Сообщество разработчиков постарается не допустить подобных ситуаций.)

Программа pg_upgrade делает всё возможное, чтобы убедиться в том, что старый и новый кластеры двоично-совместимы, в частности проверяя параметры времени компиляции и разрядность (32/64 бита) исполняемых файлов. Важно, чтобы и все внешние модули тоже были двоично-совместимыми, хотя это pg_upgrade проверить не может.

pg_upgrade поддерживает обновление с версии 9.2.X и новее до текущей основной версии Postgres Pro, включая бета-выпуски и сборки снимков кода.

> **Предупреждение**
>
> При обновлении кластера выполняется произвольный код по выбору суперпользователей исходной точки. Перед обновлением убедитесь, что суперпользователи исходной точки являются доверенными.

Параметры

pg_upgrade принимает следующие аргументы командной строки:

`-b` *`каталог_bin`*
`--old-bindir=`*`каталог_bin`*

> каталог с исполняемыми файлами старой версии Postgres Pro; переменная окружения `PGBINOLD`

`-B` *`каталог_bin`*
`--new-bindir=`*`каталог_bin`*

> каталог с исполняемыми файлами новой версии Postgres Pro, по умолчанию это каталог, в котором располагается pg_upgrade; переменная окружения `PGBINNEW`

`-c`
`--check`

> только проверить кластеры, не изменять никакие данные

`-d` *`каталог_конфигурации`*
`--old-datadir=`*`каталог_конфигурации`*

каталог конфигурации старого кластера; переменная окружения `PGDATAOLD`

`-D` *`каталог_конфигурации`*
`--new-datadir=`*`каталог_конфигурации`*

каталог конфигурации нового кластера; переменная окружения `PGDATANEW`

`-j` *`число_заданий`*
`--jobs=`*`число_заданий`*

число одновременно задействуемых процессов или потоков

`-k`
`--link`

использовать жёсткие ссылки вместо копирования файлов в новый кластер

`-N`
`--no-sync`

По умолчанию `pg_upgrade` ждёт, пока все файлы обновлённого кластера не будут надёжно записаны на диск. С данным параметром `pg_upgrade` завершается быстрее, без ожидания, но в случае последующего сбоя операционной системы каталог данных может оказаться испорченным. Этот параметр может быть полезен при тестировании; в производственной среде применять его не следует.

`-o` *`параметры`*
`--old-options` *`параметры`*

параметры, передаваемые непосредственно старой программе `postgres`; несколько параметров складываются вместе

`-O` *`параметры`*
`--new-options` *`параметры`*

параметры, передаваемые непосредственно новой программе `postgres`; несколько параметров складываются вместе

`-p` *`порт`*
`--old-port=`*`порт`*

номер порта старого кластера; переменная окружения `PGPORTOLD`

`-P` *`порт`*
`--new-port=`*`порт`*

номер порта нового кластера; переменная окружения `PGPORTNEW`

`-r`
`--retain`

сохранить SQL и журналы сообщений даже при успешном завершении

`-s` *`каталог`*
`--socketdir=`*`каталог`*

каталог, в котором будет создавать сокеты процесс postmaster во время обновления; по умолчанию выбирается текущий рабочий каталог; переменная окружения `PGSOCKETDIR`

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

имя пользователя, установившего кластер; переменная окружения `PGUSER`

`-v`
`--verbose`

включить подробные внутренние сообщения

`-V`
`--version`

показать версию, а затем завершиться

`--clone`

Использовать эффективное клонирование файлов (в ряде систем это называется «reflink») вместо копирования файлов в новый кластер. В результате файлы данных могут копироваться практически мгновенно, как и с использованием `-k`/`--link`, но последующие изменения не будут затрагивать старый кластер.

Клонирование файлов поддерживается не во всех операционных системах и только с определёнными файловыми системами. Если этот режим выбран, но клонирование не поддерживается, при выполнении pg_upgrade произойдёт ошибка. В настоящее время оно поддерживается в Linux (с ядром 4.5 или новее) с Btrfs и XFS (если файловая система была создана с поддержкой reflink), а также в macOS с APFS.

`--copy`

Копировать файлы в новый кластер. Это поведение по умолчанию. (См. также `--link` и `--clone`.)

`--copy-file-range`

Использовать системный вызов `copy_file_range` для эффективного копирования. В некоторых файловых системах результаты будут схожи с `--clone`, позволяя совместное использование физических дисковых блоков, в то время как на других могут копироваться блоки, но по оптимизированному пути. В настоящее время параметр поддерживается в Linux и FreeBSD.

`--sync-method=`*метод*

Со значением `fsync` (по умолчанию) `pg_upgrade` рекурсивно откроет и синхронизирует все файлы в каталоге данных обновлённого кластера. Поиск файлов будет осуществляться по символическим ссылкам для каталога WAL и каждого настроенного табличного пространства.

В Linux возможен вариант `syncfs`, когда от ОС требуется синхронизировать целиком каждую из файловых систем, содержащих каталог данных обновлённого кластера, файлы WAL и табличные пространства. Более подробно особенности использования `syncfs` описаны в recovery_init_sync_method.

В режиме `--no-sync` этот параметр не оказывает никакого влияния.

`-?`
`--help`

показать справку, а затем завершиться

Использование

Далее описан план обновления с использованием pg_upgrade:

1. **Переместить старый кластер (необязательно)**

   Если ваш каталог инсталляции привязан к версии, например, `/opt/PostgreSQL/17`, перемещать его не требуется. Все графические инсталляторы выбирают при установке каталоги, привязанные к версии.

   Если ваш каталог инсталляции не привязан к версии, например `/usr/local/pgsql`, необходимо переместить каталог текущей инсталляции Postgres Pro, чтобы он не конфликтовал с новой инсталляцией Postgres Pro. Когда текущий сервер Postgres Pro отключён, каталог этой инсталляции Postgres Pro можно безопасно переместить; если старый каталог `/usr/local/pgsql`, его можно переименовать, выполнив:

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

2. **Установить новые исполняемые файлы Postgres Pro**

   Установите новые исполняемые файлы сервера и вспомогательные файлы. Программа pg_upgrade включена в инсталляцию по умолчанию.

3. **Инициализировать новый кластер Postgres Pro**

   Инициализируйте новый кластер, используя `initdb`. При этом так же необходимо указать флаги `initdb`, совместимые с флагами в старом кластере. Многие готовые инсталляторы выполняют это действие автоматически. Запускать новый кластер не требуется.

4. **Установить разделяемые объектные файлы расширения**

   Многие расширения и пользовательские модули, как из `contrib`, так и из других источников, используют разделяемые объектные файлы (или библиотеки DLL), например, `pgcrypto.so`. Если они использовались в старом кластере, разделяемые объектные файлы, соответствующие новому исполняемому файлу сервера, должны быть установлены в новом кластере, обычно средствами операционной системы. Не загружайте определения схемы, например, выполняя `CREATE EXTENSION pgcrypto`, потому что они будут скопированы из старого кластера. Если доступны обновления расширений, pg_upgrade сообщит об этом и создаст скрипт, который можно будет запустить позже, чтобы обновить эти расширения.

5. **Скопировать пользовательские файлы полнотекстового поиска**

   Скопировать пользовательские файлы полнотекстового поиска (словари, тезаурусы, списки синонимов и стоп-слов) из старого кластера в новый.

6. **Настроить аутентификацию**

   Программа `pg_upgrade` будет подключаться к новому и старому серверу несколько раз, так что имеет смысл установить режим аутентификации `peer` в `pg_hba.conf` или использовать файл `~/.pgpass` (см. Раздел 32.16).

7. **Подготовиться к обновлению публикующего сервера**

   Программа pg_upgrade пытается мигрировать логические слоты. Это устраняет необходимость определять те же логические слоты на новом публикующем сервере вручную. Миграция логических слотов поддерживается только в случае, если старый кластер версии 17.0 или выше. Логические слоты на кластерах до версии 17.0 будут игнорироваться без уведомления.

   Прежде чем начать обновление публикующего кластера, убедитесь, что подписка временно отключена, выполнив команду `ALTER SUBSCRIPTION ... DISABLE`. Включите подписку после завершения обновления.

   Необходимо выполнить ряд предварительных требований, чтобы программа pg_upgrade смогла обновить логические слоты. В противном случае будет выдаваться ошибка.

   - В новом кластере `wal_level` должен иметь значение `logical`.
   - В новом кластере параметр `max_replication_slots` должен иметь значение, большее или равное числу слотов в старом кластере.
   - Модули вывода, на которые ссылаются слоты на старом кластере, должны быть установлены в каталоге с исполняемыми файлами новой версии Postgres Pro.
   - В старом кластере все транзакции и сообщения логического декодирования передавались подписчикам.
   - Все слоты в старом кластере должны использоваться, т. е. не должно быть слотов, в которых pg_replication_slots.`conflicting` не имеет значение `true`.

- В новом кластере не должно быть постоянных логических слотов, т. е. не должно быть слотов, в которых pg_replication_slots.`temporary` имеет значение `false`.

8. **Подготовиться к обновлению подписчика**

   Настройте конфигурации подписчика на новом подписчике. pg_upgrade пытается мигрировать зависимости подписки, включая информацию о таблице подписки, представленную в системном каталоге pg_subscription_rel, и об источнике репликации подписки. Таким образом логическая репликация для нового подписчика может продолжаться с того места, где остановился старый. Миграция зависимостей подписки поддерживается только в случае, если старый кластер версии 17.0 или выше. Зависимости подписки на кластерах до версии 17.0 будут игнорироваться без уведомления.

   Чтобы программа pg_upgrade смогла обновить подписки, должны выполняться определённые условия. В противном случае будет выдаваться ошибка.

   - Все таблицы подписки в старом подписчике должны быть в состоянии `i` (initialize, инициализация) или `r` (ready, готовность). Это можно проверить с помощью pg_subscription_rel.`srsubstate`.
   - В старом кластере должна быть запись об источнике репликации для каждой из подписок. Это можно проверить в системных таблицах pg_subscription и pg_replication_origin.
   - В новом кластере параметр `max_replication_slots` должен иметь значение, большее или равное числу подписок в старом.

9. **Остановить оба сервера**

   Убедитесь в том, что оба сервера баз данных остановлены. Для этого в Unix можно выполнить:

   ```
   pg_ctl -D /opt/PostgreSQL/12 stop
   pg_ctl -D /opt/PostgreSQL/17 stop
   ```

   А в Windows, с соответствующими именами служб:

   ```
   NET STOP postgresql-12
   NET STOP postgresql-17
   ```

   Ведомые серверы с потоковой репликацией и трансляцией журнала должны продолжать работать во время этого отключения, чтобы получить все изменения.

10. **Подготовиться к обновлению ведомых серверов**

    Если вы производите обновление ведомых серверов (как описано в разделе Шаг 12), удостоверьтесь, что эти серверы находятся в актуальном состоянии, запустив pg_controldata в старых ведущем и ведомых кластерах. Убедитесь в том, что «Положение последней контрольной точки» во всех кластерах одинаковое. Также смените `wal_level` на `replica` в файле `postgresql.conf` нового ведущего кластера.

11. **Запустить pg_upgrade**

    Всегда запускайте программу pg_upgrade от нового сервера, а не от старого. pg_upgrade требует указания каталогов данных старого и нового кластера, а также каталогов исполняемых файлов (`bin`). Вы можете также определить имя пользователя и номера портов, и нужно ли копировать файлы данных (по умолчанию), клонировать их или создавать ссылки на них.

    Если выбрать вариант со ссылкой на данные, обновление выполнится гораздо быстрее (так как файлы не копируются) и потребует меньше места на диске, но вы лишитесь возможности обращаться к вашему старому кластеру, запустив новый после обновления. Этот вариант также требует, чтобы каталоги данных старого и нового кластера располагались в одной файловой системе. (Табличные пространства и `pg_wal` могут находиться в других файловых системах.) Вариант с клонированием работает так же быстро и экономит место на диске, но позволяет сохранить рабочее состояние старого кластера при запуске нового. Для этого варианта тоже требуется, чтобы старый и новый каталоги данных находились в одной файловой системе. Клонирование возможно только в некоторых операционных системах с определёнными файловыми системами.

Параметр `--jobs` позволяет задействовать для копирования/связывания файлов и для выгрузки/восстановления схем баз данных несколько процессорных ядер. В качестве начального значения параметра стоит выбрать максимум из числа процессорных ядер и числа табличных пространств. Этот параметр может радикально сократить время обновления сервера со множеством баз данных, работающего в многопроцессорной системе.

В Windows вы должны войти в систему с административными полномочиями и затем запустить pg_upgrade с путями каталогов в кавычках, например, так:

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/PostgreSQL/12/data"
        --new-datadir "C:/Program Files/PostgreSQL/17/data"
        --old-bindir "C:/Program Files/PostgreSQL/12/bin"
        --new-bindir "C:/Program Files/PostgreSQL/17/bin"
```

При запуске `pg_upgrade` проверит два кластера на совместимость и, если всё в порядке, выполнит обновление. Также возможно запустить `pg_upgrade --check`, чтобы ограничиться только проверками (при этом старый сервер может продолжать работать). Команда `pg_upgrade --check` также сообщит, какие коррективы вам нужно будет внести вручную после обновления. Если вы планируете использовать режим ссылок или клонирования, укажите вместе с параметром `--check` параметр `--link` или `--clone`, чтобы включить специальные проверки для этих режимов. Программе `pg_upgrade` требуются права на запись в текущий каталог.

Очевидно, никто не должен обращаться к кластерам в процессе обновления. Программа pg_upgrade по умолчанию запускает серверы с портом 50432, чтобы не допустить нежелательных клиентских подключений. В процессе обновления оба кластера могут использовать один номер порта, так как они не будут работать одновременно. Однако для проверки старого работающего сервера новый порт должен отличаться от старого.

Если при восстановлении схемы базы данных происходит ошибка, `pg_upgrade` завершает свою работу и вы должны вернуться к старому кластеру, как описывается ниже в Шаг 18. Чтобы попробовать `pg_upgrade` ещё раз, вы должны внести коррективы в старом кластере, чтобы pg_upgrade могла успешно восстановить схему. Если проблема возникла в модуле `contrib`, может потребоваться удалить этот модуль `contrib` в старом кластере, а затем установить его в новом после обновления (предполагается, что этот модуль не хранит пользовательские данные).

12. **Обновить ведомые серверы с потоковой репликацией и трансляцией журнала**

    Если вы используете режим ссылок и у вас реализована потоковая репликация (см. Подраздел 25.2.5) или трансляция журнала (см. Раздел 25.2) для ведомых серверов, вы можете быстро обновить эти серверы следующим образом. Вам не нужно будет запускать на них pg_upgrade, вместо этого вы выполните rsync на ведущем. Не запускайте никакие серверы на этом этапе.

    Если вы *не* используете режим ссылок, либо у вас нет rsync или вы не хотите его использовать, либо если вам нужен более простой вариант, пропустите инструкции в этом разделе и просто пересоздайте ведомые серверы сразу после завершения pg_upgrade и запуска нового ведущего сервера.

    1. **Установите новые исполняемые файлы Postgres Pro на ведомых серверах**

       Убедитесь в том, что на всех ведомых серверах установлены новые исполняемые и вспомогательные файлы.

    2. **Убедитесь в том, что новые каталоги данных на ведомых серверах *не* существуют**

       Новые каталоги данных ведомых серверов должны *отсутствовать* либо быть пустыми. Если запускалась программа initdb, удалите новые каталоги данных на ведомых.

3. **Установить разделяемые объектные файлы расширения**

   Установите на новых ведомых серверах те же разделяемые объектные файлы расширения, что вы установили в новом ведущем кластере.

4. **Остановите ведомые серверы**

   Если ведомые серверы продолжают работу, остановите их, следуя приведённым выше инструкциям.

5. **Сохраните файлы конфигурации**

   Сохраните все нужные вам файлы конфигурации из старых каталогов конфигурации ведомых серверов, в частности `postgresql.conf` (и все файлы, включённые в него), `postgresql.auto.conf` и `pg_hba.conf`, так как они будут перезаписаны или удалены на следующем этапе.

6. **Запустите rsync**

   Когда используется режим ссылок, ведомые серверы можно быстро обновить, применив rsync. Для этого в каталоге, внутри которого находятся каталоги старого и нового кластера, для каждого ведомого сервера выполните на *ведущем*:

   ```
   rsync --archive --delete --hard-links --size-only --no-inc-recursive old_cluster
    new_cluster remote_dir
   ```

   Здесь каталоги `old_cluster` и `new_cluster` задаются относительно текущего каталога на ведущем, а `remote_dir` находится *над* каталогами старого и нового кластера на ведомом. Структура подкаталогов в заданных каталогах на ведущем и ведомых серверах должна быть одинаковой. Обратитесь к странице руководства rsync, где подробно описано, как указать удалённый каталог, например так:

   ```
   rsync --archive --delete --hard-links --size-only --no-inc-recursive /opt/
   PostgreSQL/12 \
         /opt/PostgreSQL/17 standby.example.com:/opt/PostgreSQL
   ```

   Проверить, что будет делать команда, можно, воспользовавшись параметром rsync `--dry-run`. Выполнить rsync на ведущем необходимо как минимум с одним ведомым, но затем, пока обновлённый ведомый остаётся остановленным, можно запускать rsync на нём для обновления других ведомых.

   В ходе этой операции записываются ссылки, созданные режимом ссылок pg_upgrade, связывающие файлы нового и старого кластера на ведущем сервере. Затем в старом кластере ведомого находятся соответствующие файлы и в новом кластере ведомого создаются ссылки на них. Файлы, не связанные ссылками на ведущем, копируются с него на ведомый. (Обычно их объём невелик.) Это позволяет произвести обновление ведомого быстро. К сожалению, при этом rsync будет напрасно копировать файлы, связанные с временными и нежурналируемыми таблицами, так как они обычно не будут существовать на ведомых серверах.

   Если у вас есть табличные пространства, вам потребуется выполнить подобную команду rsync для каталогов всех табличных пространств, например:

   ```
   rsync --archive --delete --hard-links --size-only --no-inc-recursive /vol1/
   pg_tblsp/PG_12_201909212 \
         /vol1/pg_tblsp/PG_17_202307071 standby.example.com:/vol1/pg_tblsp
   ```

   Если вы вынесли `pg_wal` за пределы каталогов данных, нужно будет запустить rsync и для этих каталогов.

7. **Настройте ведомые серверы с потоковой репликацией и трансляцией журнала**

   Настройте серверы для трансляции журнала. (Запускать `pg_backup_start()` и `pg_backup_stop()` или делать копию файловой системы не нужно, так как ведомые серверы остаются синхронизированными с ведущим.) Если версия старого ведущего ниже 17.0, слоты не будут копироваться на новый ведомый, и все слоты нужно будет воссоздавать вручную. Если старый ведущий сервер версии 17.0 или выше, логические слоты на ведущем сервере будут скопированы на новый ведомый сервер, но слоты на старом ведомом сервере копироваться не будут, и их нужно будет воссоздавать вручную.

13. **Восстановить `pg_hba.conf`**

    Если вы изменяли `pg_hba.conf`, восстановите его исходное состояние. Также может потребоваться скорректировать другие файлы конфигурации в новом кластере, чтобы они соответствовали старому, например, `postgresql.conf` (и файлы, включённые в него) и `postgresql.auto.conf`.

14. **Запустить новый сервер**

    Теперь можно безопасно запустить новый сервер, а затем ведомые серверы, синхронизированные с ним с помощью rsync.

15. **Действия после обновления**

    Если после обновления требуются какие-то дополнительные действия, программа pg_upgrade выдаст предупреждения об этом по завершении работы. Она также сгенерирует файлы скриптов, которые должны запускаться администратором. Эти скрипты будут подключаться к каждой базе данных, требующей дополнительных операций. Каждый такой скрипт следует выполнять командой:

    ```
    psql --username=postgres --file=script.sql postgres
    ```

    Эти скрипты могут выполняться в любом порядке, а после выполнения их можно удалить.

    ### Внимание

    Обычно к таблицам, задействованным в перестраивающих базу скриптах, опасно обращаться, пока эти скрипты не сделают свою работу; при этом можно получить некорректный результат или плохую производительность. К таблицам, не задействованным в таких скриптах, можно обращаться немедленно.

16. **Статистика**

    Так как статистика оптимизатора не передаётся в процессе работы `pg_upgrade`, вы получите указание запустить соответствующую команду для воссоздания этой информации после обновления. Возможно, для этого вам понадобится установить параметры подключения к новому кластеру.

    Команда `vacuumdb --all --analyze-only` эффективно генерирует такую статистику, а параметр `--jobs` поможет ускорить процесс. Параметр `--analyze-in-stages` можно использовать для быстрого получения минимального объёма статистики. Если значение `vacuum_cost_delay` не равно нулю, его можно переопределить, чтобы ускорить генерацию статистики с помощью `PGOPTIONS`, например `PGOPTIONS='-c vacuum_cost_delay=0' vacuumdb ...`.

17. **Удалить старый кластер**

    Если вы удовлетворены результатами обновления, вы можете удалить каталоги данных старого кластера, запустив скрипт, упомянутый в выводе `pg_upgrade` после обновления. (Автоматическое удаление невозможно, если в старом каталоге данных находятся дополнительные табличные пространства.) Также вы можете удалить каталоги старой инсталляции (например, `bin`, `share`).

18. **Возврат к старому кластеру**

    Если выполнив команду `pg_upgrade`, вы захотите вернуться к старому кластеру, возможны следующие варианты:

    - Если использовался ключ `--check`, в старом кластере ничего не меняется; его можно просто перезапустить.

- Если *не* использовался ключ `--link`, в старом кластере ничего не меняется; его можно просто перезапустить.
- Если использовался ключ `--link`, у старого и нового кластера могут оказаться общие файлы данных:
  - Если работа `pg_upgrade` была прервана до начала расстановки ссылок, в старом кластере ничего не меняется; его можно просто перезапустить.
  - Если вы *не* запускали новый кластер, старый кластер не претерпел никаких изменений, за исключением того, что при создании ссылки на данные к имени `$PGDATA/global/pg_control` было добавлено окончание `.old`. Чтобы продолжить использование старого кластера, достаточно убрать окончание `.old` из имени файла `$PGDATA/global/pg_control`; после этого старый кластер можно будет перезапустить.
  - Если вы запускали новый кластер, он внёс изменения в общие файлы, и использовать старый кластер небезопасно. В этом случае старый кластер нужно будет восстановить из резервной копии.

Переменные окружения

Некоторые переменные окружения могут передавать значения по умолчанию для параметров командной строки:

`PGBINOLD`

Каталог с исполняемыми файлами старой версии Postgres Pro; параметр `-b`/`--old-bindir`.

`PGBINNEW`

Каталог с исполняемыми файлами новой версии Postgres Pro; параметр `-B`/`--new-bindir`.

`PGDATAOLD`

Каталог конфигурации старого кластера; параметр `-d`/`--old-datadir`.

`PGDATANEW`

Каталог конфигурации нового кластера; параметр `-D`/`--new-datadir`.

`PGPORTOLD`

Номер порта старого кластера; параметр `-p`/`--old-port`.

`PGPORTNEW`

Номер порта нового кластера; параметр `-P`/`--new-port`.

`PGSOCKETDIR`

Каталог, в котором будет создавать сокеты postmaster во время обновления; параметр `-s`/`--socketdir`.

`PGUSER`

Имя пользователя для установки кластера; параметр `-U`/`--username`.

Примечания

Программа pg_upgrade создаёт различные временные файлы, например выгружая схему базы, в своём рабочем каталоге `pg_upgrade_output.d` внутри каталога данных нового кластера. При каждом запуске в этом рабочем каталоге создаётся подкаталог с именем, содержащим отметку времени в формате стандарта ISO 8601 (`%Y%m%dT%H%M%S`), и в данный подкаталог помещаются все создаваемые файлы. Каталог `pg_upgrade_output.d` со всем содержимым удаляется автоматически в случае успешного выполнения pg_upgrade; но если возникает проблема, сохранённые в нём файлы могут быть полезны для диагностики.

Программа pg_upgrade запускает на короткое время процессы postmaster со старым и новым каталогом данных. Временные файлы сокетов Unix для взаимодействия с этими процессами по умолчанию создаются в текущем рабочем каталоге. В некоторых ситуациях путь к файлу в текущем каталоге может оказаться слишком длинным для имени сокета. В этом случае вы можете передать параметр `-s`, чтобы файлы сокетов создавались в другом каталоге с более коротким путём. В целях безопасности этот каталог не должен быть доступен для чтения и записи другим пользователям. (В Windows это не поддерживается.)

Программа pg_upgrade сообщит обо всех актуальных для вашей инсталляции ошибках и потребностях перестроения или переиндексации базы; при этом будут созданы завершающие обновление скрипты, перестраивающие таблицы или индексы. Если вы попытаетесь автоматизировать обновление множества серверов, вы обнаружите, что для кластеров с одинаковыми схемами баз данных потребуются одинаковые действия после обновления; это объясняется тем, что эти действия диктуются схемой базы данных, а не данными пользователей.

Для проверки развёртывания новой версии создайте копию только схемы старого кластера, наполните этот кластер фиктивными данными, и попробуйте обновить его.

pg_upgrade не поддерживает обновление баз данных, в которых есть таблицы со столбцами, имеющими следующие системные типы данных `reg*`, ссылающиеся на OID:

```
regcollation
regconfig
regdictionary
regnamespace
regoper
regoperator
regproc
regprocedure
regprofile
```

(Обновление `regclass`, `regrole` и `regtype` поддерживается.)

При обновлении экземпляра Postgres Pro версии 9.6 или старее для баз данных с многобайтовой кодировкой утилита pg_upgrade может сгенерировать файлы SQL с командами `REINDEX`/`VALIDATE`. Вы должны запустить эти файлы, чтобы перестроить индексы или перепроверить ограничения. Такие файлы могут появиться, когда в старом кластере используются индексы или ограничения, которые зависят от правил сортировки, отличных от основного правила сортировки БД, `C` или `POSIX`.

Если вы хотите использовать режим ссылок на данные, но при этом исключить изменения в старом кластере при запуске нового, вам может подойти режим клонирования. Если же этот режим недоступен, сделайте копию старого кластера и обновите его в этом режиме. Чтобы получить рабочую копию старого кластера, воспользуйтесь командой `rsync` и создайте предварительную копию кластера при работающем сервере, а затем отключите старый сервер и ещё раз запустите `rsync --checksum`, чтобы привести эту копию в согласованное состояние. (Ключ `--checksum` необходим, потому что `rsync` различает время с точностью только до секунд.) При этом вы можете исключить некоторые файлы, например `postmaster.pid`, как описано в Подразделе 24.3.4. Если ваша файловая система поддерживает снимки файловой системы или копирование при записи, вы можете воспользоваться этим для создания копии старого кластера и табличных пространств; при этом важно, чтобы такие снимки и копии файлов создавались одномоментно или когда сервер баз данных отключён.

См. также

initdb, pg_ctl, pg_dump, postgres

pg_waldump

pg_waldump — вывести журнал предзаписи кластера БД Postgres Pro в понятном человеку виде

Синтаксис

`pg_waldump` [`параметр`...] [`начальный_сегмент` [`конечный_сегмент`]]

Описание

Программа `pg_waldump` показывает содержимое журнала предзаписи (WAL) и прежде всего полезна для целей отладки и обучения.

Эту утилиту может запускать только пользователь, установивший сервер, так как ей требуется доступ на чтение к каталогу данных.

Параметры

Следующие аргументы командной строки задают расположение данных и формат вывода:

`начальный_сегмент`

Начать чтение с указанного файла сегмента WAL. Это неявно определяет каталог, в котором будут находиться файлы, и целевую линию времени.

`конечный_сегмент`

Остановиться после чтения указанного файла сегмента WAL.

`-b`
`--bkp-details`

Выводить подробные сведения о блоках-копиях страниц.

`-B` *`блок`*
`--block=`*`блок`*

Выводить только записи, изменяющие указанный блок. При этом также необходимо указать отношение в аргументе `--relation` или `-R`.

`-e` *`конец`*
`--end=`*`конец`*

Прекратить чтение в заданной позиции в WAL, а не читать поток до конца.

`-f`
`--follow`

Достигнув конца корректного WAL, проверять раз в секунду поступление новых записей WAL.

`-F` *`слой`*
`--fork=`*`слой`*

Выводить только записи, изменяющие блоки в указанном слое. Допустимые значения: `main` (основной слой), `fsm` (карта свободного пространства), `vm` (карта видимости) и `init` (слой инициализации).

`-n` *`предел`*
`--limit=`*`предел`*

Вывести заданное число записей и остановиться.

`-p` *`путь`*
`--path=`*`путь`*

Задаёт каталог, содержащий файлы сегментов WAL, либо каталог с подкаталогом `pg_wal`, содержащим такие файлы. По умолчанию в поисках этих файлов просматривается текущий каталог, подкаталог `pg_wal` текущего каталога и подкаталог `pg_wal` каталога `PGDATA`.

`-q`
`--quiet`

Не выводить ничего кроме ошибок. Этот ключ может быть полезен, когда вы хотите узнать, можно ли полностью разобрать диапазон записей WAL, но собственно содержимое записей вас не интересует.

`-r` *`менеджер_ресурсов`*
`--rmgr=`*`менеджер_ресурсов`*

Выводить только записи, созданные указанным менеджером ресурсов. Вы можете указать параметр несколько раз, чтобы выбрать несколько менеджеров ресурсов. Когда в качестве имени менеджера передаётся `list`, программа выводит только список возможных имён менеджеров ресурсов и завершается.

Расширения могут определять пользовательские менеджеры ресурсов, но pg_waldump не загружает модули расширений и поэтому не может распознать пользовательские менеджеры ресурсов по имени. Вместо этого можно обозначить пользовательского менеджера ресурсов в форме `custom###`, где *###* — трёхзначный идентификатор менеджера ресурсов. Обозначения в этой форме всегда будут считаться действительными.

`-R` *`табл_пространство/бд/отношение`*
`--relation=`*`табл_пространство/бд/отношение`*

Выводить только записи, изменяющие блоки в указанном отношении. Отношение указывается с OID табличного пространства, OID базы данных и номером файла, разделёнными косой чертой, например `1234/12345/12345`. В этом же формате отношения обозначаются в выводе программы.

`-s` *`начало`*
`--start=`*`начало`*

Позиция в WAL, с которой нужно начать чтение. По умолчанию чтение начинается с первой корректной записи WAL в самом первом из найденных файлов.

`-t` *`линия_времени`*
`--timeline=`*`линия_времени`*

Линия времени, из которой будут читаться записи WAL. По умолчанию используется значение, заданное параметром *`начальный_сегмент`*, если он присутствует, а иначе — 1. Значение можно задавать в десятичном или шестнадцатеричном представлении, например `17` или `0x11`.

`-V`
`--version`

Вывести версию pg_waldump и завершиться.

`-w`
`--fullpage`

Выводить только записи, содержащие образы полных страниц.

`-x` *`xid`*
`--xid=`*`xid`*

Вывести только записи, относящиеся к указанной транзакции.

`-z`
`--stats[=record]`

Вывести общую статистику (число и размер записей и образов полных страниц) вместо отдельных записей. Возможен вариант получения статистики по записям, а не по менеджерам ресурсов.

Если программа pg_waldump прерывается сигналом SIGINT (**Control**+**C**), она выдаёт сводку по данным, которые были обработаны к моменту прерывания. Эта операция не поддерживается в Windows.

`--save-fullpage=`*`путь_хранения`*

Сохранять образы полных страниц, записываемые в WAL, в каталог, указанный в *`пути_хранения`*. Фильтровать сохраняемые образы и ограничивать их вывод можно так же, как и для записей.

Образы полных страниц сохраняются с именами в таком формате: *`TIMELINE-LSN.RELTABLESPACE.DATOID.RELNODE.BLKNO_FORK`*. Имена файлов состоят из следующих компонентов:

Компонент	**Описание**
TIMELINE	линия времени файла-сегмента WAL, в котором хранится запись, состоит из одного шестнадцатеричного числа шириной 8 символов `%08X`
LSN	LSN записи с этим образом, состоит из двух шестнадцатеричных чисел шириной 8 символов `%08X-%08X`
RELTABLESPACE	OID табличного пространства, содержащего блок
DATOID	OID базы данных, содержащей блок
RELNODE	номер файлового узла для блока
BLKNO	номер блока
FORK	имя слоя, в котором был создан образ полной страницы: `main`, `fsm`, `vm` или `init`

`-?`
`--help`

Вывести справку об аргументах командной строки pg_waldump и завершиться.

Переменные окружения

`PGDATA`

Каталог данных; также см. параметр `-p`.

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Когда сервер работает, результаты могут быть некорректными.

Выводятся записи только указанной линии времени (или линии времени по умолчанию, если она не задана явно). Записи в других линиях времени игнорируются.

pg_waldump не будет читать файлы WAL с расширением `.partial`. Если требуется прочитать такие файлы, расширение `.partial` нужно убрать из их имён.

См. также

Раздел 27.6

pg_walsummary

pg_walsummary — вывести содержимое файлов сводок WAL

Синтаксис

`pg_walsummary` [*параметр*...] [*file*...]

Описание

pg_walsummary используется для вывода содержимого файлов сводок WAL. Эти двоичные файлы находятся в подкаталоге `pg_wal/summaries` каталога данных и могут быть преобразованы в текст с помощью этого инструмента. Обычно это не требуется, поскольку файлы сводок WAL в первую очередь необходимы для поддержки инкрементального резервного копирования, но они могут быть также полезны для отладки.

Файл сводки WAL индексируется по OID табличного пространства, а также по OID и слою отношения. Для каждого слоя отношения он хранит список блоков, изменённых WAL в диапазоне, для которого была создана сводка в файле. В нём также может храниться «предельный блок», который равен 0, если слой отношения был создан или усечён в соответствующем диапазоне WAL, а в противном случае — наименьшей длине, до которой был усечён слой отношения. Если слой отношения не был создан, удалён или усечён в соответствующем диапазоне WAL, предельный блок не будет выводится этим инструментом, поскольку либо не определён, либо равен бесконечности.

Параметры

`-i`
`--individual`

По умолчанию `pg_walsummary` выводит одну строку для каждого диапазона из одного или нескольких последовательных изменённых блоков. Так вывод становится намного короче, поскольку отношение, в котором все блоки от 0 до 999 были изменены, создаст только одну строку вывода, а не 1 000 отдельных строк. При указании этого параметра для каждого изменённого блока выводится отдельная строка.

`-q`
`--quiet`

Не выводить никаких данных, кроме сообщений об ошибках. Это может быть полезно, чтобы узнать, можно ли разобрать файл сводки WAL вне зависимости от его содержимого.

`-V`
`--version`

Показать версию, а затем завершиться.

`-?`
`--help`

Показать справку об аргументах командной строки pg_walsummary и завершиться.

Переменные окружения

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

См. также

pg_basebackup, pg_combinebackup

postgres

postgres — Сервер баз данных Postgres Pro

Синтаксис

`postgres` [*параметр*...]

Описание

`postgres` это сервер баз данных Postgres Pro. Для получения доступа к базе данных клиент устанавливает соединение (локально или по сети) с сервером `postgres`. После установки соединения сервер `postgres` поднимает выделенный процесс для его обслуживания.

Один экземпляр `postgres` всегда управляет данными ровно одного кластера баз данных. Кластер — это коллекция баз данных, хранящихся в файловой системе в определённом размещении («области данных»). На одном физическом сервере можно запустить несколько экземпляров `postgres` одновременно, при условии, что они используют различные области данных и порты. При запуске `postgres` необходимо указать размещение данных, которое задаётся в параметре `-D` или переменной окружения `PGDATA`, значение по умолчанию отсутствует. Обычно `-D` или `PGDATA` указывает на каталог, созданный во время развёртывания кластера с помощью initdb. Иные варианты рассмотрены в Разделе 18.2.

По умолчанию `postgres` запускается не в фоновом режиме, а вывод журнала осуществляет в стандартный поток ошибок. На практике `postgres` должен запускаться в фоновом режиме, возможно, при старте системы.

Команду `postgres` также возможно использовать в однопользовательском режиме. В основном этот режим используется на этапе инициализации при выполнении initdb. Иногда он также применяется в целях отладки или после аварийного сбоя. Заметьте, что однопользовательский режим не вполне подходит для отладки сервера ввиду отсутствия в нём межпроцессного взаимодействия и блокировок. Когда сервер запускается в однопользовательском режиме из командной строки, он может принимать запросы и выводить их результаты на экран, но формат этого вывода ориентирован больше на разработчиков, чем на обычных пользователей. В этом режиме текущим пользователем считается пользователь под номером 1, который неявно наделяется правами суперпользователя. При этом данный пользователь может фактически не существовать, поэтому в ряде случаев этот режим даёт возможность вручную восстановить базу при повреждении системных каталогов.

Параметры

Программа `postgres` принимает следующие аргументы командной строки. За подробным описанием параметров обратитесь к Главе 18. От необходимости вводить большинство этих параметров можно избавиться, записав их в файл конфигурации. Некоторые (безопасные) параметры можно также задать со стороны подключающегося клиента (в зависимости от приложения), чтобы они применялись только к одному сеансу. Например, если установлена переменная окружения `PGOPTIONS`, клиенты на базе libpq передадут эту строку серверу, который воспримет её как параметры, передаваемые в командной строке `postgres`.

Параметры общего назначения

`-B` *`количество-буферов`*

Устанавливает количество разделяемых между процессами буферов. Значение по умолчанию выбирается автоматически при развёртывании кластера с помощью initdb. Установка флага аналогична конфигурации параметра shared_buffers.

`-c` *`имя=значение`*

Устанавливает заданный параметр времени выполнения. Конфигурационные параметры, поддерживаемые Postgres Pro, описаны в Главе 18. Большинство других параметров командной строки на самом деле представляют собой краткие формы такого присваивания значений параметрам. Для установления нескольких параметров `-c` можно указывать многократно.

`-C` *имя*

Отображает значение заданного параметра времени выполнения и завершается. (См. также описание параметра `-c` выше.) При этом выдаются значения из `postgresql.conf` с учётом возможных переопределений параметров при вызове. Однако параметры, установленные при запущенном сервере, в выводе не отражаются.

Для большинства параметров можно запросить значения при работающем сервере. Однако для некоторых параметров, вычисляемых во время выполнения, сервер должен быть отключён (например, shared_memory_size, shared_memory_size_in_huge_pages и wal_segment_size).

Параметр предназначен для приложений, взаимодействующих с сервером, например pg_ctl, и запрашивающих параметры конфигурации. Пользовательские приложения должны использовать команду `SHOW` или представление `pg_settings`.

`-d` *уровень-отладки*

Устанавливает уровень отладки (от 1 до 5). Чем выше значение, тем подробнее осуществляется вывод в журнал сервера. Также возможно передать `-d 0` для отдельного сеанса, что предотвратит в его рамках влияние выставленного для `postgres` значения.

`-D` *каталог_данных*

Указывает размещение конфигурационных файлов базы в пределах файловой системы. За подробностями обратитесь к Разделу 18.2.

`-e`

Устанавливает формат вводимых дат по умолчанию в «European» с последовательностью значений `DMY`. Также влияет на вывод дня, идущего перед значением месяца, более подробно см. Раздел 8.5.

`-F`

Отключает вызовы `fsync` для увеличения производительности, но с увеличением рисков потери данных в случае краха системы. Этот параметр работает аналогично параметру конфигурации fsync. Внимательно прочтите документацию перед использованием данного параметра!

`-h` *компьютер*

Указывает IP-адрес или имя компьютера, на котором сервер `postgres` принимает клиентские подключения по TCP/IP. Значением может быть список адресов, разделённых запятыми, либо символ `*`, обозначающий все доступные интерфейсы. Если значение опущено, то подключения принимаются только через Unix-сокеты. По умолчанию принимаются подключения только к localhost. Флаг работает аналогично конфигурационному параметру listen_addresses.

`-i`

Позволяет клиентам подключаться по TCP/IP. Без этого параметра допускаются лишь локальные подключения. Действие этого параметра аналогично действию параметра конфигурации `listen_addresses` со значением `*` в `postgresql.conf` или ключа `-h`.

Параметр устарел, так как не даёт полной функциональности listen_addresses. Лучше устанавливать значение `listen_addresses` напрямую.

`-k` *каталог*

Указывает каталог Unix-сокета, через который `postgres` будет принимать подключения. Значением параметра может быть список каталогов через запятую. Если это значение пустое, использование Unix-сокетов запрещается, разрешаются только подключения по TCP/IP. По умолчанию выбирается каталог `/tmp`, но его можно сменить на этапе компиляции. Этот параметр действует аналогично параметру конфигурации unix_socket_directories.

`-l`

Включает поддержку безопасных соединений с использованием SSL шифрования. Postgres Pro необходимо скомпилировать с поддержкой SSL для использования этого флага. Подробнее использование SSL описано в Раздел 17.9.

`-N` *`максимальное количество соединений`*

Устанавливает максимально возможное количество одновременных клиентских соединений. Значение по умолчанию устанавливается автоматически на этапе развёртывания с помощью initdb. Флаг работает аналогично конфигурационному параметру max_connections.

`-p` *`порт`*

Указывает порт TCP/IP или расширение файла локального Unix-сокета, через который `postgres` принимает подключения клиентских приложений. По умолчанию принимает значение переменной окружения `PGPORT`, или, если значение `PGPORT` не установлено, то используется значение, установленное на этапе компиляции (обычно это 5432). Если значение порта меняется, то на стороне клиентов это необходимо учитывать, установив, либо `PGPORT`, либо флаг командной строки.

`-s`

Отображает информацию о времени и другую статистику после каждой выполненной команды, что полезно для оценки производительности во время настройки количества буферов.

`-S` *`рабочая-память`*

Указывает базовый объём памяти, который сервер будет использовать для сортировок и хеш-таблиц, прежде чем прибегнуть к использованию временных файлов на диске. Обратитесь к описанию параметра `work_mem`, приведённому в Подразделе 18.4.1.

`-V`
`--version`

Отображает версию postgres и прерывает дальнейшее выполнение.

`--`*`имя`*`=`*`значение`*

Устанавливает заданный параметр времени выполнения. Является короткой формой ключа `-c`.

`--describe-config`

Выводит значения конфигурационных переменных сервера, их описаний и значений по умолчанию в формате команды `COPY` со знаком табуляции в качестве разделителя. В основном это предназначено для средств администрирования.

`-?`
`--help`

Выводит помощь по аргументам команды postgres.

Параметры для внутреннего использования

Далее описанные параметры, в основном, применяются в целях отладки, а в некоторых случаях при восстановлении сильно повреждённых баз данных. Их описание приведено для системных разработчиков Postgres Pro, поэтому они могут быть изменены без уведомления.

`-f { s | i | o | b | t | n | m | h }`

Запрещает использование специфических методов сканирования и объединения: `s` и `i` выключают последовательное сканирование и по индексу соответственно, а `o`, `b` и `t` выключает сканирование только по индексу, сканирование по битовым векторам, и сканирование по ID кортежей соответственно, в то время как `n`, `m` и `h` выключает вложенные циклы, слияния и хеширование соответственно.

Ни последовательное сканирование, ни вложенные циклы невозможно выключить полностью. Флаги `-fs` и `-fn` просто указывают планировщику избегать выполнения этих операций при наличии других альтернатив.

`-O`

Разрешает модифицировать структуру системных таблиц. Используется командой `initdb`.

`-P`

Игнорировать системные индексы при чтении, но продолжать обновлять их при изменениях системных таблиц. Это используется при их восстановлении после повреждения.

`-t pa[rser] | pl[anner] | e[xecutor]`

Выводит статистику по времени исполнения каждого запроса в контексте каждого системного модуля. Использование флага совместно с `-s` невозможно.

`-T`

Параметр предназначен для отладки сервера в случае аномального завершения процесса. Обычная практика в таком случае — завершение порождённых процессов путём отправки сигналов SIGQUIT. С этим параметром будет отправляться сигнал SIGABRT, а в результате будут создаваться файлы дампа памяти.

`-v` *протокол*

Указывает для данного сеанса версию протокола взаимодействия сервера с клиентом. Флаг используется лишь для внутренних целей.

`-W` *секунды*

При старте сервера производится задержка на указанное количество секунд, после чего производится процедура аутентификации, что позволяет подключить отладчик к процессу.

`-Z`

Проверяет, совместим ли текущий исполняемый файл postgres с определённым кластером. Если тип архитектуры или другие параметры компиляции, влияющие на совместимость кластера, не совпадают, программа возвращает код завершения 1 и выдаёт сообщение об ошибке, описывающее первую обнаруженную несовместимость. Иначе она завершается с кодом успешного результата, при этом не запуская кластер.

Для проведения этой проверки необходимо передать путь к каталогу данных целевого кластера.

Эта проверка будет полноценной, только если кластер и исполняемый файл postgres рассчитаны на одну архитектуру процессора и одинаковый порядок байтов.

Подсказка

В случае выявления несовместимости запустите pgpro_controldata с ключом `-P` или `-C`, чтобы получить список всех параметров, от которых зависит совместимость.

Параметры для однопользовательского режима

Следующие параметры применимы только для однопользовательского режима (см. раздел Однопользовательский режим ниже).

`--single`

Устанавливает однопользовательский режим. Параметр должен идти первым в командной строке.

`база_данных`

Указывает имя базы данных, к которой производится подключение. Параметр должен идти последним в командной строке. Если не указан, то используется имя текущего системного пользователя.

`-E`

Выводить все команды в устройство стандартного вывода прежде чем выполнять их.

`-j`

Считать признаком окончания ввода команды точку с запятой с двумя переводами строки, а не один перевод строки.

`-r` *`имя_файла`*

Отправляет вывод журнала сервера в файл *`filename`*. Этот параметр применяется лишь при запуске из командной строки.

Переменные окружения

`PGCLIENTENCODING`

Кодировка, используемая клиентом по умолчанию. Может переопределяться на стороне клиента, а также устанавливаться в конфигурационном файле сервера.

`PGDATA`

Каталог размещения данных кластера по умолчанию

`PGDATESTYLE`

Значение по умолчанию для параметра времени выполнения DateStyle. Применение этой переменной является устаревшим.

`PGPORT`

Порт по умолчанию, лучше устанавливать в конфигурационном файле.

Диагностика

Ошибки с упоминанием о `semget` или `shmget` говорят о возможной необходимости проведения более оптимального конфигурирования ядра. Подробнее это обсуждается в Разделе 17.4. Отложить переконфигурирование можно, уменьшив shared_buffers для снижения общего потребления разделяемой памяти Postgres Pro и/или уменьшив max_connections для снижения затрат на использование семафоров.

Необходимо внимательно проверять сообщения об ошибке с упоминанием о другом запущенном экземпляре, например, с использованием команды

```
$ ps ax | grep postgres
```

или

```
$ ps -ef | grep postgres
```

в зависимости от ОС. Если есть полная уверенность, что противоречий нет, необходимо самостоятельно удалить упомянутый в сообщении запирающий файл и повторить попытку.

Упоминание о невозможности привязки к порту в сообщениях об ошибках может указывать на то, что он уже занят другим процессом помимо Postgres Pro. Также сообщение может возникнуть при мгновенном рестарте `postgres` на том же порту. В этом случае нужно немного подождать, пока ОС не закроет порт, и повторить попытку. Ещё возможна ситуация, в которой используется резервный системный порт. Например, многие Unix-подобные ОС резервируют «доверительные» порты от 1024 и ниже, и лишь суперпользователь имеет к ним доступ.

Примечания

Для комфортного запуска и остановки сервера можно использовать утилиту pg_ctl.

Если возможно, *не используйте* сигнал `SIGKILL` для головного процесса `postgres`. В этом случае `postgres` не освободит системные ресурсы, например, разделяемую память и семафоры. Это может привести к проблемам при повторном запуске `postgres`.

Для корректного завершения `postgres` используются сигналы `SIGTERM`, `SIGINT` или `SIGQUIT`. При первом будут ожидаться все дочерние процессы до их завершения, второй приведёт к принудительному закрытию соединений, а третий — к незамедлительному выходу без корректного завершения, приводящему к необходимости выполнения процедуры восстановления на следующем старте.

Получая сигнал `SIGHUP`, сервер перечитывает свои файлы конфигурации. Также возможно отправить `SIGHUP` отдельному процессу, но это чаще всего бессмысленно.

Для отмены исполняющегося запроса, отправьте `SIGINT` обслуживающему его процессу. Для чистого завершения серверного процесса отправьте ему `SIGTERM`. Также см. `pg_cancel_backend` и `pg_terminate_backend` в Подразделе 9.28.2, которые являются аналогами в форме SQL-инструкций.

Сервер `postgres` обрабатывает `SIGQUIT` для завершения дочерних процессов в грязную, и сигнал *не должен* отправляться пользователем. Также не стоит посылать `SIGKILL` серверному процессу — головной `postgres` процесс расценит это как аварию и принудительно завершит остальные порождённые, как это было бы сделано при процедуре восстановления после сбоя.

Ошибки

Флаги, начинающиеся с `--` не работают в ОС FreeBSD или OpenBSD. Чтобы обойти это, используйте `-c`. Это ошибка ОС. В будущих релизах Postgres Pro будет предоставлен обходной путь, если ошибка так и не будет устранена.

Однопользовательский режим

Для запуска сервера в однопользовательском режиме используется, например, команда

```
postgres --single -D /usr/local/pgsql/data другие параметры my_database
```

Необходимо указать корректный путь к каталогу хранения данных в параметре `-D`, или установить переменную окружения `PGDATA`. Также замените имя базы данных на необходимое.

Обычно перевод строки в однопользовательском режиме сервер воспринимает как завершение ввода команды; точка с запятой не имеет для него такого значения, как для psql. Поэтому, чтобы ввести команду, занимающую несколько строк, необходимо добавлять в конце каждой строки, кроме последней, обратную косую черту. Обратная косая черта и следующий за ней символ перевода строки автоматически убираются из вводимой команды. Заметьте, что это происходит даже внутри строковой константы или комментария.

Но если применить аргумент командной строки `-j`, одиночный символ перевода строки не будет завершать ввод команды; это будет делать последовательность «точка с запятой, перевод строки, перевод строки». То есть для завершения команды нужно ввести точку с запятой, и сразу за ней пустую строку. Просто точка с запятой, дополненная переводом строки, в этом режиме не имеет специального значения. Внутри строковых констант и комментариев такая завершающая последовательность воспринимается в том же ключе.

Вне зависимости от режима ввода, символ точки с запятой, введённый не прямо перед или в составе последовательности завершения команды, воспринимается как разделитель команд. После ввода завершающей последовательности введённые с разделителями несколько операторов будут выполняться в одной транзакции.

Для завершения сеанса введите символ конца файла (EOF, обычно это сочетание **Control+D**). Если вы вводили текст после окончания ввода последней команды, символ EOF будет воспринят как символ завершения команды, и для выхода потребуется ещё один EOF.

Заметьте, что однопользовательский режим не предоставляет особых возможностей для редактирования команд (например, нет истории команд). Также в однопользовательском режиме не производятся никакие фоновые действия, например, не выполняются автоматические контрольные точки или репликация.

Примеры

Для запуска `postgres` в фоновом режиме с параметрами по умолчанию:

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

Для запуска `postgres` с определённым портом, например, 1234:

```
$ postgres -p 1234
```

Для соединения с помощью psql укажите этот порт в параметре -p:

```
$ psql -p 1234
```

или в переменной окружения `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Именованный параметр времени выполнения можно указать одним из приведённых способом:

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Любой из методов переопределяет значение `work_mem` конфигурации `postgresql.conf`. Символ подчёркивания в именах можно указать и в виде тире. Задавать параметры обычно (не считая кратковременных экспериментов) лучше в `postgresql.conf`, а не в аргументах командной строки.

См. также

initdb, pg_ctl

Часть VII. Внутреннее устройство

Содержит разнообразную информацию, полезную для разработчиков Postgres Pro.

Глава 50. Обзор внутреннего устройства Postgres Pro

Автор

Основой этой главы послужил материал дипломной работы sim98, написанной Стефаном Симковичем (Stefan Simkovics) в Венском техническом университете под руководством профессора Георга Готлоба (Georg Gottlob) и его ассистентки Катрин Сейр (Katrin Seyr).

В этой главе даётся обзор внутренней организации сервера Postgres Pro. Прочитав следующие разделы, вы получите представление о том, как обрабатывается запрос. Основная цель этой главы — помочь читателю понять общую последовательность действий, выполняемых сервером с момента получения запроса до момента выдачи результатов клиенту.

50.1. Путь запроса

Ниже мы кратко опишем этапы, которые проходит запрос для получения результата.

1. Прикладная программа устанавливает подключение к серверу Postgres Pro. Эта программа передаёт запрос на сервер и ждёт от него результатов.
2. На *этапе разбора запроса* сервер выполняет синтаксическую проверку запроса, переданного прикладной программой, и создаёт *дерево запроса*.
3. *Система правил* принимает дерево запроса, созданное на стадии разбора, и ищет в *системных каталогах правила* для применения к этому дереву. Обнаружив подходящие правила, она выполняет преобразования, заданные в *теле правил*.

   Одно из применений системы правил заключается в реализации *представлений*. Когда выполняется запрос к представлению (т. е. *виртуальной таблице*), система правил преобразует запрос пользователя в запрос, обращающийся не к представлению, а к *базовым таблицам* из *определения представления*.
4. *Планировщик/оптимизатор* принимает дерево запроса (возможно, переписанное) и создаёт *план запроса*, который будет передан *исполнителю*.

   Он выбирает план, сначала рассматривая все возможные варианты получения одного и того же результата. Например, если для обрабатываемого отношения создан индекс, прочитать отношение можно двумя способами. Во-первых, можно выполнить простое последовательное сканирование, а во-вторых, можно использовать индекс. Затем оценивается стоимость каждого варианта и выбирается самый дешёвый. Затем выбранный вариант разворачивается в полноценный план, который сможет использовать исполнитель.
5. Исполнитель рекурсивно проходит по *дереву плана* и получает строки тем способом, который указан в плане. Он сканирует отношения, обращаясь к *системе хранения*, выполняет *сортировку* и *соединения*, вычисляет *условия фильтра* и, наконец, возвращает полученные строки.

В следующих разделах мы более подробно рассмотрим каждый из этих этапов, чтобы дать представление о внутренних механизмах и структурах данных Postgres Pro.

50.2. Как устанавливаются соединения

Postgres Pro реализует простую клиент-серверную модель по схеме «процесс для пользователя». В такой схеме один *клиентский процесс* подключается к одному отдельному *серверному процессу*. Так как мы не знаем заранее, сколько подключений будет, нам нужен «главный процесс», который будет запускать новый процесс при каждом запросе подключения. Главный процесс называется *postmaster* и принимает входящие подключения в заданном порту TCP/IP. Получив запрос на подключение, процесс `postgres` порождает новый серверный процесс. Серверные задачи взаимодействуют между собой и другими процессами *экземпляра СУБД* через *семафоры* и *разделяемую память*, чтобы обеспечить целостность данных при одновременном обращении к ним.

Клиентским процессом может быть любая программа, которая понимает протокол Postgres Pro, описанный в Главе 53. Многие клиенты базируются на библиотеке libpq для языка C, но есть и другие независимые реализации этого протокола, например, драйвер JDBC для Java.

Установив подключение, клиентский процесс может передать запрос серверу. Запрос передаётся в обычном текстовом виде, клиент не занимается его анализом. Сервер разбирает запрос, строит *план выполнения*, выполняет его и возвращает полученные строки клиенту, передавая их через установленное подключение.

50.3. Этап разбора

Этап разбора разделяется на две части:

- *Разбор*, алгоритм которого описан в `gram.y` и `scan.l`, а программный код генерируется инструментами Unix bison и flex.
- *Преобразование*, в процессе которого модифицируются и дополняются структуры данных, полученные после разбора запроса.

50.3.1. Разбор

При разборе проверяется сначала синтаксис строки запроса (поступающей в виде неструктурированного текста). Если он правильный, строится *дерево запроса* и передаётся дальше, в противном случае возвращается ошибка. Лексический и синтаксический анализ реализован с применением хорошо известных средств Unix bison и flex.

Лексическая структура определяется в файле `scan.l` и описывает *идентификаторы*, *ключевые слова SQL* и т. д. Для каждого найденного ключевого слова или идентификатора генерируется *символ языка*, который затем передаётся синтаксическому анализатору.

Синтаксис языка определён в файле `gram.y` в виде набора *грамматических правил* и *действий*, которые должны выполняться при срабатывании правил. Для построения дерева разбора используется код действий (это действительно код на C).

Файл `scan.l` преобразуется в программу на C `scan.c` с помощью flex, а `gram.y` — в `gram.c` с помощью bison. После этих преобразований исполняемый код анализатора создаётся обычным компилятором C. Никогда не вносите коррективы в сгенерированные файлы C, так как они будут перезаписаны при следующем вызове flex или bison.

> **Примечание**
>
> Упомянутые преобразования и компиляция обычно производятся автоматически сборочными файлами *Makefile*, поставляемыми в составе дистрибутива Postgres Pro.

Подробное описание bison и грамматических правил в `gram.y` выходит за рамки данной главы. Узнать больше о flex и bison можно из книг и документации. Изучение грамматики, описанной в `gram.y`, следует начать со знакомства с bison, иначе будет трудно понять, что там происходит.

50.3.2. Преобразование

На этой стадии дерево разбора создаётся только с фиксированными знаниями о синтаксической структуре SQL. При его создании не просматриваются системные каталоги, что не даёт возможность понять конкретную семантику запрошенной операции. После этого выполняется *процедура преобразования*, которая принимает дерево разбора от анализатора и выполняет семантический анализ, необходимый для понимания, к каким именно таблицам, функциям и операторам обращается запрос. Структура данных, которая создаётся для представления этой информации, называется *деревом запроса*.

Синтаксический разбор отделён от семантического анализа, потому что обращаться к системным каталогам можно только внутри транзакции, а начинать транзакцию сразу после получения строки с запросом нежелательно. Синтаксического разбора достаточно, чтобы распознать команды управления транзакциями (`BEGIN`, `ROLLBACK` и т. д.), поэтому их можно выполнить без дальнейшего анализа. Убедившись, что мы имеем дело с собственно запросом (например, `SELECT` или `UPDATE`), можно начинать транзакцию, если она ещё не начата. Только после этого можно переходить к процедуре преобразования.

Дерево запроса, создаваемое процедурой преобразования, по структуре во многом похоже на дерево разбора, но отличается во многих деталях. Например, узел `FuncCall` в дереве разбора представляет то, что по синтаксису похоже на вызов функции. Этот узел может быть преобразован в узел `FuncExpr` или `Aggref` в зависимости от того, какой (обычной или агрегатной) окажется функция с заданным именем. Кроме того, в дерево запроса добавляется информация о фактических типах данных столбцов и результатов выражений.

50.4. Система правил Postgres Pro

Postgres Pro поддерживает мощную *систему правил* для создания *представлений* и возможности *изменения представлений*. Система правил Postgres Pro претерпела две реализации:

- Первый вариант производил обработку на *уровне строк* и был внедрён глубоко в *исполнителе*. Этот обработчик правил вызывался при обращении к каждой отдельной строке. Эта реализация была ликвидирована в 1995 г., когда последний официальный выпуск Berkeley Postgres превратился в Postgres95.
- Во втором воплощении системы правил применили так называемое *переписывание запроса*. *Система переписывания* реализована в механизме, внедрённом между *анализатором* и *планировщиком/оптимизатором*. Этот механизм работает и сегодня.

Механизм переписывания запросов подробно обсуждается в Главе 39, так что здесь мы его не рассматриваем. Мы только отметим, что и на входе, и на выходе у него деревья запросов, то есть представление или уровень семантической детализации он не меняет. Переписывание запроса можно считать формой расширения макросов.

50.5. Планировщик/оптимизатор

Задача *планировщика/оптимизатора* — построить наилучший план выполнения. Определённый SQL-запрос (а значит, и дерево запроса) на самом деле можно выполнить самыми разными способами, при этом получая одни и те же результаты. Если это не требует больших вычислений, оптимизатор запросов будет перебирать все возможные варианты планов, чтобы в итоге выбрать тот, который должен выполниться быстрее остальных.

Примечание

В некоторых ситуациях рассмотрение всех возможных вариантов выполнения запросов занимает слишком много времени и памяти. В частности, это имеет место при выполнении запросов с большим количеством операций соединения. Поэтому, чтобы выбрать разумный (но не обязательно наилучший) план запроса за приемлемое время, Postgres Pro использует *генетический оптимизатор запросов* (см. Главу 58), когда количество соединений превышает некоторый предел (см. geqo_threshold).

Процедура поиска лучшего плана на самом деле работает со структурами данных, называемыми *путями*, которые представляют собой упрощённые схемы планов, содержащие минимум информации, необходимый планировщику для принятия решений. Когда наиболее выгодный план выбран, строится полноценное *дерево плана*, которое и передаётся исполнителю. Оно описывает желаемый план выполнения достаточно подробно, чтобы исполнитель мог обработать его. В продолжении этого раздела мы будем считать, что планы и пути по сути одно и то же.

50.5.1. Выработка возможных планов

Сначала планировщик/оптимизатор вырабатывает планы для сканирования каждого отдельного отношения (таблицы), используемого в запросе. Множество возможных планов определяется в зависимости от наличия индексов в каждом отношении. Произвести последовательное сканирование отношения можно в любом случае, так что план последовательного сканирования создаётся всегда. Предположим, что для отношения создан индекс (например, индекс-B-дерево) и запрос содержит ограничение `отношение.атрибут ОПЕР константа`. Если окажется, что `отношение.атрибут` совпадает с ключом индекса-B-дерева и `ОПЕР` — один из операторов, входящих в *класс операторов* индекса, создаётся ещё один план, с использованием индекса-B-дерева для чтения отношения. Если находятся другие индексы, ключи которых соответствуют ограничениям запроса, могут добавиться и другие планы. Планы сканирования индекса также создаются для индексов, если их порядок сортировки соответствует предложению `ORDER BY` (если оно есть), или этот порядок может быть полезен для соединения слиянием (см. ниже).

Если в запросе требуется соединить два или несколько отношений, после того, как будут определены все подходящие планы сканирования отдельных отношений, рассматриваются планы соединения. При этом возможны три стратегии соединения:

- *соединение с вложенным циклом*: Правое отношение сканируется один раз для каждой строки, найденной в левом отношении. Эту стратегию легко реализовать, но она может быть очень трудоёмкой. (Однако если правое отношение можно сканировать по индексу, эта стратегия может быть удачной. Тогда значения из текущей строки левого отношения могут использоваться как ключи для сканирования по индексу справа.)
- *соединение слиянием*: Каждое отношение сортируется по атрибутам соединения до начала соединения. Затем два отношения сканируются параллельно и соответствующие строки, объединяясь, формируют строки соединения. Этот тип соединения более привлекательный, так как каждое отношение сканируется только один раз. Требуемый порядок сортировки можно получить, либо добавив явный этап сортировки, либо просканировав отношение в нужном порядке, используя индекс по ключу соединения.
- *соединение по хешу*: сначала сканируется правое отношение и формируется хеш-таблица, ключ в которой вычисляется по атрибутам соединения. Затем сканируется левое отношение и по тем же атрибутам в каждой строке вычисляется ключ для поиска в этой хеш-таблице соответствующих строк справа.

Когда в запросе задействованы более двух отношений, окончательный результат должен быть получен из дерева с узлами соединения, имеющими по два входа. Планировщик рассматривает все возможные последовательности соединения и выбирает самую выгодную.

Если число задействованных в запросе отношений меньше geqo_threshold, для поиска оптимальной последовательности соединений производится практически полный перебор. Планировщик отдаёт предпочтение соединениям между двумя отношениями, для которых есть соответствующее предложение соединения в условии `WHERE` (то есть, для которых находится ограничение вида `where табл1.атр1=табл2.атр2`). Пары соединения без подобного предложения рассматриваются, только если нет другого выбора, то есть когда для определённого отношения не находятся предложения соединения с каким-либо другим отношением. Планировщик рассматривает все возможные планы для каждой пары соединения и выбирает самый выгодный из них (по его оценке).

Если `geqo_threshold` превышается, последовательность соединений выбирается эвристическим путём, как описано в Главе 58. В остальном процесс планирования тот же.

Законченное дерево плана содержит узлы сканирования по индексу или последовательного сканирования базовых отношений, плюс узлы соединения с вложенным циклом, соединения слиянием или соединения по хешу (если требуется), плюс, возможно, узлы дополнительных действий, например, сортировки или вычисления агрегатных функций. Большинство из этих узлов могут дополнительно производить *отбор* (отбрасывать строки, не удовлетворяющие заданному логическому условию) и *расчёты* (вычислять производный набор столбцов по значениям заданных столбцов, то есть вычислять скалярные выражения). Одна из задач планировщика — добавить условия отбора из предложения `WHERE` и вычисления требуемых выходных выражений к наиболее подходящим узлам дерева плана.

50.6. Исполнитель

Исполнитель принимает план, созданный планировщиком/оптимизатором, и обрабатывает его рекурсивно, чтобы получить требуемый набор строк. Обработка выполняется по конвейеру, с получением данных по требованию. При вызове любого узла плана он должен выдать очередную строку, либо сообщить, что выдача строк завершена.

В качестве более конкретного примера, давайте предположим, что верхним узлом плана оказался узел `MergeJoin`. Для того чтобы выполнить какое-либо соединение, необходимо выбрать две строки (одну из каждого вложенного плана). Поэтому исполнитель рекурсивно вызывает себя для обработки вложенных планов (он начинает с плана `левого дерева`). Новый верхний узел (верхний узел левого вложенного плана) может быть, например, узлом `Sort`, и тогда для получения входной строки снова требуется рекурсия. Дочерним узлом `Sort` может быть узел `SeqScan`, представляющий собственно чтение таблицы. В результате выполнения этого узла исполнитель выбирает одну строку из таблицы и возвращает её вызывающему узлу. Узел `Sort`, в свою очередь, будет продолжать вызывать дочерний узел, пока не получит все строки для сортировки. Когда строки закончатся (дочерний узел сообщит об этом, возвратив NULL вместо строки), узел `Sort` выполнит сортировку, и наконец сможет выдать свою первую строку, а именно строку первую по порядку сортировки. Остальные строки будут сохраняться в нём, чтобы он мог выдавать их по порядку при последующих вызовах.

Узел `MergeJoin` подобным образом затребует первую строку и у вложенного плана справа. Затем он сравнивает две строки и определяет, можно ли их соединить; если да, он возвращает соединённую строки вызывающему узлу. При следующем вызове, или немедленно, если он не может соединить текущую пару поступивших строк, он переходит к следующей строке в одном отношении или в другом (в зависимости от результата сравнения) и снова проверяет соответствие. В конце концов, данные в одном или другом вложенном плане заканчиваются и узел `MergeJoin` возвращает NULL, показывая тем самым, что другие строки соединения получить нельзя.

Сложные запросы могут содержать много уровней вложенности узлов плана, но общий подход тот же: каждый узел вычисляет и возвращает следующую полученную строку при очередном вызове. Каждый узел также должен производить отбор и расчёты, которые были назначены ему планировщиком.

Механизм исполнителя применяется для обработки всех пяти основных типов SQL-запросов: `SELECT`, `INSERT`, `UPDATE`, `DELETE` и `MERGE`. С `SELECT` код исполнителя верхнего уровня должен только выдать клиенту все строки, полученные от дерева плана запроса. Запросы `INSERT ... SELECT`, `UPDATE`, `DELETE` и `MERGE` по сути выполняются как `SELECT` под специальным узлом `ModifyTable` на верхнем уровне плана.

Команда `INSERT ... SELECT` подаёт строки в узел `ModifyTable` для добавления в отношение. С `UPDATE` планировщик делает так, чтобы каждая вычисленная строка включала значения всех изменённых столбцов плюс *TID* (Tuple ID, идентификатор кортежа) исходной целевой строки; эти данные подаются в узел `ModifyTable`, который использует эту информацию, чтобы создать новую изменённую строку и пометить старую строку как удалённую. С `DELETE` план фактически возвращает только один столбец, TID, а узел `ModifyTable` использует значения TID, чтобы найти каждую целевую строку и пометить её как удалённую. С `MERGE` планировщик соединяет исходные и целевые отношения и включает все значения столбцов, необходимые для любого из предложений `WHEN`, плюс TID целевой строки; эти данные подаются в узел `ModifyTable`, который использует эту информацию, чтобы определить, какое предложение `WHEN` выполнять, а затем вставляет, изменяет или удаляет целевую строку по мере необходимости.

Простая команда `INSERT ... VALUES` создаёт тривиальное дерево плана, содержащее единственный узел `Result`, который вычисляет ровно одну строку результата и подаёт её в вышестоящий узел `ModifyTable` для добавления в отношение.

Глава 51. Системные каталоги

Системные каталоги — это место, где система управления реляционной базой данных хранит метаданные схемы, в частности информацию о таблицах и столбцах, а также служебные сведения. Системные каталоги Postgres Pro представляют собой обычные таблицы. Поэтому вы можете удалить и пересоздать их, добавить столбцы, изменить и добавить строки, т. е. разными способами вмешаться в работу системы. Обычно модифицировать системные каталоги вручную не следует, для всего этого, как правило, есть команды SQL. (Например, `CREATE DATABASE` вставляет строку в каталог `pg_database` — и фактически создаёт базу данных на диске.) Исключение составляют только особенные эзотерические операции, но многие из них со временем становятся выполнимыми посредством SQL-команд, так что потребность напрямую модифицировать системные каталоги постоянно уменьшается.

51.1. Обзор

В Таблице 51.1 перечислены системные каталоги. Подробное описание каждого каталога следует далее.

Большинство системных каталогов копируются из базы-шаблона при создании базы данных и затем принадлежат этой базе. Но некоторые каталоги физически разделяются всеми базами данных в кластере; это отмечено в их описаниях.

Таблица 51.1. Системные каталоги

Имя каталога	Предназначение
`pg_aggregate`	агрегатные функции
`pg_am`	методы доступа отношений
`pg_amop`	операторы методов доступа
`pg_amproc`	опорные функции методов доступа
`pg_attrdef`	значения столбцов по умолчанию
`pg_attribute`	столбцы таблиц («атрибуты»)
`pg_authid`	идентификаторы для авторизации (роли)
`pg_auth_members`	отношения членства для объектов авторизации
`pg_cast`	приведения (преобразования типов данных)
`pg_class`	таблицы, индексы, последовательности, представления («отношения»)
`pg_collation`	правила сортировки (параметры локали)
`pg_constraint`	ограничения-проверки, ограничения уникальности, ограничения первичного ключа и внешних ключей
`pg_conversion`	информация о перекодировках
`pg_database`	базы данных в этом кластере
`pg_db_role_setting`	параметры, задаваемые на уровне ролей и баз данных
`pg_default_acl`	права по умолчанию для различных типов объектов
`pg_depend`	зависимости между объектами базы данных
`pg_description`	описания или комментарии к объектам базы данных
`pg_enum`	определения меток и значений перечислений

Имя каталога	Предназначение
pg_event_trigger	событийные триггеры
pg_extension	установленные расширения
pg_foreign_data_wrapper	определения обёрток сторонних данных
pg_foreign_server	определения сторонних серверов
pg_foreign_table	дополнительные свойства сторонних таблиц
pg_index	дополнительные свойства индексов
pg_inherits	иерархия наследования таблиц
pg_init_privs	начальные права для объектов
pg_language	языки для написания функций
pg_largeobject	страницы данных для больших объектов
pg_largeobject_metadata	метаданные для больших объектов
pg_namespace	схемы
pg_opclass	классы операторов методов доступа
pg_operator	операторы
pg_opfamily	семейства операторов методов доступа
pg_parameter_acl	параметры конфигурации, для которых были назначены права
pg_partitioned_table	информация о ключах секционирования таблиц
pg_policy	политики защиты строк
pg_proc	функции и процедуры
pg_profile	профили, представляющие собой наборы ограничений аутентификации
pg_publication	публикации для логической репликации
pg_publication_namespace	сопоставление схем с публикациями
pg_publication_rel	сопоставление отношений с публикациями
pg_range	информация о типах диапазонов
pg_replication_origin	зарегистрированные источники репликации
pg_rewrite	правила перезаписи запросов
pg_role_password	история паролей пользователей
pg_seclabel	метки безопасности для объектов базы данных
pg_sequence	информация о последовательностях
pg_shdepend	зависимости общих объектов
pg_shdescription	комментарии к общим объектам
pg_shseclabel	метки безопасности для общих объектов баз данных
pg_statistic	статистика планировщика
pg_statistic_ext	расширенная статистика планировщика (определение)
pg_statistic_ext_data	расширенная статистика планировщика (собранная статистика)
pg_subscription	подписки логической репликации

Имя каталога	Предназначение
`pg_subscription_rel`	состояние отношений для подписок
`pg_tablespace`	табличные пространства в этом кластере баз данных
`pg_transform`	трансформации (тип данных для преобразований процедурных языков)
`pg_trigger`	триггеры
`pg_ts_config`	конфигурации текстового поиска
`pg_ts_config_map`	сопоставления фрагментов в конфигурациях текстового поиска
`pg_ts_dict`	словари текстового поиска
`pg_ts_parser`	анализаторы текстового поиска
`pg_ts_template`	шаблоны текстового поиска
`pg_type`	типы данных
`pg_user_mapping`	сопоставления пользователей для сторонних серверов

51.2. `pg_aggregate`

В каталоге `pg_aggregate` хранится информация об агрегатных функциях. Агрегатная функция — это такая функция, которая работает с множеством значений (обычно, с содержимым одного столбца всех строк, удовлетворяющих условию запроса) и возвращает одно значение, вычисленное по этому множеству. Типичные агрегатные функции — `sum`, `count` и `max`. Все записи в `pg_aggregate` представляют собой дополнение записей в `pg_proc`. Запись в `pg_proc` определяет имя агрегатной функции, типы входных и выходных данных, а также другие свойства, подобные имеющимся у обычных функций.

Таблица 51.2. Столбцы `pg_aggregate`

Тип столбца **Описание**
`aggfnoid` `regproc` (ссылается на `pg_proc` `.oid`) OID агрегатной функции в `pg_proc`
`aggkind` `char` Тип агрегатной функции: `n` — обычная («normal»), `o` — сортирующая («ordered-set») или `h` — гипотезирующая («hypothetical-set»)
`aggnumdirectargs` `int2` Число непосредственных (не агрегируемых) аргументов для сортирующей или гипотезирующей агрегатной функции (переменный массив аргументов считается одним аргументом). Если равняется `pronargs`, агрегатная функция должна принимать переменный массив и этот массив описывает как агрегатные аргументы, так и окончательные непосредственные аргументы. Всегда равно нулю для обычных агрегатных функций.
`aggtransfn` `regproc` (ссылается на `pg_proc` `.oid`) Функция перехода
`aggfinalfn` `regproc` (ссылается на `pg_proc` `.oid`) Функция завершения (ноль, если её нет)
`aggcombinefn` `regproc` (ссылается на `pg_proc` `.oid`) Комбинирующая функция (ноль, если её нет)
`aggserialfn` `regproc` (ссылается на `pg_proc` `.oid`) Функция сериализации (ноль, если её нет)

Тип столбца **Описание**
`aggdeserialfn` `regproc` (ссылается на `pg_proc` `.oid`) Функция десериализации (ноль, если её нет)
`aggmtransfn` `regproc` (ссылается на `pg_proc` `.oid`) Функция прямого перехода для режима движущегося агрегата (ноль, если её нет)
`aggminvtransfn` `regproc` (ссылается на `pg_proc` `.oid`) Функция обратного перехода для режима движущегося агрегата (ноль, если её нет)
`aggmfinalfn` `regproc` (ссылается на `pg_proc` `.oid`) Функция завершения для режима движущегося агрегата (ноль, если её нет)
`aggfinalextra` `bool` Со значением True в `aggfinalfn` передаются дополнительные фиктивные аргументы
`aggmfinalextra` `bool` Со значением True в `aggmfinalfn` передаются дополнительные фиктивные аргументы
`aggfinalmodify` `char` Признак модифицирования переходного состояния функцией `aggfinalfn`: `r`, если состояние только читается; `s`, если функцию `aggtransfn` нельзя применять после `aggfinalfn`; и `w`, если состояние перезаписывается
`aggmfinalmodify` `char` Аналогично `aggfinalmodify`, но для `aggmfinalfn`
`aggsortop` `oid` (ссылается на `pg_operator` `.oid`) Связанный оператор сортировки (ноль, если его нет)
`aggtranstype` `oid` (ссылается на `pg_type` `.oid`) Тип данных внутреннего состояния (перехода) агрегатной функции
`aggtransspace` `int4` Приблизительный средний размер (в байтах) данных состояния перехода, либо ноль для выбора значения по умолчанию
`aggmtranstype` `oid` (ссылается на `pg_type` `.oid`) Тип данных внутреннего состояния (перехода) для агрегатной функции в режиме движущегося агрегата (ноль в случае отсутствия)
`aggmtransspace` `int4` Приблизительный средний размер (в байтах) данных состояния перехода для режима движущегося агрегата, либо ноль для выбора значения по умолчанию
`agginitval` `text` Начальное значение для состояния перехода. Это текстовое поле, содержащее значение в виде внешнего строкового представления. Если это поле содержит NULL, начальным значением состояния перехода будет NULL.
`aggminitval` `text` Начальное значение для состояния перехода в режиме движущегося агрегата. Это текстовое поле, содержащее значение в виде внешнего строкового представления. Если это поле содержит NULL, начальным значением состояния перехода будет NULL.

Новые агрегатные функции создаются командой `CREATE AGGREGATE`. За дополнительными сведениями о разработке агрегатных функций, значении функций перехода и т. д. обратитесь к Разделу 36.12.

51.3. `pg_am`

В каталоге `pg_am` хранится информация о методах доступа отношений. Каждая строка в нём описывает один метод доступа, поддерживаемый системой. В настоящее время методы доступа задаются только для таблиц и индексов. Требования для реализации табличных и индексных методов доступа подробно рассматриваются в Главе 59 и Главе 60.

Таблица 51.3. Столбцы `pg_am`

Тип столбца	Описание
`oid` `oid`	Идентификатор строки
`amname` `name`	Имя метода доступа
`amhandler` `regproc` (ссылается на `pg_proc` `.oid`)	OID функции-обработчика, предоставляющей информацию о методе доступа
`amtype` `char`	`t` = таблица (включая материализованные представления), `i` = индекс.

> **Примечание**
>
> До Postgres Pro 9.6, в `pg_am` было много дополнительных столбцов, представляющих свойства индексных методов доступа. Теперь эти данные непосредственно видны только на уровне кода C. Однако, чтобы SQL-запросы всё же могли проверять свойства индексных методов, была введена функция `pg_index_column_has_property()` и ряд связанных функций; см. Таблицу 9.74.

51.4. `pg_amop`

В каталоге `pg_amop` хранится информация об операторах, связанных с семействами операторов методов доступа. Каждая строка в нём описывает оператор, являющийся членом семейства операторов. Членом семейства может быть либо оператор *поиска*, либо оператор *упорядочивания*. Оператор может относиться к нескольким семействам, но он не может находиться в одном семействе в более чем одной позиции поиска или позиции упорядочивания. (Допустимо, хотя и маловероятно, что оператор будет использоваться и для поиска, и для упорядочивания.)

Таблица 51.4. Столбцы `pg_amop`

Тип столбца	Описание
`oid` `oid`	Идентификатор строки
`amopfamily` `oid` (ссылается на `pg_opfamily` `.oid`)	Семейство операторов, к которому относится эта запись
`amoplefttype` `oid` (ссылается на `pg_type` `.oid`)	Тип данных левого операнда оператора
`amoprighttype` `oid` (ссылается на `pg_type` `.oid`)	Тип данных правого операнда оператора
`amopstrategy` `int2`	Номер стратегии оператора
`amoppurpose` `char`	Предназначение оператора: `s` — поиск (search), `o` — упорядочивание (ordering)
`amopopr` `oid` (ссылается на `pg_operator` `.oid`)	OID оператора
`amopmethod` `oid` (ссылается на `pg_am` `.oid`)	Индексный метод доступа, для которого предназначено семейство операторов
`amopsortfamily` `oid` (ссылается на `pg_opfamily` `.oid`)	

Тип столбца Описание
Семейство операторов B-дерева, в соответствии с которым сортирует данный оператор, если это оператор упорядочивания; ноль, если это оператор поиска

Запись оператора «поиска» указывает, что по индексу этого семейства операторов можно производить поиск и найти все строки, удовлетворяющие условию `WHERE` *`столбец_индекса оператор константа`*. Очевидно, такой оператор должен возвращать тип `boolean`, а тип его левого операнда должен соответствовать типу данных столбца индекса.

Запись оператора «упорядочивания» указывает, что по индексу этого семейства операторов можно провести сканирование и получить строки в порядке, заданном предложением `ORDER BY` *`столбец_индекса оператор константа`*. Такой оператор может возвращать любой сортируемый тип данных, хотя тип левого операнда должен так же соответствовать типу данных столбца индекса. Точная семантика `ORDER BY` определяется столбцом `amopsortfamily`, который должна указывать на семейство операторов B-дерева для типа, возвращаемого оператором.

Примечание

В настоящее время предполагается, что порядком сортировки для упорядочивающего оператора будет порядок по умолчанию для соответствующего семейства операторов, то есть, `ASC NULLS LAST`. Когда-нибудь для большей гибкости могут быть добавлены дополнительные столбцы, позволяющие явно задавать параметры сортировки.

Поле `amopmethod` в записи оператора должно соответствовать полю `opfmethod` содержащего его семейства (`amopmethod` добавлен сюда намеренно, эта денормализация структуры каталога объясняется соображениями производительности). Также, поля `amoplefttype` и `amoprighttype` должны соответствовать полям `oprleft` и `oprright` в записи `pg_operator`, на которую ссылается данная.

51.5. pg_amproc

В каталоге `pg_amproc` хранится информация об опорных функциях, связанных с семействами операторов методов доступа. Строки в нём описывают все опорные функции, принадлежащие семейству операторов.

Таблица 51.5. Столбцы `pg_amproc`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`amprocfamily` `oid` (ссылается на `pg_opfamily` `.oid`) Семейство операторов, к которому относится эта запись
`amproclefttype` `oid` (ссылается на `pg_type` `.oid`) Тип данных левого операнда связанного оператора
`amprocrighttype` `oid` (ссылается на `pg_type` `.oid`) Тип данных правого операнда связанного оператора
`amprocnum` `int2` Номер опорной функции
`amproc` `regproc` (ссылается на `pg_proc` `.oid`) OID функции

Обычно принято, что `amproclefttype` и `amprocrighttype` определяют типы левого и правого операнда оператора, который поддерживает конкретная опорная функция. Для некоторых методов доступа они соответствуют типам входных данных самой опорной функции, для других — нет. Есть понятие «стандартных» опорных функций для индекса; это такие функции, у которых `amproclefttype` и `amprocrighttype` равняются `opcintype` класса оператора индекса.

51.6. `pg_attrdef`

В каталоге `pg_attrdef` хранятся значения столбцов по умолчанию. Основная информация о столбцах хранится в `pg_attribute`. Данный же каталог содержит записи только для тех столбцов, для которых явно задаётся значение по умолчанию.

Таблица 51.6. Столбцы `pg_attrdef`

Тип столбца **Описание**
`oid` `oid` Идентификатор строки
`adrelid` `oid` (ссылается на `pg_class` `.oid`) Таблица, к которой принадлежит столбец
`adnum` `int2` (ссылается на `pg_attribute` `.attnum`) Номер столбца
`adbin` `pg_node_tree` Значение столбца по умолчанию, в формате `nodeToString()` . Чтобы преобразовать его в SQL-выражение, воспользуйтесь функцией `pg_get_expr(adbin, adrelid)` .

51.7. `pg_attribute`

В каталоге `pg_attribute` хранится информация о столбцах таблицы. Для каждого столбца каждой таблицы в `pg_attribute` существует ровно одна строка. (Также в этом каталоге будут записи для индексов и на самом деле для всех объектов, присутствующих в `pg_class`.)

Термин «атрибут» равнозначен «столбцу» и употребляется по историческим причинам.

Таблица 51.7. Столбцы `pg_attribute`

Тип столбца **Описание**
`attrelid` `oid` (ссылается на `pg_class` `.oid`) Таблица, к которой принадлежит столбец
`attname` `name` Имя столбца
`atttypid` `oid` (ссылается на `pg_type` `.oid`) Тип данных этого столбца (ноль для удалённого столбца)
`attlen` `int2` Копия `pg_type.typlen` из записи типа столбца
`attnum` `int2` Порядковый номер столбца. Обычные столбцы нумеруются по возрастанию, начиная с 1. Системные столбцы, такие как `ctid`, имеют (обычно) отрицательные номера.
`attcacheoff` `int4` Всегда -1 в постоянном хранилище, но когда запись загружается в память, в этом поле может кешироваться смещение атрибута в строке
`atttypmod` `int4` В поле `atttypmod` записывается дополнительное число, связанное с определённым типом данных, указываемое при создании таблицы (например, максимальный размер столбца `varchar`). Это значение передаётся функциям ввода и преобразования длины конкретного типа. Для типов, которым не нужен `atttypmod`, это обычно -1.
`attndims` `int2`

Тип столбца	**Описание**
	Число размерностей, если столбец имеет тип массива; ноль в противном случае. (В настоящее время число размерностей массива не контролируется, поэтому любое ненулевое значение по сути означает «это массив».)
`attbyval` `bool`	Копия `pg_type.typbyval` из записи типа столбца
`attalign` `char`	Копия `pg_type.typalign` из записи типа столбца
`attstorage` `char`	Обычно копия `pg_type.typstorage` из записи типа столбца. Для типов, поддерживающих TOAST, можно изменять это значение после создания столбца и таким образом управлять политикой хранения.
`attcompression` `char`	Текущий метод сжатия столбца. Обычно задано значение `'\0'`, то есть действует метод, установленный по умолчанию (см. default_toast_compression). В других случаях: если указано значение `'p'`, используется алгоритм сжатия `pglz`, если указано `'l'` — алгоритм LZ4. Это значение игнорируется, если `attstorage` не поддерживает сжатие.
`attnotnull` `bool`	Представляет ограничение NOT NULL.
`atthasdef` `bool`	Столбец имеет значение по умолчанию или генерирующее выражение, в этом случае в каталоге `pg_attrdef` будет соответствующая запись, собственно задающая это выражение. (Определить, является ли это выражение генерирующим или оно вычисляет значение по умолчанию, можно по атрибуту `attgenerated`.)
`atthasmissing` `bool`	Столбец имеет значение, которое используется, когда он полностью отсутствует в строке. Это имеет место, когда столбец добавляется с неизменчивым значением `DEFAULT` после создания строки. Фактическое значение хранится в `attmissingval`.
`attidentity` `char`	Пустой символ (`''`) указывает, что это не столбец идентификации. Символ `a` указывает, что значение генерируется всегда, а `d` — что значение генерируется по умолчанию.
`attgenerated` `char`	Если нулевой байт (`''`), это не генерируемый столбец. В противном случае, `s` означает, что генерируемое значение хранится (stored). (Другие значения могут быть добавлены в будущем.)
`attisdropped` `bool`	Столбец был удалён и теперь не является рабочим. Удалённый столбец может по-прежнему физически присутствовать в таблице, но анализатор запросов его игнорирует, так что обратиться к нему из SQL нельзя.
`attislocal` `bool`	Столбец определён локально в данном отношении. Заметьте, что столбец может быть определён локально и при этом наследоваться.
`attinhcount` `int2`	Число прямых предков этого столбца. Столбец с ненулевым числом предков нельзя удалить или переименовать.
`attcollation` `oid` (ссылается на `pg_collation` `.oid`)	Заданное для столбца правило сортировки, либо ноль, если тип столбца не сортируемый.
`attstattarget` `int2`	Столбец `attstattarget` управляет детализацией статистики, собираемой по этому столбцу командой `ANALYZE`. Нулевое значение указывает, что статистика не собирается. При

Тип столбца	Описание
	значении NULL используется системное ограничение статистики по умолчанию. Точное значение положительных величин определяется типом данных. Для скалярных типов данных, `attstattarget` задаёт и целевое число собираемых «самых частых значений», и целевое число создаваемых групп гистограммы.
`attacl aclitem[]`	Права доступа к столбцу, если они были заданы непосредственно для этого столбца
`attoptions text[]`	Параметры уровня атрибута, в виде строк «ключ=значение»
`attfdwoptions text[]`	Параметры уровня атрибута для обёрток сторонних данных, в виде строк «ключ=значение»
`attmissingval anyarray`	В данном столбце размещается массив с одним элементом. Значение в этом элементе используется, когда столбец полностью отсутствует в строке, что имеет место, когда столбец добавляется с неизменчивым значением `DEFAULT` после создания строки. Это значение используется, только когда `atthasmissing` равен true. Если значение отсутствует, столбец будет содержать NULL.

В записи удалённого столбца в `pg_attribute` поле `atttypid` сбрасывается в ноль, но `attlen` и другие поля, копируемые из `pg_type`, сохраняют актуальные значения. Это нужно, чтобы справиться с ситуацией, когда после удаления столбца удаляется и его тип данных, так что записи в `pg_type` больше не будет. В таких случаях для интерпретации содержимого строки таблицы могут использоваться `attlen` и другие поля.

51.8. `pg_authid`

В каталоге `pg_authid` содержится информация об идентификаторах для авторизации (ролях). Роль включает в себя концепции «пользователей» и «групп». Пользователь по существу представляет собой частный случай роли, с флагом `rolcanlogin`. Любая роль (с или без флага `rolcanlogin`) может включать другие роли в качестве членов; см. `pg_auth_members`.

Так как в этом каталоге содержатся пароли, он не должен быть открыт для всех. Для общего пользования предназначено представление `pg_roles` на базе `pg_authid`, в котором поле пароля очищено.

За подробной информацией о пользователях и управлении правами обратитесь к Главе 20.

Так как пользователи определяются глобально, каталог `pg_authid` разделяется всеми базами данных кластера; есть только единственная копия `pg_authid` в кластере, а не отдельные в каждой базе данных.

Таблица 51.8. Столбцы `pg_authid`

Тип столбца	Описание
`oid oid`	Идентификатор строки
`rolname name`	Имя роли
`rolsuper bool`	Роли имеет права суперпользователя
`rolinherit bool`	Роль автоматически наследует права ролей, в которые она включена
`rolcreaterole bool`	

Тип столбца **Описание**
Роль может создавать другие роли
`rolcreatedb` `bool` Роль может создавать базы данных
`rolcanlogin` `bool` Роль может подключаться к серверу. То есть эта роль может быть указана в качестве начального идентификатора авторизации сеанса.
`rolreplication` `bool` Роль является ролью репликации. Такие роли могут устанавливать соединения для репликации и создавать/удалять слоты репликации.
`rolbypassrls` `bool` Роль не подчиняется никаким политикам защиты на уровне строк; за подробностями обратитесь к Разделу 5.9.
`rolconnlimit` `int4` Для ролей, которые могут подключаться к серверу, это значение задаёт максимально разрешённое для этой роли число одновременных подключений. При значении -1 ограничения нет.
`rolprofile` `regprofile` (ссылается на `pg_profile` `.oid`) OID профиля пользователя
`rolloginattempts` `int4` Число следующих подряд неудачных попыток входа пользователя. Это число равно `0`, если параметр `FAILED_LOGIN_ATTEMPTS` соответствующего профиля равен `UNLIMITED` (см. CREATE PROFILE). После успешного входа значение `rolloginattempts` также сбрасывается в `0`.
`rollastlogin` `timestamptz` Время последнего подключения пользователя роли
`rolfirstfailedauth` `timestamptz` Время первой неудачной попытки подключения пользователя роли
`rolstatus` `int2` Состояние роли: `0` — роль активна, `1` — роль заблокирована вручную (см. параметр `ACCOUNT LOCK` в ALTER ROLE), `2` — роль заблокирована из-за бездействия (см. параметр `USER_INACTIVE_TIME` в CREATE PROFILE), `4` — роль заблокирована из-за превышения предельного числа неудачных попыток входа (см. параметр `FAILED_LOGIN_ATTEMPTS` в CREATE PROFILE)
`rolpassword` `text` Зашифрованный пароль; NULL, если он не задан. Его формат зависит от используемого вида шифрования.
`rolvaliduntil` `timestamptz` Срок действия пароля (используется только при аутентификации по паролю); NULL, если срок действия не ограничен
`rolpasssetat` `timestamptz` Время установления текущего пароля (используется только при аутентификации по паролю); null, если пароль не установлен.

Если пароль зашифрован MD5, значение в `rolpassword` начинается со строки `md5`, за которой идёт 32-символьный шестнадцатеричный хеш MD5. Этот хеш вычисляется для пароля пользователя с добавленным за ним его именем. Например, если у пользователя `joe` пароль `xyzzy`, Postgres Pro сохранит в этом поле md5-хеш строки `xyzzyjoe`.

Если пароль зашифрован по алгоритму SCRAM-SHA-256, он имеет формат:

```
SCRAM-SHA-256$<число итераций>:<соль>$<СохранённыйКлюч>:<КлючСервера>
```

Здесь *соль*, *СохранённыйКлюч* и *КлючСервера* кодируются в формате Base64. Этот формат соответствует стандарту *RFC 5803*.

51.9. pg_auth_members

Каталог `pg_auth_members` представляет отношения членства между ролями. Допускается любая не зацикленная иерархия отношений.

Так как пользователи определяются глобально, `pg_auth_members` разделяется всеми базами данных кластера; есть только единственная копия `pg_auth_members` в кластере, а не отдельные в каждой базе данных.

Таблица 51.9. Столбцы pg_auth_members

Тип столбца Описание
`oid` `oid` Идентификатор строки
`roleid` `oid` (ссылается на `pg_authid`.`oid`) Идентификатор роли, включающей другую
`member` `oid` (ссылается на `pg_authid`.`oid`) Идентификатор роли, включаемой в роль `roleid`
`grantor` `oid` (ссылается на `pg_authid`.`oid`) Идентификатор роли, разрешившей членство
`admin_option` `bool` True, если `member` может разрешать членство в `roleid` другим ролям
`inherit_option` `bool` True, если член роли наследует привилегии выданной роли
`set_option` `bool` True, если член роли может выполнять `SET ROLE` для выданной роли

51.10. pg_cast

В каталоге `pg_cast` хранятся пути приведения типов (как встроенных, так и пользовательских).

Следует заметить, что `pg_cast` представляет не каждое приведение, которое может выполнять система, а только те, которые нельзя вывести по некоторым общим правилам. Например, приведение типа домена к его базовому типу не представляется явно в `pg_cast`. Ещё одно важное исключение составляют «автоматические приведения ввода-вывода», которые выполняются с применением собственных функций ввода-вывода типа данных, преобразующих тип в `text` или другие строковые типы — они также не представляются явно в `pg_cast`.

Таблица 51.10. Столбцы pg_cast

Тип столбца Описание
`oid` `oid` Идентификатор строки
`castsource` `oid` (ссылается на `pg_type`.`oid`) OID исходного типа данных
`casttarget` `oid` (ссылается на `pg_type`.`oid`) OID целевого типа данных
`castfunc` `oid` (ссылается на `pg_proc`.`oid`) OID функции, выполняющей приведение, или ноль, если для данного метода приведения не требуется функция.

Тип столбца **Описание**
`castcontext char` Определяет, в каких контекстах может выполняться приведение. Символ `e` означает только явное приведение (explicit), с применением синтаксиса `CAST` или `::`. Символ `a` означает неявное присваивание (assignment) целевому столбцу, а также явное приведение. Символ `i` разрешает неявное приведение (implicit), а также все остальные варианты.
`castmethod char` Показывает, как выполняется приведение. Символ `f` означает, что используется функция, указанная в поле `castfunc`. Символ `i` означает, что используются функции ввода-вывода. Символ `b` означают, что типы являются двоично-сводимыми, так что преобразование не требуется.

Функции приведения, перечисленные в `pg_cast`, должны всегда принимать исходный тип приведения в качестве типа первого аргумента и возвращать результат, имеющий целевой тип. Функция приведения может иметь до трёх аргументов. Второй аргумент, если он присутствует, должен иметь тип `integer`; в нём передаётся модификатор типа, связанный с целевым типом, либо -1, если такого модификатора нет. Третий аргумент, если он присутствует, должен иметь тип `boolean`; в нём передаётся `true`, если приведение выполняется явно, и `false` в противном случае.

Вполне возможно создать запись в `pg_cast`, в которой исходный тип будет совпадать с целевым, если связанная функция приведения принимает больше одного аргумента. Такие записи представляют «функции преобразования длины», которые приводят значения некоторого типа в соответствие с заданным значением модификатора.

Когда в записи `pg_cast` исходный и целевой типы приведения различаются и функция принимает более одного аргумента, эта запись представляет преобразование типа из одного в другой и сведение к нужной длине за один шаг. Если же такой записи не находится, приведение к типу с определённым модификатором выполняется в два этапа, сначала выполняется преобразование типа, а затем применяется модификатор типа.

51.11. pg_class

В каталоге `pg_class` описываются таблицы и прочие объекты, имеющие столбцы или каким-то образом подобные таблицам. Сюда входят индексы (но смотрите также `pg_index`), последовательности (но смотрите также `pg_sequence`), представления, материализованные представления, составные типы и таблицы TOAST; см. `relkind`. Далее, подразумевая все эти типы объектов, мы будем говорить об «отношениях». Не все столбцы `pg_class` здесь имеют смысл для всех видов отношений.

Таблица 51.11. Столбцы `pg_class`

Тип столбца **Описание**
`oid oid` Идентификатор строки
`relname name` Имя таблицы, индекса, представления и т. п.
`relnamespace oid` (ссылается на `pg_namespace .oid`) OID пространства имён, содержащего это отношение
`reltype oid` (ссылается на `pg_type .oid`) OID типа данных, соответствующего типу строки этой таблицы, если таковой есть; ноль для индексов, последовательностей и TOAST-таблиц, так как они не имеют записи в `pg_type`
`reloftype oid` (ссылается на `pg_type .oid`)

Тип столбца **Описание**
OID нижележащего составного типа для типизированных таблиц; ноль для всех других отношений
`relowner` `oid` (ссылается на `pg_authid`.`oid`) Владелец отношения
`relam` `oid` (ссылается на `pg_am`.`oid`) Метод доступа к таблице или индексу. Если у отношения нет файла на диске или оно является последовательностью, указанное значение ни на что не влияет, кроме случаев с секционированными таблицами. Для них это значение имеет приоритет над `default_table_access_method` при определении метода доступа, используемого для создаваемых секций, если он не указан явно в команде создания.
`relfilenode` `oid` Имя файла на диске с этим отношением; ноль означает, что это «отображённое» представление, имя файла для которого определяется состоянием на нижнем уровне
`reltablespace` `oid` (ссылается на `pg_tablespace`.`oid`) Табличное пространство, в котором хранится это отношение. Если указан ноль, подразумевается табличное пространство базы данных по умолчанию. Если у отношения нет файла на диске, указанное значение ни на что не влияет, кроме случаев с секционированными таблицами, где это табличное пространство, в котором будут создаваться секции, если пространство не указано явно в команде создания.
`relpages` `int4` Размер представления этой таблицы на диске (в страницах размера `BLCKSZ`). Это лишь примерная оценка, используемая планировщиком. Она обновляется командами `VACUUM`, `ANALYZE` и несколькими командами DDL, например `CREATE INDEX`.
`reltuples` `float4` Число актуальных строк в таблице. Это лишь примерная оценка, используемая планировщиком. Она обновляется командами `VACUUM`, `ANALYZE` и несколькими командами DDL, например `CREATE INDEX`. Если таблица ещё не подвергалась очистке или анализу, поле `reltuples` будет содержать `-1`, указывая на то, что количество строк неизвестно.
`relallvisible` `int4` Число страниц, помеченных как «полностью видимые» в карте видимости таблицы. Это лишь примерная оценка, используемая планировщиком. Она обновляется командами `VACUUM`, `ANALYZE` и несколькими командами DDL, например `CREATE INDEX`.
`reltoastrelid` `oid` (ссылается на `pg_class`.`oid`) OID таблицы TOAST, связанной с данной таблицей, или ноль, если таковой нет. В таблицу TOAST, как во вторичную, «выносятся» большие атрибуты.
`relhasindex` `bool` True, если это таблица и она имеет (или недавно имела) индексы
`relisshared` `bool` True, если эта таблица разделяется всеми базами данных в кластере. Разделяемыми являются только некоторые системные каталоги (например `pg_database`).
`relpersistence` `char` `p` = постоянная таблица/последовательность (permanent), `u` = нежурналируемая таблица/последовательность (unlogged), `t` = временная таблица/последовательность (temporary)
`relkind` `char` `r` = обычная таблица (Relation), `i` = индекс (Index), `S` = последовательность (Sequence), `t` = таблица TOAST, `v` = представление (View), `m` = материализованное представление (Materialized view), `c` = составной тип (Composite type), `f` = сторонняя таблица (Foreign table), `p` = секционированная таблица (Partitioned table), `I` = секционированный индекс (partitioned Index)

Тип столбца	Описание
`relnatts` `int2`	Число пользовательских столбцов в отношении (системные столбцы не считаются). Столько же соответствующих строк должно быть в `pg_attribute` . См. также `pg_attribute` .`attnum`.
`relchecks` `int2`	Число ограничений `CHECK` в таблице; см. каталог `pg_constraint`
`relhasrules` `bool`	True, если для таблицы определены (или были определены) правила; см. каталог `pg_rewrite`
`relhastriggers` `bool`	True, если для таблицы определены (или были определены) триггеры; см. каталог `pg_trigger`
`relhassubclass` `bool`	True, если у таблицы или индекса есть (или были) секции или потомки в иерархии наследования.
`relrowsecurity` `bool`	True, если для таблицы включена защита на уровне строк; см. каталог `pg_policy`
`relforcerowsecurity` `bool`	True, если защита на уровне строк (когда она включена) также применяется к владельцу таблицы; см. каталог `pg_policy`
`relispopulated` `bool`	True, если отношение наполнено данными (это истинно для всех отношений, кроме некоторых материализованных представлений)
`relreplident` `char`	Столбцы, формирующие «идентификатор реплики» для строк: `d` = по умолчанию (первичный ключ, если есть), `n` = никакие (nothing), `f` = все столбцы, `i` = индекс со свойством `indisreplident` (если ранее использованный индекс удалён, действует так же, как `n`)
`relispartition` `bool`	True, если таблица или индекс является секцией
`relrewrite` `oid` (ссылается на `pg_class` .`oid`)	Для новых отношений, записываемых в процессе операции DDL, требующей перезаписи таблицы, это поле содержит OID исходного отношения; в противном случае — ноль. Это состояние видимо только внутри; в этом поле никогда не должно быть ненулевого значения для видимого пользователем отношения.
`relfrozenxid` `xid`	Идентификаторы транзакций, предшествующие данному, в этой таблице заменены постоянным («замороженным») идентификатором транзакции. Это нужно для определения, когда требуется очищать таблицу для предотвращения зацикливания идентификаторов или для сокращения объёма `pg_xact` . Если это отношение — не таблица, значение равно нулю (`InvalidTransactionId`).
`relminmxid` `xid`	Идентификаторы мультитранзакций, предшествующие данному, в этой таблице заменены другим идентификатором транзакции. Это нужно для определения, когда требуется очищать таблицу для предотвращения зацикливания идентификаторов мультитранзакций или для сокращения объёма `pg_multixact` . Если это отношение — не таблица, значение равно нулю (`InvalidMultiXactId`).
`relacl` `aclitem[]`	Права доступа; за подробностями обратитесь к Разделу 5.8.

Тип столбца Описание
`reloptions` `text[]` Специальные параметры для методов доступа, в виде строк «ключ=значение»
`relpartbound` `pg_node_tree` Если таблица является секцией (см. `relispartition`), внутреннее представление границ секции

Некоторые логические флаги в `pg_class` поддерживаются не строго: гарантируется, что они будут установлены при переходе в определённое состояние, но они могут не сбрасываться немедленно, когда условия поменяются. Например, `relhasindex` устанавливается командой `CREATE INDEX`, но никогда не сбрасывается командой `DROP INDEX`. Вместо этого, флаг `relhasindex` сбрасывается командой `VACUUM`, если она находит, что в таблице нет индексов. Такая организация позволяет избежать состояния гонки и способствует параллельному использованию.

51.12. pg_collation

В каталоге `pg_collation` описываются доступные правила сортировки, которые по сути представляют собой сопоставления идентификаторов SQL с категориями локалей операционной системы. За дополнительными сведениями обратитесь к Разделу 22.2.

Таблица 51.12. Столбцы `pg_collation`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`collname` `name` Имя правила сортировки (уникальное для пространства имён и кодировки)
`collnamespace` `oid` (ссылается на `pg_namespace` `.oid`) OID пространства имён, содержащего это правило сортировки
`collowner` `oid` (ссылается на `pg_authid` `.oid`) Владелец правила сортировки
`collprovider` `char` Провайдер правила сортировки: `d` = установленный в базе по умолчанию, `b` = builtin (встроенный), `c` = libc, `i` = icu
`collisdeterministic` `bool` Является ли правило сортировки детерминированным?
`collencoding` `int4` Кодировка, для которой применимо это правило, или -1, если оно работает с любой кодировкой
`collcollate` `text` `LC_COLLATE` для этого объекта. Если провайдер локалей — не `libc`, `collcollate` имеет значение `NULL` и вместо него используется `colllocale`.
`collctype` `text` `LC_CTYPE` для этого объекта. Если провайдер локалей — не `libc`, `collcollate` имеет значение `NULL` и вместо него используется `colllocale`.
`colllocale` `text` Имя локали провайдера правил сортировки для этого объекта. Если провайдер — `libc`, `collcollate` имеет значение `NULL` и вместо него используются `colllocale` и `collctype`.
`collicurules` `text` Правила сортировки ICU для данного объекта

Тип столбца Описание
`collversion` `text` Определяемая провайдером версия правила сортировки. Она записывается при создании правила сортировки и проверяется при использовании для обнаружения изменений в его определении, чреватых повреждением данных.

Заметьте, что уникальный ключ в этом каталоге определён как (`collname`, `collencoding`, `collnamespace`), а не просто как (`collname`, `collnamespace`). Вообще Postgres Pro игнорирует все правила сортировки, для которых `collencoding` не равняется кодировке текущей базы данных или -1, а создание новых записей с тем же именем, которое уже имеет запись с `collencoding` = -1, запрещено. Таким образом, достаточно использовать полное имя SQL (*схема.имя*) для указания правила сортировки, несмотря на то, что оно может быть неуникальным согласно определению каталога. Такая организация каталога объясняется тем, что программа initdb наполняет его в момент инициализации кластера записями для всех локалей, обнаруженных в системе, так что она должна иметь возможность сохранить записи для всех кодировок, которые могут вообще когда-либо применяться в кластере.

В базе данных `template0` может быть полезно создать правила сортировки, кодировки которых не соответствуют кодировке этой базы, но которые могут оказаться у баз данных, скопированных впоследствии из `template0`. В настоящее время это придётся проделать вручную.

51.13. pg_constraint

В каталоге `pg_constraint` хранятся ограничения-проверки, ограничения-исключения, а также ограничения первичного ключа, уникальности и внешних ключей, определённые для таблиц, а также ограничения на NULL, определённые для доменов. (Ограничения столбцов описываются как и все остальные. Любое ограничение столбца равнозначно некоторому ограничению таблицы.) Ограничения на NULL, определяемые для отношений, представляются не здесь, а в каталоге `pg_attribute`.

Для пользовательских триггеров ограничений (создаваемых командой `CREATE CONSTRAINT TRIGGER`) в этой таблице также создаётся запись.

Здесь также хранятся ограничения доменов.

Таблица 51.13. Столбцы `pg_constraint`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`conname` `name` Имя ограничения (не обязательно уникальное!)
`connamespace` `oid` (ссылается на `pg_namespace` `.oid`) OID пространства имён, содержащего это ограничение
`contype` `char` `c` = ограничение-проверка (check), `f` = внешний ключ (foreign key), `n` = ограничение на NULL (not-null, только для доменов), `p` = первичный ключ (primary key), `u` = ограничение уникальности (unique), `t` = триггер ограничения (trigger), `x` = ограничение-исключение (exclusion)
`condeferrable` `bool` Является ли ограничение откладываемым?
`condeferred` `bool` Является ли ограничение отложенным по умолчанию?
`convalidated` `bool`

Тип столбца **Описание**
Было ли ограничение проверено? В настоящее время значение false возможно только для внешних ключей и ограничений CHECK
`conrelid` `oid` (ссылается на `pg_class` .oid) Таблица, для которой установлено это ограничение; ноль, если это не ограничение таблицы
`contypid` `oid` (ссылается на `pg_type` .oid) Домен, к которому относится это ограничение; ноль, если это не ограничение домена
`conindid` `oid` (ссылается на `pg_class` .oid) Индекс, поддерживающий это ограничение, если это ограничение уникальности, первичного или внешнего ключа, либо ограничение-исключение; в противном случае — ноль
`conparentid` `oid` (ссылается на `pg_constraint` .oid) Соответствующее ограничение в родительской секционированной таблице, если это ограничение в секции; иначе ноль
`confrelid` `oid` (ссылается на `pg_class` .oid) Если это внешний ключ, таблица, на которую он ссылается; иначе ноль
`confupdtype` `char` Код действия при изменении внешнего ключа: `a` = нет действия, `r` = ограничить (restrict), `c` = каскадное действие (cascade), `n` = присвоить NULL, `d` = поведение по умолчанию
`confdeltype` `char` Код действия при удалении внешнего ключа: `a` = нет действия, `r` = ограничить (restrict), `c` = каскадное действие (cascade), `n` = присвоить NULL, `d` = поведение по умолчанию
`confmatchtype` `char` Тип сопоставления внешнего ключа: `f` = полное (full), `p` = частичное (partial), `s` = простое (simple)
`conislocal` `bool` Ограничение определено локально в данном отношении. Заметьте, что ограничение может быть определено локально и при этом наследоваться.
`coninhcount` `int2` Число прямых предков этого ограничения. Ограничение с ненулевым числом предков нельзя удалить или переименовать.
`connoinherit` `bool` Ограничение определено локально для данного отношения и является ненаследуемым.
`conkey` `int2[]` (ссылается на `pg_attribute` .attnum) Для ограничений таблицы (включая внешние ключи, но не триггеры ограничений), определяет список столбцов, образующих ограничение
`confkey` `int2[]` (ссылается на `pg_attribute` .attnum) Для внешнего ключа определяет список столбцов, на которые он ссылается
`conpfeqop` `oid[]` (ссылается на `pg_operator` .oid) Для внешнего ключа — список операторов равенства для сравнений PK = FK
`conppeqop` `oid[]` (ссылается на `pg_operator` .oid) Для внешнего ключа — список операторов равенства для сравнений PK = PK
`conffeqop` `oid[]` (ссылается на `pg_operator` .oid) Для внешнего ключа — список операторов равенства для сравнений FK = FK
`confdelsetcols` `int2[]` (ссылается на `pg_attribute` .attnum)

Тип столбца **Описание**
Для внешнего ключа с указанием `SET NULL` или `SET DEFAULT` при удалении — список изменяемых столбцов. Значение NULL показывает, что изменены будут все столбцы, на которые ссылается ключ.
`conexclop` `oid[]` (ссылается на `pg_operator` .`oid`) Для ограничения-исключения — список операторов исключения по столбцам
`conbin` `pg_node_tree` Для ограничения-проверки — внутреннее представление выражения. (Чтобы извлечь определение ограничения-проверки, рекомендуется использовать `pg_get_constraintdef()` .)

В случае с ограничением-исключением значение `conkey` полезно только для элементов ограничений, представляющих простые ссылки на столбцы. Для других случаев в `conkey` задаётся ноль, и чтобы получить выражение, определяющее ограничение, надо обратиться к соответствующему индексу. (Таким образом, поле `conkey` имеет то же содержимое, что и `pg_index`.`indkey` для индекса.)

> **Примечание**
>
> Поле `pg_class.relchecks` должно согласовываться с числом ограничений-проверок, описанных в данной таблице для каждого отношения.

51.14. pg_conversion

В каталоге `pg_conversion` описываются функции преобразования кодировки. За дополнительными сведениями обратитесь к CREATE CONVERSION.

Таблица 51.14. Столбцы `pg_conversion`

Тип столбца **Описание**
`oid` `oid` Идентификатор строки
`conname` `name` Имя перекодировки (уникальное в пространстве имён)
`connamespace` `oid` (ссылается на `pg_namespace` .`oid`) OID пространства имён, содержащего эту перекодировку
`conowner` `oid` (ссылается на `pg_authid` .`oid`) Владелец перекодировки
`conforencoding` `int4` ID кодировки источника (`pg_encoding_to_char()` может преобразовать этот номер в имя кодировки)
`contoencoding` `int4` ID кодировки назначения (`pg_encoding_to_char()` может преобразовать этот номер в имя кодировки)
`conproc` `regproc` (ссылается на `pg_proc` .`oid`) Функция преобразования
`condefault` `bool` True, если это перекодировка по умолчанию

51.15. pg_database

В каталоге `pg_database` хранится информация о доступных базах данных. Базы данных создаются командой `CREATE DATABASE`. Подробнее о предназначении некоторых свойств баз можно узнать в Главе 21.

В отличие от большинства системных каталогов, `pg_database` разделяется всеми базами данных кластера: есть только один экземпляр `pg_database` в кластере, а не отдельные в каждой базе данных.

Таблица 51.15. Столбцы `pg_database`

Тип столбца	Описание
`oid` `oid`	Идентификатор строки
`datname` `name`	Имя базы данных
`datdba` `oid` (ссылается на `pg_authid` `.oid`)	Владелец базы данных, обычно пользователь, создавший её
`encoding` `int4`	Кодировка символов для этой базы данных (`pg_encoding_to_char()` может преобразовать этот номер в имя кодировки)
`datlocprovider` `char`	Провайдер локали для этой базы данных: `b` = builtin, `c` = libc, `i` = icu
`datistemplate` `bool`	Если true, базу данных сможет клонировать любой пользователь с правами `CREATEDB`; в противном случае клонировать эту базу смогут только суперпользователи и её владелец.
`datallowconn` `bool`	Если false, никто не сможет подключаться к этой базе данных. Это позволяет защитить базу данных `template0` от модификаций.
`dathasloginevt` `bool`	Указывает, что для этой базы данных заданы событийные триггеры входа в систему. Этот флаг используется, чтобы избежать дополнительных операций поиска в таблице `pg_event_trigger` при запуске каждого обслуживающего процесса. Этот флаг используется внутри Postgres Pro и не должен изменяться вручную или считываться в целях мониторинга.
`datconnlimit` `int4`	Задаёт максимально допустимое число одновременных подключений к этой базе данных. С -1 ограничения нет, с -2 к базе данных нельзя подключаться.
`datfrozenxid` `xid`	Все идентификаторы транзакций, предшествующие данному, в этой базе данных заменены постоянным («замороженным») идентификатором транзакции. Это нужно для определения, когда требуется очищать базу данных для предотвращения зацикливания идентификаторов или для сокращения объёма `pg_xact` . Это значение вычисляется как минимум значений `pg_class` `.relfrozenxid` для всех таблиц.
`datminmxid` `xid`	Идентификаторы мультитранзакций, предшествующие данному, в этой базе данных заменены другим идентификатором транзакции. Это нужно для определения, когда требуется очищать базу данных для предотвращения зацикливания идентификаторов мультитранзакций или для сокращения объёма `pg_multixact` . Это значение вычисляется как минимум значений `pg_class` `.relminmxid` для всех таблиц.
`dattablespace` `oid` (ссылается на `pg_tablespace` `.oid`)	

Тип столбца Описание
Табличное пространство по умолчанию для данной базы данных. Если таблица базы находится в этом пространстве, для неё значение `pg_class .reltablespace` будет нулевым; в частности, в нём окажутся все частные системные каталоги этой базы.
`datcollate text` LC_COLLATE для этой базы данных
`datctype text` LC_CTYPE для этой базы данных
`datlocale text` Имя локали провайдера правил сортировки для этой базы данных. Если провайдер — `libc`, `datlocale` имеет значение `NULL` и вместо него используются `datcollate` и `datctype`.
`daticurules text` Правила сортировки ICU для этой базы данных
`datcollversion text` Определяемая провайдером версия правила сортировки. Она записывается при создании базы данных и проверяется при использовании для обнаружения изменений в его определении, чреватых повреждением данных.
`datacl aclitem[]` Права доступа; за подробностями обратитесь к Разделу 5.8.

51.16. pg_db_role_setting

В каталоге `pg_db_role_setting` записываются значения по умолчанию, присваиваемые переменным конфигурации во время выполнения, для различных комбинаций ролей и баз данных.

В отличие от большинства системных каталогов, `pg_db_role_setting` разделяется всеми базами данных кластера: есть только один экземпляр `pg_db_role_setting` в кластере, а не отдельные в каждой базе данных.

Таблица 51.16. Столбцы `pg_db_role_setting`

Тип столбца Описание
`setdatabase oid` (ссылается на `pg_database .oid`) OID базы данных, к которой относится это присваивание переменных, или ноль, если оно не связано с базой данных
`setrole oid` (ссылается на `pg_authid .oid`) OID роли, к которой применимо это присваивание переменных, или ноль, если оно не связано с ролью
`setconfig text[]` Значения по умолчанию для переменных конфигурации времени выполнения

51.17. pg_default_acl

В каталоге `pg_default_acl` хранятся определения прав, изначально присваиваемые создаваемым объектам.

Таблица 51.17. Столбцы `pg_default_acl`

Тип столбца Описание
`oid oid`

Тип столбца **Описание**
Идентификатор строки
`defaclrole` `oid` (ссылается на `pg_authid` `.oid`) OID роли, связанной с этой записью
`defaclnamespace` `oid` (ссылается на `pg_namespace` `.oid`) OID пространства имён, связанного с этой записью, или ноль, если запись глобальная
`defaclobjtype` `char` Тип объекта, права для которого определяет эта запись: `r` = отношение (relation) (таблица, представление), `S` = последовательность (sequence), `f` = функция (function), `T` = тип (type), `n` = схема (schema)
`defaclacl` `aclitem[]` Права доступа, назначаемые объекту данного типа при создании

В каталоге `pg_default_acl` описываются начальные права доступа, которые будут связаны с объектом, принадлежащим заданному пользователю. В настоящее время есть два типа записей: «глобальные», с `defaclnamespace` = ноль, и «внутрисхемные», относящиеся к конкретной схеме. Если присутствует глобальная запись, она *переопределяет* обычный жёстко фиксированный набор прав для данного типа объектов. Если присутствует внутрисхемная запись, она представляет набор прав, *добавляемый* к набору прав, определённых глобально или жёстко заданных по умолчанию.

Заметьте, что когда поле ACL в другом каталоге содержит NULL, под этим подразумеваются жёстко заданные права по умолчанию для этого объекта, а *не* те, которые могут быть в `pg_default_acl` в данный момент. Права в `pg_default_acl` учитываются только при создании объекта.

51.18. pg_depend

В каталоге `pg_depend` записываются отношения зависимости между объектами базы данных. Благодаря этой информации, команды `DROP` могут найти, какие объекты должны удаляться при использовании `DROP CASCADE`, или когда нужно запрещать удаление при `DROP RESTRICT`.

Также смотрите описание каталога `pg_shdepend`, который играет подобную роль в отношении совместно используемых объектов в кластере баз данных.

Таблица 51.18. Столбцы `pg_depend`

Тип столбца **Описание**
`classid` `oid` (ссылается на `pg_class` `.oid`) OID системного каталога, в котором находится зависимый объект
`objid` `oid` (ссылается на какой-либо столбец OID) OID определённого зависимого объекта
`objsubid` `int4` Для столбца таблицы это номер столбца (`objid` и `classid` указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
`refclassid` `oid` (ссылается на `pg_class` `.oid`) OID системного каталога, в котором находится вышестоящий объект
`refobjid` `oid` (ссылается на какой-либо столбец OID) OID определённого вышестоящего объекта
`refobjsubid` `int4` Для столбца таблицы это номер столбца (`refobjid` и `refclassid` указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
`deptype` `char`

Тип столбца **Описание**
Код, определяющий конкретную семантику данного отношения зависимости; см. текст

Во всех случаях, запись в `pg_depend` показывает, что вышестоящий объект нельзя удалить, не удаляя подчинённый объект. Однако есть несколько подвидов зависимости, задаваемых в поле `deptype`:

`DEPENDENCY_NORMAL` (`n`)

Обычное отношение между отдельно создаваемыми объектами. Подчинённый объект можно удалить, не затрагивая вышестоящий объект. Вышестоящий объект можно удалить только с указанием `CASCADE`, при этом будет удалён и подчинённый объект. Например, столбец таблицы находится в обычной зависимости от своего типа данных.

`DEPENDENCY_AUTO` (`a`)

Подчинённый объект может быть удалён отдельно от вышестоящего и должен быть удалён автоматически (вне зависимости от указаний `RESTRICT` и `CASCADE`), если удаляется вышестоящий объект. Например, именованное ограничение для таблицы находится в автоматической зависимости от таблицы, так что оно исчезнет при удалении таблицы.

`DEPENDENCY_INTERNAL` (`i`)

Подчинённый объект был создан в процессе создания вышестоящего и на самом деле является только частью его внутренней реализации. Для такого объекта будет запрещена команда `DROP` (мы подскажем пользователю, что вместо этого надо выполнить `DROP` для вышестоящего объекта). Действие `DROP` для вышестоящего объекта будет автоматически распространено и на этот подчинённый объект, вне зависимости от присутствия указания `CASCADE`. Если подчинённый объект должен быть удалён вследствие зависимости от какого-то другого удаляемого объекта, удаление подчинённого преобразуется в удаление вышестоящего, то есть зависимости `NORMAL` и `AUTO` подчинённого объекта во многом действуют как зависимости вышестоящего объекта. Например, правило `ON SELECT` для представления автоматически становится внутренне зависимым от этого представления, что не позволяет удалить это правило, пока существует представление. Зависимости этого правила (например, от таблиц, которые в нём фигурируют) будут действовать так же, как если бы они были зависимостями самого представления.

`DEPENDENCY_PARTITION_PRI` (`P`)
`DEPENDENCY_PARTITION_SEC` (`S`)

Подчинённый объект (секция) был создан в процессе создания вышестоящего (секционированного отношения) и на самом деле является только частью его внутренней реализации; однако у него есть несколько вышестоящих объектов, что отличает эту зависимость от `INTERNAL`. Такой подчинённый объект должен удаляться только тогда, когда удаляется хотя бы один из этих вышестоящих объектов; в этом случае не должно иметь значения наличие указания `CASCADE`. Также с такой зависимостью, в отличие от `INTERNAL`, попытка удаления некоторого другого объекта, от которого зависит подчинённый, не приводит к автоматическому удалению какого-либо из связанных секционированных отношений. Таким образом, если удаление не доходит до минимум одного из вышестоящих объектов по какому-то пути, оно не будет произведено. (В большинстве случаев подчинённый объект разделяет все свои несекционные зависимости с минимум одним секционно-связанным объектом, чтобы это ограничение не блокировало каскадные удаления.) Первичные (PRI) и вторичные (SEC) секционные зависимости похожи, и отличаются только тем, что первичные зависимости предпочитаются при формировании сообщений об ошибках; поэтому секционно-подчинённый объект, как правило, будет иметь одну первичную секционную зависимость и одну или несколько вторичных. Заметьте, что секционные зависимости создаются в дополнение, но не вместо обычных зависимостей, которые будет иметь объект. Это упрощает операции `ATTACH/DETACH PARTITION`: они будут просто добавлять или удалять секционные зависимости. Например, дочерний секционированный индекс оказывается секционно-зависимым и от собственной секционированной таблицы, и от родительского секционированного индекса, так что он будет удалён при удалении одного из этих объектов, но не в других случаях. Зависимость от родительского секционированного индекса является первичной, поэтому если пользователь пытается удалить дочерний секционированный индекс, в сообщении об ошибке будет предложено удалить родительский индекс (а не таблицу).

`DEPENDENCY_EXTENSION` (e)

Подчинённый объект входит в состав *расширения*, которое является вышестоящим объектом (см. `pg_extension`). Удалить подчинённый объект можно, только выполнив команду `DROP EXTENSION` для вышестоящего объекта. Функционально этот тип зависимости действует так же, как и внутренняя (`INTERNAL`) зависимость, но он выделен для наглядности и упрощения pg_dump.

`DEPENDENCY_AUTO_EXTENSION` (x)

Подчинённый объект не входит в состав расширения, являющегося вышестоящим объектом (и поэтому он не должен игнорироваться программой pg_dump), но он не может функционировать без расширения и должен удаляться автоматически при удалении расширения. Такой объект также может быть удалён сам по себе. Функционально эта зависимость работает как зависимость `AUTO`, но она выделена для наглядности и упрощения pg_dump.

В будущем могут появиться и другие подвиды зависимости.

Заметьте, что два объекта вполне могут быть связаны между собой более чем одной записью в `pg_depend`. Например, дочерний секционированный индекс будет иметь и зависимость секционного типа от связанной секционированной таблицы, и автоматическую зависимость от каждого столбца таблицы, входящего в индекс. В подобных ситуациях имеет место совмещение зависимостей, несущих разных смысл. И если какая-либо из зависимостей удовлетворяет условиям для автоматического удаления, подчинённый объект может быть удалён без указания `CASCADE`. При этом, конечно, должны быть удовлетворены все ограничения зависимостей, определяющих подлежащие удалению объекты.

Большинство объектов, созданных во время работы initdb, считаются «закреплёнными», что означает, что сама система зависит от них. Поэтому их нельзя удалять ни при каких условиях. Кроме того, зная, что закреплённые объекты не будут удалены, механизм зависимостей не создаёт в `pg_depend` записи, показывающие зависимости от этих объектов. Так, например, столбец таблицы типа `numeric` условно имеет зависимость `NORMAL` от типа данных `numeric`, но на самом деле такая запись не появляется в `pg_depend`.

51.19. pg_description

В каталоге `pg_description` хранятся дополнительные описания (комментарии) для объектов баз данных. Описания можно задавать с помощью команды `COMMENT` и просматривать в psql, используя команды `\d`. В начальном содержимом `pg_description` представлены описания многих встроенных системных объектов.

Также смотрите каталог `pg_shdescription`, который играет подобную роль в отношении совместно используемых объектов в кластере баз данных.

Таблица 51.19. Столбцы pg_description

Тип столбца Описание
`objoid` `oid` (ссылается на какой-либо столбец OID) OID объекта, к которому относится это описание
`classoid` `oid` (ссылается на `pg_class` .oid) OID системного каталога, к которому относится этот объект
`objsubid` `int4` Для комментария к столбцу таблицы это номер столбца (`objoid` и `classoid` указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
`description` `text` Произвольный текст, служащий описанием данного объекта

51.20. pg_enum

В каталоге `pg_enum` содержатся записи, определяющие значения и метки для всех типов-перечислений. Внутренним представлением значения перечисления на самом деле является OID соответствующей строки в `pg_enum`.

Таблица 51.20. Столбцы `pg_enum`

Тип столбца	Описание
`oid` `oid`	Идентификатор строки
`enumtypid` `oid` (ссылается на `pg_type`.`oid`)	OID записи в `pg_type`, к которой относится данное значение перечисления
`enumsortorder` `float4`	Порядок сортировки этого значения внутри перечисления
`enumlabel` `name`	Текстовая метка данного значения перечисления

Идентификаторы OID в строках `pg_enum` подчиняются особому правилу: чётные OID гарантированно упорядочиваются по порядку сортировки их типа перечисления. То есть, если к одному перечислению относятся два чётных OID, меньшему OID должно соответствовать меньшее значение `enumsortorder`. Нечётные значения OID могут быть не связаны с этим порядком сортировки. Это правило позволяет во многих случаях сравнивать значения перечислений, не обращаясь к каталогам. Процедуры, создающие и изменяющие перечисления, пытаются присваивать значениям перечислений чётные OID, если это возможно.

Когда создаётся тип перечисления, его членам назначаются позиции по порядку сортировки 1..*n*. Но у членов, добавляемых позже, могут оказаться отрицательные или дробные значения `enumsortorder`. Единственное, что требуется — чтобы эти значения были правильно упорядочены и уникальны в рамках перечисления.

51.21. pg_event_trigger

В каталоге `pg_event_trigger` хранится информация о событийных триггерах. За дополнительными сведениями обратитесь к Главе 38.

Таблица 51.21. Столбцы `pg_event_trigger`

Тип столбца	Описание
`oid` `oid`	Идентификатор строки
`evtname` `name`	Имя триггера (должно быть уникальным)
`evtevent` `name`	Указывает, для какого события срабатывает данный триггер
`evtowner` `oid` (ссылается на `pg_authid`.`oid`)	Владелец событийного триггера
`evtfoid` `oid` (ссылается на `pg_proc`.`oid`)	Вызываемая функция
`evtenabled` `char`	Устанавливает, в каких режимах session_replication_role срабатывает событийный триггер: `O` = триггер срабатывает в режимах «origin» (источник) и «local» (локально), `D` =

Тип столбца Описание
триггер отключён, `R` = триггер срабатывает в режиме «replica» (реплика), `A` = триггер срабатывает всегда.
`evttags text[]` Теги команд, для которых будет срабатывать триггер. Если NULL, срабатывание триггера не будет ограничено в зависимости от тега команды.

51.22. pg_extension

В каталоге `pg_extension` хранится информация об установленных расширениях. Подробнее о расширениях можно узнать в Разделе 36.17.

Таблица 51.22. Столбцы pg_extension

Тип столбца Описание
`oid oid` Идентификатор строки
`extname name` Имя расширения
`extowner oid` (ссылается на `pg_authid .oid`) Владелец расширения
`extnamespace oid` (ссылается на `pg_namespace .oid`) Схема, содержащая экспортируемые расширением объекты
`extrelocatable bool` True, если расширение можно переместить в другую схему
`extversion text` Имя версии расширения
`extconfig oid[]` (ссылается на `pg_class .oid`) Массив с идентификаторами `regclass`, указывающими на таблицы конфигурации расширения, либо `NULL`, если таких таблиц нет
`extcondition text[]` Массив с условиями фильтра `WHERE` для таблиц конфигурации расширения, либо `NULL`, если таких условий нет

Заметьте, что в отличие от большинства каталогов со столбцом «namespace», здесь `extnamespace` не подразумевает, что расширение принадлежит данной схеме. Имена расширений никогда не дополняются схемой. Вместо этого, `extnamespace` показывает, что в этой схеме содержатся все или большинство объектов расширения. Если `extrelocatable` имеет значение true, эта схема должна фактически содержать все относящиеся к схеме объекты, составляющие это расширение.

51.23. pg_foreign_data_wrapper

В каталоге `pg_foreign_data_wrapper` хранятся определения обёрток сторонних данных. Обёрткой сторонних данных называется механизм, через который становятся доступны внешние данные, расположенные на сторонних серверах.

Таблица 51.23. Столбцы pg_foreign_data_wrapper

Тип столбца Описание
`oid oid` Идентификатор строки

Тип столбца Описание
`fdwname name` Имя обёртки сторонних данных
`fdwowner oid` (ссылается на `pg_authid .oid`) Владелец обёртки сторонних данных
`fdwhandler oid` (ссылается на `pg_proc .oid`) Указывает на функцию-обработчик, отвечающую за выдачу набора исполняемых процедур для обёртки сторонних данных. Ноль говорит об отсутствии такого обработчика
`fdwvalidator oid` (ссылается на `pg_proc .oid`) Указывает на функцию проверки, которая отвечает за проверку правильности параметров, передаваемых обёртке сторонних данных, а также параметров для сторонних серверов и сопоставлений пользователей, задаваемых через эту обёртку. Ноль говорит об отсутствии такой функции
`fdwacl aclitem[]` Права доступа; за подробностями обратитесь к Разделу 5.8.
`fdwoptions text[]` Специальные параметры обёртки сторонних данных, в виде строк «ключ=значение»

51.24. pg_foreign_server

В каталоге `pg_foreign_server` хранятся определения сторонних серверов. Запись стороннего сервера описывает источник внешних данных, например удалённый сервер. Обращение к сторонним серверам происходит через обёртки сторонних данных.

Таблица 51.24. Столбцы `pg_foreign_server`

Тип столбца Описание
`oid oid` Идентификатор строки
`srvname name` Имя стороннего сервера
`srvowner oid` (ссылается на `pg_authid .oid`) Владелец стороннего сервера
`srvfdw oid` (ссылается на `pg_foreign_data_wrapper .oid`) OID обёртки сторонних данных для этого стороннего сервера
`srvtype text` Тип сервера (необязателен)
`srvversion text` Версия сервера (необязательна)
`srvacl aclitem[]` Права доступа; за подробностями обратитесь к Разделу 5.8.
`srvoptions text[]` Специальные параметры стороннего сервера, в виде строк «ключ=значение»

51.25. pg_foreign_table

В каталоге `pg_foreign_table` содержится дополнительная информация о сторонних таблицах. Прежде всего сторонняя таблица представляется записью в `pg_class`, как и обычная. Запись в `pg_foreign_table` для неё содержит свойства, применимые только к сторонним таблицам, но не к каким-либо другим типам отношений.

Таблица 51.25. Столбцы `pg_foreign_table`

Тип столбца Описание
`ftrelid` `oid` (ссылается на `pg_class` `.oid`) OID записи в `pg_class` для этой сторонней таблицы
`ftserver` `oid` (ссылается на `pg_foreign_server` `.oid`) OID стороннего сервера для этой сторонней таблицы
`ftoptions` `text[]` Параметры сторонней таблицы, в виде строк «ключ=значение»

51.26. `pg_index`

В каталоге `pg_index` содержится часть информации об индексах. Остальная информация в основном находится в `pg_class`.

Таблица 51.26. Столбцы `pg_index`

Тип столбца Описание
`indexrelid` `oid` (ссылается на `pg_class` `.oid`) OID записи в `pg_class` для этого индекса
`indrelid` `oid` (ссылается на `pg_class` `.oid`) OID записи в `pg_class` для таблицы, к которой относится этот индекс
`indnatts` `int2` Общее число столбцов в индексе (повторяет значение `pg_class.relnatts`). В это число входят и ключевые, и неключевые атрибуты.
`indnkeyatts` `int2` Число *ключевых столбцов* в индексе, без учёта *неключевых столбцов*, которые хранятся в индексе, но не учитываются в его семантике
`indisunique` `bool` Если true, это уникальный индекс
`indnullsnotdistinct` `bool` Используется только для уникальных индексов. Если false, значения NULL для этого уникального индекса будут считаться различными (поэтому индекс сможет содержать несколько значений NULL в столбце — это поведение Postgres Pro по умолчанию). Если true, значения NULL будут считаться равными (и индекс сможет содержать только одно значение NULL в столбце).
`indisprimary` `bool` Если true, этот индекс представляет первичный ключ таблицы (в этом случае и в поле `indisunique` должно быть значение true)
`indisexclusion` `bool` Если true, этот индекс поддерживает ограничение-исключение
`indimmediate` `bool` Если true, проверка уникальности осуществляется непосредственно при добавлении данных (неприменимо, если значение `indisunique` не true)
`indisclustered` `bool` Если true, таблица в последний раз кластеризовалась по этому индексу
`indisvalid` `bool` Если true, индекс можно применять в запросах. Значение false означает, что индекс, возможно, неполный: он будет тем не менее изменяться командами `INSERT`/`UPDATE`, но безопасно применять его в запросах нельзя. Если он уникальный, свойство уникальности также не гарантируется.

Тип столбца **Описание**
`indcheckxmin` `bool` Если true, запросы не должны использовать этот индекс, пока поле `xmin` данной записи в `pg_index` не окажется ниже их горизонта событий `TransactionXmin`, так как таблица может содержать оборванные цепочки HOT с видимыми несовместимыми строками
`indisready` `bool` Если true, индекс готов к добавлению данных. Значение false означает, что индекс игнорируется операциями `INSERT`/`UPDATE`.
`indislive` `bool` Если false, индекс находится в процессе удаления и его следует игнорировать для любых целей (включая вопрос применимости HOT)
`indisreplident` `bool` Если true, этот индекс выбран в качестве «идентификатора реплики» командой `ALTER TABLE ... REPLICA IDENTITY USING INDEX ...`
`indkey` `int2vector` (ссылается на `pg_attribute` `.attnum`) Это массив из `indnatts` значений, указывающих, какие столбцы таблицы индексирует этот индекс. Например, значения `1 3` будут означать, что в индекс входят первый и третий столбцы таблицы. Ключевые столбцы указываются перед неключевыми (включаемыми) столбцами. Ноль в этом массиве означает, что соответствующий атрибут индекса определяется выражением со столбцами таблицы, а не просто ссылкой на столбец.
`indcollation` `oidvector` (ссылается на `pg_collation` `.oid`) Для каждого столбца в ключе индекса этот массив (из `indnkeyatts` значений) содержит OID правила сортировки для применения в этом индексе либо 0, если тип данных этого столбца не сортируемый.
`indclass` `oidvector` (ссылается на `pg_opclass` `.oid`) Для каждого столбца в ключе индекса этот массив (из `indnkeyatts` значений) содержит OID применяемых классов операторов. Подробнее это рассматривается в описании `pg_opclass` .
`indoption` `int2vector` Это массив из `indnkeyatts` значений, в которых хранятся битовые флаги для отдельных столбцов. Значение этих флагов определяется методом доступа конкретного индекса.
`indexprs` `pg_node_tree` Деревья выражений (в представлении `nodeToString()`) для атрибутов индекса, не являющихся простыми ссылками на столбцы. Этот список содержит один элемент для каждого нулевого значения в `indkey`. Значением может быть NULL, если все атрибуты индекса представляют собой простые ссылки.
`indpred` `pg_node_tree` Дерево выражения (в представлении `nodeToString()`) для предиката частичного индекса, либо NULL, если это не частичный индекс.

51.27. `pg_inherits`

В каталоге `pg_inherits` содержится информация об иерархиях наследования таблиц и индексов. Для каждой непосредственной связи «родительский-дочерний объект» в ней содержится одна запись. (Косвенное наследование можно определить, просмотрев цепочку записей.)

Таблица 51.27. Столбцы `pg_inherits`

Тип столбца **Описание**
`inhrelid` `oid` (ссылается на `pg_class` `.oid`) OID дочерней таблицы или индекса

Тип столбца **Описание**
`inhparent` `oid` (ссылается на `pg_class` .`oid`) OID родительской таблицы или индекса
`inhseqno` `int4` Если у дочерней таблицы есть несколько непосредственных родителей (множественное наследование), это число определяет порядок, в котором располагаются наследованные столбцы. Нумерация начинается с 1. Для индексов множественное наследование невозможно, так как они могут наследоваться только при декларативном секционировании.
`inhdetachpending` `bool` `true` для отсоединяемой в данный момент секции; в противном случае — `false`.

51.28. `pg_init_privs`

В каталоге `pg_init_privs` содержится информация об изначально назначаемых правах для объектов в системе. Для каждого объекта в базе данных, имеющего нестандартный (отличный от NULL) начальный набор прав, в ней содержится одна запись.

Начальные права доступа для объектов могут задаваться либо при инициализации базы данных (программой initdb), либо когда объект создаётся в процессе `CREATE EXTENSION` и скрипт расширения задаёт права, задействуя систему `GRANT`. Заметьте, что эта система автоматически записывает права, устанавливаемые скриптом расширения, так что авторам расширений достаточно использовать в своих скриптах только `GRANT` и `REVOKE`, чтобы права были сохранены. Столбец `privtype` показывает, были ли начальные права заданы программой initdb или в процессе выполнения команды `CREATE EXTENSION`.

Для объектов, которым начальные права были назначены программой initdb, записи в `privtype` помечаются буквой `'i'`, а для объектов, которым права назначались в процессе `CREATE EXTENSION`, — буквой `'e'`.

Таблица 51.28. Столбцы `pg_init_privs`

Тип столбца **Описание**
`objoid` `oid` (ссылается на какой-либо столбец OID) OID определённого объекта
`classoid` `oid` (ссылается на `pg_class` .`oid`) OID системного каталога, в котором находится объект
`objsubid` `int4` Для столбца таблицы это номер столбца (`objoid` и `classoid` указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
`privtype` `char` Код, определяющий тип начального права для объекта; см. текст
`initprivs` `aclitem[]` Начальные права доступа; за подробностями обратитесь к Разделу 5.8

51.29. `pg_language`

В каталоге `pg_language` регистрируются языки, на которых возможно писать функции или хранимые процедуры. За дополнительной информацией о языковых обработчиках обратитесь к описанию CREATE LANGUAGE и Главе 40.

Таблица 51.29. Столбцы `pg_language`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`lanname` `name` Имя языка
`lanowner` `oid` (ссылается на `pg_authid` .`oid`) Владелец языка
`lanispl` `bool` Для внутренних языков (например, SQL) содержит false, а для пользовательских языков — true. В настоящее время, pg_dump всё ещё пользуется этим признаком, чтобы определить, какие языки нужно выгружать, но в будущем ему на смену может прийти другой механизм.
`lanpltrusted` `bool` True, если это доверенный язык, что означает, что можно рассчитывать на то, что он не открывает доступ куда-либо за пределы обычной среды исполнения SQL. Создавать функции на недоверенных языках могут только суперпользователи.
`lanplcallfoid` `oid` (ссылается на `pg_proc` .`oid`) Для не внутренних языков это значение указывает на языковой обработчик, который представляет собой специальную функцию, отвечающую за выполнение всех процедур, написанных на этом языке. Для внутренних языков — ноль.
`laninline` `oid` (ссылается на `pg_proc` .`oid`) Это значение указывает на функцию, отвечающую за выполнение «внедрённых» анонимных блоков кода (блоков DO). Ноль, если внедрённые блоки не поддерживаются.
`lanvalidator` `oid` (ссылается на `pg_proc` .`oid`) Это значение указывает на функцию проверки языка, которая отвечает за проверку синтаксиса и правильности новых функций в момент их создания. Ноль, если функция проверки отсутствует.
`lanacl` `aclitem[]` Права доступа; за подробностями обратитесь к Разделу 5.8.

51.30. `pg_largeobject`

В каталоге `pg_largeobject` содержатся данные, образующие «большие объекты». Большой объект идентифицируется по OID, назначаемому при его создании. Каждый большой объект разделяется на сегменты или «страницы», достаточно небольшие для удобного размещения в строках таблицы `pg_largeobject`. Объём данных на странице определяется как `LOBLKSIZE` (что в настоящее время составляет `BLCKSZ/4`, то есть обычно 2 Кб).

До PostgreSQL 9.0 большие объекты не были связаны с механизмом разрешений. В результате таблица `pg_largeobject` была доступна на чтение для всех и через неё можно было получить OID (и содержимое) всех больших объектов в системе. Теперь это не так; для получения списка OID больших объектов нужно обратиться к `pg_largeobject_metadata`.

Таблица 51.30. Столбцы `pg_largeobject`

Тип столбца Описание
`loid` `oid` (ссылается на `pg_largeobject_metadata` .`oid`) Идентификатор большого объекта, включающего эту страницу
`pageno` `int4` Номер этой страницы в большом объекте (начиная с нуля)

Тип столбца Описание
`data bytea` Собственно данные, хранящиеся в большом объекте. Их объём не может превышать `LOBLKSIZE`, но может быть меньше.

В каждой строке `pg_largeobject` содержатся данные для одной строки большого объекта, начиная со смещения (`pageno * LOBLKSIZE`) внутри него (в байтах). Эта реализация допускает разреженное хранилище: страницы могут отсутствовать и могут быть короче `LOBLKSIZE`, даже если это не последние страницы объектов. Пропущенные области в большом объекте будут считываться как нулевые.

51.31. pg_largeobject_metadata

В каталоге `pg_largeobject_metadata` содержатся метаданные, связанные с большими объектами. Собственно данные больших объектов хранятся в `pg_largeobject`.

Таблица 51.31. Столбцы pg_largeobject_metadata

Тип столбца Описание
`oid oid` Идентификатор строки
`lomowner oid` (ссылается на `pg_authid .oid`) Владелец большого объекта
`lomacl aclitem[]` Права доступа; за подробностями обратитесь к Разделу 5.8.

51.32. pg_namespace

В `pg_namespace` сохраняются пространства имён. Пространство имён представляет собой структуру, на которой основываются схемы SQL: в каждом пространстве имён без конфликтов может существовать отдельный набор отношений, типов и т. д.

Таблица 51.32. Столбцы pg_namespace

Тип столбца Описание
`oid oid` Идентификатор строки
`nspname name` Имя пространства имён
`nspowner oid` (ссылается на `pg_authid .oid`) Владелец пространства имён
`nspacl aclitem[]` Права доступа; за подробностями обратитесь к Разделу 5.8.

51.33. pg_opclass

В каталоге `pg_opclass` определяются классы операторов для индексных методов доступа. Каждый класс операторов устанавливает конкретную операцию для индексируемых столбцов определённого типа данных и определённого метода доступа. Класс операторов по сути устанавливает, что некоторое семейство операторов применимо к определённому индексируемому типу столбца.

Набор операторов из семейства, которые действительно можно использовать с индексируемым столбцом, образуют те, что принимают тип данных столбца в качестве левого операнда.

Классы операторов углублённо рассматриваются в Разделе 36.16.

Таблица 51.33. Столбцы `pg_opclass`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`opcmethod` `oid` (ссылается на `pg_am` `.oid`) Индексный метод доступа, для которого создан этот класс операторов
`opcname` `name` Имя этого класса операторов
`opcnamespace` `oid` (ссылается на `pg_namespace` `.oid`) Пространство имён этого класса операторов
`opcowner` `oid` (ссылается на `pg_authid` `.oid`) Владелец класса операторов
`opcfamily` `oid` (ссылается на `pg_opfamily` `.oid`) Семейство операторов, содержащее этот класс операторов
`opcintype` `oid` (ссылается на `pg_type` `.oid`) Тип данных, индексируемый данным классом операторов
`opcdefault` `bool` True, если этот класс операторов применяется по умолчанию для `opcintype`
`opckeytype` `oid` (ссылается на `pg_type` `.oid`) Тип данных, хранимых в индексе, или ноль, если он совпадает с `opcintype`

Значение `opcmethod` класса операторов должно совпадать с `opfmethod` для содержащего его семейства операторов. Кроме того, должно быть не больше одной строки в `pg_opclass`, в которой `opcdefault` равно true для любой данной комбинации `opcmethod` и `opcintype`.

51.34. `pg_operator`

В каталоге `pg_operator` хранится информация об операторах. За дополнительными сведениями обратитесь к описанию CREATE OPERATOR и Разделу 36.14.

Таблица 51.34. Столбцы `pg_operator`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`oprname` `name` Имя оператора
`oprnamespace` `oid` (ссылается на `pg_namespace` `.oid`) OID пространства имён, содержащего этот оператор
`oprowner` `oid` (ссылается на `pg_authid` `.oid`) Владелец оператора
`oprkind` `char` `b` = инфиксный оператор («both») или `l` = префиксный оператор («left»)
`oprcanmerge` `bool`

Тип столбца Описание
Этот оператор поддерживает соединение слиянием
oprcanhash bool Этот оператор поддерживает соединение по хешу
oprleft oid (ссылается на pg_type .oid) Тип левого операнда (ноль для префиксного оператора)
oprright oid (ссылается на pg_type .oid) Тип правого операнда
oprresult oid (ссылается на pg_type .oid) Тип результата (ноль, если это ещё не определённый оператор-пустышка)
oprcom oid (ссылается на pg_operator .oid) Коммутирующий для данного оператора; ноль, если его нет
oprnegate oid (ссылается на pg_operator .oid) Обратный для данного оператора; ноль, если его нет
oprcode regproc (ссылается на pg_proc .oid) Функция, реализующая этот оператор (ноль, если это ещё не определённый оператор-пустышка)
oprrest regproc (ссылается на pg_proc .oid) Функция оценки избирательности ограничения для данного оператора (ноль, если её нет)
oprjoin regproc (ссылается на pg_proc .oid) Функция оценки избирательности соединения для данного оператора (ноль, если её нет)

51.35. pg_opfamily

В каталоге `pg_opfamily` определяются семейства операторов. Каждое семейство операторов представляет собой набор операторов и связанных с ними опорных процедур, реализующих операции, требуемые для определённого индексного метода доступа. Более того, все операторы в семействе являются «совместимыми», в том смысле, который определяется методом доступа. Концепция семейства операторов позволяет применять в индексах межтиповые операторы и рассматривать их, используя знание семантики метода доступа.

Семейства операторов углублённо рассматриваются в Разделе 36.16.

Таблица 51.35. Столбцы pg_opfamily

Тип столбца Описание
oid oid Идентификатор строки
opfmethod oid (ссылается на pg_am .oid) Индексный метод доступа, для которого предназначено семейство операторов
opfname name Имя семейства операторов
opfnamespace oid (ссылается на pg_namespace .oid) Пространство имён семейства операторов
opfowner oid (ссылается на pg_authid .oid) Владелец семейства операторов

Основная часть информации, определяющей семейство операторов, находится не в строке `pg_opfamily`, а в связанных строках в `pg_amop`, `pg_amproc` и `pg_opclass`.

51.36. `pg_parameter_acl`

В каталоге `pg_parameter_acl` представлены параметры конфигурации, для которых назначены права одной или нескольким ролям. Параметры, для которых действуют стандартные права, в этом каталоге отсутствуют.

В отличие от большинства системных каталогов, `pg_parameter_acl` разделяется всеми базами данных кластера: есть только один экземпляр `pg_parameter_acl` в кластере, а не отдельные для каждой базы данных.

Таблица 51.36. Столбцы `pg_parameter_acl`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`parname` `text` Имя параметра конфигурации, для которого были назначены права
`paracl` `aclitem[]` Права доступа; за подробностями обратитесь к Разделу 5.8.

51.37. `pg_partitioned_table`

В каталоге `pg_partitioned_table` хранится информация о секционировании таблиц.

Таблица 51.37. Столбцы `pg_partitioned_table`

Тип столбца Описание
`partrelid` `oid` (ссылается на `pg_class` `.oid`) OID записи в `pg_class` для этой секционированной таблицы
`partstrat` `char` Стратегия секционирования; `h` = секционирование по хешу (Hash), `l` = секционирование по спискам (List), `r` = секционирование по диапазонам (Range)
`partnatts` `int2` Число столбцов в ключе секционирования
`partdefid` `oid` (ссылается на `pg_class` `.oid`) OID записи в `pg_class` для секции по умолчанию в данной секционированной таблице или ноль, если в этой секционированной таблице нет секции по умолчанию
`partattrs` `int2vector` (ссылается на `pg_attribute` `.attnum`) Это массив из `partnatts` значений, указывающих, какие столбцы таблицы входят в ключ секционирования. Например, значения `1 3` будут означать, что ключ секционирования составляют первый и третий столбцы таблицы. Ноль в этом массиве означает, что соответствующей частью ключа секционирования является выражение, а не ссылка на отдельный столбец.
`partclass` `oidvector` (ссылается на `pg_opclass` `.oid`) Для каждого столбца в ключе секционирования этот массив содержит OID применяемых классов операторов. Подробнее это рассматривается в описании `pg_opclass` .
`partcollation` `oidvector` (ссылается на `pg_collation` `.oid`) Для каждого столбца в ключе секционирования этот массив содержит OID правила сортировки для секционирования либо 0, если тип данных этого столбца не сортируемый.
`partexprs` `pg_node_tree` Деревья выражений (в представлении `nodeToString()`) для частей ключа секционирования, не являющихся простыми ссылками на столбцы. Этот список содержит один эле-

Тип столбца Описание
мент для каждого нулевого значения в `partattrs`. Значением может быть NULL, если все части ключа секционирования являются простыми указаниями столбцов.

51.38. pg_policy

В каталоге `pg_policy` хранятся политики защиты на уровне строк для таблиц. Описание политики включает тип команды, к которой она применяется (это могут быть все команды), роли, к которым она применяется, выражение, добавляемое к условию барьера безопасности в запросы, обращающиеся к таблице, и выражение, добавляемое к условию `WITH CHECK` в запросы, которые пытаются добавить в таблицу новые записи.

Таблица 51.38. Столбцы pg_policy

Тип столбца Описание
`oid oid` Идентификатор строки
`polname name` Имя политики.
`polrelid oid` (ссылается на `pg_class .oid`) Таблица, к которой применяется политика
`polcmd char` Тип команды, к которой применяется политика: `r` обозначает SELECT, `a` — INSERT, `w` — UPDATE, `d` — DELETE, а `*` — все команды
`polpermissive bool` Является ли политика разрешительной или ограничительной?
`polroles oid[]` (ссылается на `pg_authid .oid`) Роли, к которым применяется политика; ноль для `PUBLIC` (как правило, будет единственным элементом массива)
`polqual pg_node_tree` Дерево выражения, добавляемое к условиям барьера безопасности в запросы, использующие таблицу
`polwithcheck pg_node_tree` Дерево выражения, добавляемое к условиям WITH CHECK в запросы, которые пытаются добавлять строки в таблицу

Примечание

Политики хранятся в `pg_policy` и применяются только когда для этой таблицы установлено свойство `pg_class.relrowsecurity`.

51.39. pg_proc

В каталоге `pg_proc` хранится информация об обычных функциях, процедурах, агрегатных и оконных функциях (в совокупности также называемых подпрограммами). За дополнительными сведениями обратитесь к описанию CREATE FUNCTION, CREATE PROCEDURE и Разделу 36.3.

Если `prokind` указывает, что данная запись описывает агрегатную функцию, в `pg_aggregate` должна быть соответствующая строка.

Таблица 51.39. Столбцы `pg_proc`

Тип столбца	Описание
`oid` `oid`	Идентификатор строки
`proname` `name`	Имя функции
`pronamespace` `oid` (ссылается на `pg_namespace` `.oid`)	OID пространства имён, содержащего эту функцию
`proowner` `oid` (ссылается на `pg_authid` `.oid`)	Владелец функции
`prolang` `oid` (ссылается на `pg_language` `.oid`)	Язык реализации или интерфейс вызова для этой функции
`procost` `float4`	Примерная стоимость выполнения (в единицах cpu_operator_cost); если установлен признак `proretset`, это стоимость выдачи одной строки
`prorows` `float4`	Примерное число возвращаемых строк (ноль, если признак `proretset` не установлен)
`provariadic` `oid` (ссылается на `pg_type` `.oid`)	Тип данных элементов переменного массива параметров, либо 0, если функция не принимает переменное число параметров
`prosupport` `regproc` (ссылается на `pg_proc` `.oid`)	Вспомогательная функция планировщика для данной функции (см. Раздел 36.11) или ноль, если её нет
`prokind` `char`	`f` — обычная функция (Function), `p` — процедура (Procedure), `a` — агрегатная функция (Aggregate function) или `w` — оконная функция (Window function)
`prosecdef` `bool`	Функция определяет контекст безопасности (т. е. это функция «setuid»)
`proleakproof` `bool`	Функция не имеет побочных эффектов. Никакая информация о её аргументах не выдаётся, кроме как через возвращаемое значение. Любая функция, которая может выдать ошибку, в зависимости от значений аргументов, не является герметичной.
`proisstrict` `bool`	Функция возвращает NULL, если любой из аргументов при вызове NULL. В этом случае функция фактически не будет вызываться вовсе. Функции, не являющиеся «строгими», должны быть готовы принять значения NULL.
`proretset` `bool`	Функция возвращает множество (т. е. множество значений указанного типа данных)
`provolatile` `char`	Свойство `provolatile` говорит, зависит ли результат функции только от её входных аргументов, либо на него влияют внешние факторы. Буквой `i` обозначаются постоянные функции («immutable»), которые всегда возвращают один результат для одних и тех же аргументов. Буквой `s` обозначаются стабильные функции («stable»), результаты которых (для одних и тех же аргументов) не меняются в ходе одного сканирования. Буквой `v` обозначаются изменчивые функции («volatile»), результаты которых могут меняться в любое время. (Так же `v` следует выбирать для функций с побочными эффектами, чтобы система не оптимизировала их вызовы.)
`proparallel` `char`	

Тип столбца	Описание
	Свойство `proparallel` говорит, может ли эта функция безопасно выполняться в параллельном режиме. Символом `s` в нём обозначаются функции, которые могут выполняться в параллельном режиме без ограничений. Символом `r` обозначаются функции, которые могут выполняться в параллельном режиме, но только в ведущем процессе группы; в параллельных рабочих процессах вызывать их нельзя. Символом `u` отмечаются функции небезопасные в параллельном режиме; присутствие такой функции влечёт выбор последовательного плана выполнения запроса.
`pronargs` `int2`	Число входных аргументов
`pronargdefaults` `int2`	Число аргументов, для которых определены значения по умолчанию
`prorettype` `oid` (ссылается на `pg_type` .`oid`)	Тип данных возвращаемого значения
`proargtypes` `oidvector` (ссылается на `pg_type` .`oid`)	Массив типов аргументов функции. В нём учитываются только входные аргументы функции (включая аргументы `INOUT` и `VARIADIC`), так что он представляет сигнатуру вызова функции.
`proallargtypes` `oid[]` (ссылается на `pg_type` .`oid`)	Массив типов аргументов функции. В нём учитываются все аргументы (включая аргументы `OUT` и `INOUT`); однако если все аргументы только входные (`IN`), это поле будет содержать NULL. Заметьте, что индексы в нём начинаются с 1, тогда как в `proargtypes` по историческим причинами они начинаются с 0.
`proargmodes` `char[]`	Массив режимов аргументов функций, закодированных как `i` для входных аргументов (`IN`), `o` для выходных аргументов (`OUT`), `b` для аргументов входных и выходных одновременно (`INOUT`), `v` для переменных аргументов (`VARIADIC`) и `t` для табличных аргументов (`TABLE`). Если все аргументы являются аргументами `IN`, это поле может содержать NULL. Заметьте, что позиции в этом массиве соответствуют позициям в `proallargtypes`, а не в `proargtypes`.
`proargnames` `text[]`	Массив имён аргументов функции. Для аргументов без имени в этом массиве задаются пустые строки. Если все аргументы функции безымянные, это поле может содержать NULL. Заметьте, что позиции в этом массиве соответствуют позициям в `proallargtypes`, а не в `proargtypes`.
`proargdefaults` `pg_node_tree`	Деревья выражений (в представлении `nodeToString()`) для значений аргументов по умолчанию. Это список, содержащий `pronargdefaults` элементов, соответствующих последним *N* *входным* аргументам (т. е., последним *N* позициям в `proargtypes`). Если значение по умолчанию не имеет никакой аргумент, это поле может содержать NULL.
`protrftypes` `oid[]` (ссылается на `pg_type` .`oid`)	Массив типов аргументов/результатов, к которым должны применяться трансформации (заданные в предложении `TRANSFORM` объявления функции), либо NULL, если таковых нет.
`prosrc` `text`	Это значение говорит обработчику функции, как вызывать данную функцию. Это может быть собственно исходный код функции для интерпретируемых языков, объектный символ, имя файла или что-то другое, в зависимости от языка реализации/соглашения о вызовах.
`probin` `text`	Дополнительная информация о том, как вызывать функцию. Интерпретация этого значения так же зависит от языка.

Тип столбца	**Описание**
`prosqlbody` `pg_node_tree`	Предварительно разобранное тело SQL-функции. Используется для функций на языке SQL, когда тело функции оформлено по стандарту SQL, а не в виде строки. В других случаях — NULL.
`proconfig` `text[]`	Локальные присваивания конфигурационных переменных времени выполнения, действующие в рамках функции
`proacl` `aclitem[]`	Права доступа; за подробностями обратитесь к Разделу 5.8.

Для скомпилированных функций, как встроенных, так и динамически загружаемых, поле `prosrc` содержит имя функции на языке C (объектный символ). Для функций на языке SQL: если функция определена в виде строки, поле `prosrc` содержит исходный код функции; если же функция оформлена по стандарту SQL, поле `prosrc` не используется (обычно это пустая строка), а поле `prosqlbody` содержит предварительно разобранное определение функции. Для всех других известных сегодня языков поле `prosrc` содержит исходный код функции. Поле `probin` используется только для динамически загружаемых функций на C, для которых оно задаёт имя разделяемой библиотеки, содержащей эти функции.

51.40. pg_profile

В каталоге `pg_profile` перечисляются все профили, имеющиеся в кластере. Подробнее о профилях рассказывается в CREATE PROFILE.

Профили определяются на уровне кластера, поэтому каталог `pg_profile` является общим для всех баз данных кластера. Данный каталог недоступен на чтение для всех.

Таблица 51.40. Столбцы `pg_profile`

Тип столбца	**Описание**
`oid` `oid`	Идентификатор строки
`pflname` `name`	Имя профиля
`pflfailedloginattempts` `int4`	Значение параметра `FAILED_LOGIN_ATTEMPTS`
`pflfailedauthkeeptime` `int4`	Значение параметра `FAILED_AUTH_KEEP_TIME` (в секундах)
`pfluserinactivetime` `int4`	Значение параметра `USER_INACTIVE_TIME` (в секундах)
`pflpasswordreusetime` `int4`	Значение параметра `PASSWORD_REUSE_TIME` (в секундах)
`pflpasswordreusemax` `int4`	Значение параметра `PASSWORD_REUSE_MAX`
`pflpasswordlifetime` `int4`	Значение параметра `PASSWORD_LIFE_TIME` (в секундах)
`pflpasswordgracetime` `int4`	Значение параметра `PASSWORD_GRACE_TIME` (в секундах)
`pflpasswordminuniqchars` `int4`	Значение параметра `PASSWORD_MIN_UNIQUE_CHARS`

Тип столбца Описание
`pflpasswordminlen` `int4` Значение параметра `PASSWORD_MIN_LEN`
`pflpasswordrequirecomplex` `int4` Значение параметра `PASSWORD_REQUIRE_COMPLEX`

Значения `UNLIMITED` и `DEFAULT` представляются в `pg_profile` числами `-1` и `-2`, соответственно.

51.41. pg_publication

Каталог `pg_publication` содержит все публикации, созданные в базе данных. Подробнее о публикациях можно узнать в Разделе 28.1.

Таблица 51.41. Столбцы `pg_publication`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`pubname` `name` Имя публикации
`pubowner` `oid` (ссылается на `pg_authid` `.oid`) Владелец публикации
`puballtables` `bool` Если true, эта публикация автоматически включает все таблицы в базе данных, в том числе и те, что будут созданы в будущем.
`pubinsert` `bool` Если true, операции INSERT реплицируются для таблиц в публикации.
`pubupdate` `bool` Если true, операции UPDATE реплицируются для таблиц в публикации.
`pubdelete` `bool` Если true, операции DELETE реплицируются для таблиц в публикации.
`pubtruncate` `bool` Если true, операции TRUNCATE реплицируются для таблиц в публикации.
`pubviaroot` `bool` Если true, операции с конечной секцией реплицируются не через эту секцию, а через включённую в публикацию родительскую секцию самого верхнего уровня, с её именем и схемой.

51.42. pg_publication_namespace

Каталог `pg_publication_namespace` содержит сопоставление схем и публикаций в базе данных (это сопоставление вида многие-ко-многим).

Таблица 51.42. Столбцы `pg_publication_namespace`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`pnpubid` `oid` (ссылается на `pg_publication` `.oid`) Ссылка на публикацию
`pnnspid` `oid` (ссылается на `pg_namespace` `.oid`)

Тип столбца **Описание**
Ссылка на схему

51.43. pg_publication_rel

Каталог `pg_publication_rel` содержит сопоставления отношений и публикаций в базе данных (это сопоставления вида многие-ко-многим). Более понятное пользователю представление этой информации можно также получить в Разделе 52.17.

Таблица 51.43. Столбцы `pg_publication_rel`

Тип столбца **Описание**
`oid` `oid` Идентификатор строки
`prpubid` `oid` (ссылается на `pg_publication`.`oid`) Ссылка на публикацию
`prrelid` `oid` (ссылается на `pg_class`.`oid`) Ссылка на отношение
`prqual` `pg_node_tree` Дерево выражения (в представлении `nodeToString()`) для условия, ограничивающего публикацию для отношения, либо NULL, если такое условие отсутствует.
`prattrs` `int2vector` (ссылается на `pg_attribute`.`attnum`) Это массив значений, указывающих, какие столбцы таблицы входят в публикацию. Например, значения `1 3` будут означать, что публикуются первый и третий столбцы таблицы. Значение NULL указывает, что публикуются все столбцы.

51.44. pg_range

В каталоге `pg_range` хранится информация о типах диапазонов. Эта информация дополняет записи типов в `pg_type`.

Таблица 51.44. Столбцы `pg_range`

Тип столбца **Описание**
`rngtypid` `oid` (ссылается на `pg_type`.`oid`) OID типа диапазона
`rngsubtype` `oid` (ссылается на `pg_type`.`oid`) OID типа элемента (подтипа) данного типа диапазона
`rngmultitypid` `oid` (ссылается на `pg_type`.`oid`) OID мультидиапазонного типа для данного диапазонного типа
`rngcollation` `oid` (ссылается на `pg_collation`.`oid`) OID правила сортировки, применяемого для сравнения диапазонов, либо ноль в случае его отсутствия
`rngsubopc` `oid` (ссылается на `pg_opclass`.`oid`) OID класса операторов подтипа, применяемого для сравнения диапазонов
`rngcanonical` `regproc` (ссылается на `pg_proc`.`oid`) OID функции, преобразующей значение диапазона в каноническую форму, либо ноль в случае её отсутствия
`rngsubdiff` `regproc` (ссылается на `pg_proc`.`oid`) OID функции, возвращающей разницу между значениями двух элементов в значении `double precision`, либо ноль в случае её отсутствия

Значение `rngsubopc` (в сочетании с `rngcollation`, если тип элемента сортируемый) определяет порядок сортировки для типа диапазона. Значение `rngcanonical` используется, когда тип элемента дискретный. Значение `rngsubdiff` может отсутствовать, но его рекомендуется задавать для увеличения производительности индексов GiST с диапазонным типом.

51.45. pg_replication_origin

В каталоге `pg_replication_origin` содержатся все созданные источники репликации. Подробно источники репликации описаны в Главе 48.

В отличие от большинства системных каталогов, `pg_replication_origin` разделяется всеми базами данных кластера: есть только один экземпляр `pg_replication_origin` в кластере, а не отдельные в каждой базе.

Таблица 51.45. Столбцы `pg_replication_origin`

Тип столбца Описание
`roident oid` Уникальный в рамках кластера идентификатор для источника репликации. Не должен покидать пределы системы.
`roname text` Определяемое пользователем внешнее имя источника репликации.

51.46. pg_rewrite

В каталоге `pg_rewrite` хранятся правила перезаписи для таблиц и представлений.

Таблица 51.46. Столбцы `pg_rewrite`

Тип столбца Описание
`oid oid` Идентификатор строки
`rulename name` Имя правила
`ev_class oid` (ссылается на `pg_class.oid`) Таблица, к которой относится это правило
`ev_type char` Тип события, которое обрабатывается этим правилом: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE
`ev_enabled char` Устанавливает, в каких режимах session_replication_role срабатывает правило: `O` = правило срабатывает в режимах «origin» (источник) и «local» (локально), `D` = правило отключено, `R` = правило срабатывает в режиме «replica» (реплика), `A` = правило срабатывает всегда.
`is_instead bool` True, если это правило `INSTEAD`
`ev_qual pg_node_tree` Дерево выражения (в форме представления `nodeToString()`) для условия применения правила
`ev_action pg_node_tree` Дерево запроса (в форме представления `nodeToString()`) для действия правила

Примечание

Если для таблицы есть какие-либо правила в этом каталоге, значением `pg_class.relhasrules` для неё должно быть true.

51.47. `pg_role_password`

В каталоге `pg_role_password` ведётся история паролей, что позволяет реализовывать ограничения, связанные с аутентификацией.

В отличие от большинства системных каталогов, `pg_role_password` разделяется всеми базами данных кластера: есть только один экземпляр `pg_role_password` в кластере, а не отдельные в каждой базе данных.

Таблица 51.47. Столбцы `pg_role_password`

Тип столбца Описание
`passsetat` `timestamptz` Время установления пароля
`passrole` `regprofile` (ссылается на `pg_authid` `.oid`) OID роли
`passpassword` `text` Пароль роли (возможно, зашифрованный)

51.48. `pg_seclabel`

В каталоге `pg_seclabel` хранятся метки безопасности для объектов баз данных. Управлять метками безопасности можно с помощью команды `SECURITY LABEL`. Более простой способ просмотра меток безопасности описан в Разделе 52.22.

Также обратите внимание на каталог `pg_shseclabel`, который выполняет ту же функцию для меток безопасности глобальных объектов в кластере баз данных.

Таблица 51.48. Столбцы `pg_seclabel`

Тип столбца Описание
`objoid` `oid` (ссылается на какой-либо столбец OID) OID объекта, к которому относится эта метка безопасности
`classoid` `oid` (ссылается на `pg_class` `.oid`) OID системного каталога, к которому относится этот объект
`objsubid` `int4` Для метки безопасности, связанной со столбцом таблицы, это номер столбца (`objoid` и `classoid` указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
`provider` `text` Поставщик меток безопасности, связанный с этой меткой.
`label` `text` Метка безопасности, применённая к этому объекту.

51.49. `pg_sequence`

В каталоге `pg_sequence` содержится информация о последовательностях. Некоторая информация о последовательностях, в частности имя и схема, хранится в `pg_class`.

Таблица 51.49. Столбцы `pg_sequence`

Тип столбца **Описание**
`seqrelid` `oid` (ссылается на `pg_class`.`oid`) OID записи в `pg_class` для этой последовательности
`seqtypid` `oid` (ссылается на `pg_type`.`oid`) Тип данных последовательности
`seqstart` `int8` Начальное значение последовательности
`seqincrement` `int8` Шаг увеличения последовательности
`seqmax` `int8` Максимальное значение последовательности
`seqmin` `int8` Минимальное значение последовательности
`seqcache` `int8` Размер кеша последовательности
`seqcycle` `bool` Зацикливается ли последовательность

51.50. pg_shdepend

В каталоге `pg_shdepend` записываются отношения зависимости между объектами баз данных и разделяемыми объектами, такими как роли. Эта информация позволяет Postgres Pro удостовериться, что эти объекты не используется, прежде чем удалять их.

Также смотрите каталог `pg_depend`, который играет подобную роль в отношении зависимостей объектов в одной базе данных.

В отличие от большинства системных каталогов, `pg_shdepend` разделяется всеми базами данных кластера: есть только один экземпляр `pg_shdepend` в кластере, а не отдельные в каждой базе данных.

Таблица 51.50. Столбцы `pg_shdepend`

Тип столбца **Описание**
`dbid` `oid` (ссылается на `pg_database`.`oid`) OID базы данных, в которой находится зависимый объект, или ноль, если это глобальный объект
`classid` `oid` (ссылается на `pg_class`.`oid`) OID системного каталога, в котором находится зависимый объект
`objid` `oid` (ссылается на какой-либо столбец OID) OID определённого зависимого объекта
`objsubid` `int4` Для столбца таблицы это номер столбца (`objid` и `classid` указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
`refclassid` `oid` (ссылается на `pg_class`.`oid`) OID системного каталога, к которому относится вышестоящий объект (это должен быть разделяемый каталог)
`refobjid` `oid` (ссылается на какой-либо столбец OID) OID определённого вышестоящего объекта

Тип столбца Описание
`deptype` `char` Код, определяющий конкретную семантику данного отношения зависимости; см. текст

Во всех случаях запись в `pg_shdepend` показывает, что вышестоящий объект нельзя удалить, не удаляя подчинённый объект. Однако есть несколько подвидов зависимости, задаваемых в поле `deptype`:

`SHARED_DEPENDENCY_OWNER` (`o`)

Вышестоящий объект (это должна быть роль) является владельцем зависимого объекта.

`SHARED_DEPENDENCY_ACL` (`a`)

Вышестоящий объект (это должна быть роль) упоминается в ACL (списке управления доступом, то есть списке прав) подчинённого объекта. (Запись `SHARED_DEPENDENCY_ACL` не создаётся для владельца объекта, так как для владельца всё равно имеется запись `SHARED_DEPENDENCY_OWNER`.)

`SHARED_DEPENDENCY_INITACL` (`i`)

Вышестоящий объект (это должна быть роль) упоминается в записи `pg_init_privs` зависимого объекта.

`SHARED_DEPENDENCY_POLICY` (`r`)

Вышестоящий объект (это должна быть роль) упомянут в качестве целевого в объекте зависимой политики.

`SHARED_DEPENDENCY_TABLESPACE` (`t`)

Вышестоящий объект (это должно быть табличное пространство) упомянут в качестве табличного пространства для отношения, которое нигде не хранится.

В будущем могут появиться и другие подвиды зависимости. Заметьте в частности, что с текущим определением вышестоящими объектами могут быть только роли и табличные пространства.

Как и в каталоге `pg_depend`, большинство объектов, созданных во время работы initdb, считаются «закреплёнными». В `pg_shdepend` не добавляются никакие записи, в которых закреплённые объекты являлись бы зависимыми или зависящими.

51.51. pg_shdescription

В каталоге `pg_shdescription` хранятся дополнительные описания (комментарии) для глобальных объектов баз данных. Описания можно задавать с помощью команды `COMMENT` и просматривать в psql, используя команды `\d`.

Также смотрите каталог `pg_description`, который играет подобную роль в отношении описаний объектов в одной базе данных.

В отличие от большинства системных каталогов, `pg_shdescription` разделяется всеми базами данных кластера: есть только один экземпляр `pg_shdescription` в кластере, а не отдельные в каждой базе данных.

Таблица 51.51. Столбцы `pg_shdescription`

Тип столбца Описание
`objoid` `oid` (ссылается на какой-либо столбец OID) OID объекта, к которому относится это описание
`classoid` `oid` (ссылается на `pg_class` `.oid`)

Тип столбца Описание
OID системного каталога, к которому относится этот объект
`description` `text` Произвольный текст, служащий описанием данного объекта

51.52. `pg_shseclabel`

В каталоге `pg_shseclabel` хранятся метки безопасности для глобальных объектов баз данных. Управлять метками безопасности можно с помощью команды `SECURITY LABEL`. Более простой способ просмотра меток безопасности описан в Разделе 52.22.

Также смотрите каталог `pg_seclabel`, который играет подобную роль в отношении меток безопасности объектов в одной базе данных.

В отличие от большинства системных каталогов, `pg_shseclabel` разделяется всеми базами данных кластера: есть только один экземпляр `pg_shseclabel` в кластере, а не отдельные в каждой базе данных.

Таблица 51.52. Столбцы `pg_shseclabel`

Тип столбца Описание
`objoid` `oid` (ссылается на какой-либо столбец OID) OID объекта, к которому относится эта метка безопасности
`classoid` `oid` (ссылается на `pg_class` `.oid`) OID системного каталога, к которому относится этот объект
`provider` `text` Поставщик меток безопасности, связанный с этой меткой.
`label` `text` Метка безопасности, применённая к этому объекту.

51.53. `pg_statistic`

В каталоге `pg_statistic` хранится статистическая информация о содержимом базы данных. Записи в нём создаются командой `ANALYZE`, а затем используются планировщиком запросов. Заметьте, что все эти данные по природе своей неточные, даже если предполагается, что они актуальны.

Обычно для каждого столбца, подлежащего анализу, в этом каталоге есть одна запись со значением `stainherit = false`. Если у таблицы имеются секции или потомки в иерархии наследования, также создаётся вторая запись с `stainherit = true`. Эта строка представляет статистику по столбцу в дереве наследования, то есть статистику по данным, которые возвратит запрос `SELECT` *`столбец`* `FROM` *`таблица`*`*`, тогда как строка с `stainherit = false` представляет результаты запроса `SELECT` *`столбец`* `FROM ONLY` *`таблица`*.

В `pg_statistic` также хранится статистическая информация о значениях выражений индексов. Она описывается так же, как если бы это были столбцы данных; в частности, `starelid` ссылается на индекс. Однако для столбцов, задействуемых в индексе без выражений, дополнительная запись не добавляется, так как она повторяла бы запись для нижележащего столбца таблицы. В настоящее время во всех записях для выражений индексов `stainherit = false`.

Так как для различных типов данных могут быть уместны различные типы статистики, в каталоге `pg_statistic` не делается конкретных предположений о том, какая статистика в нём хранится. Отдельные столбцы в `pg_statistic` выделены только для самых общих свойств (например, доля NULL). Всё остальное хранится в «слотах», представляющих собой группы связанных столбцов, содержимое которых определяется кодовым числом в одном из столбцов слотов.

Каталог `pg_statistic` не должен быть доступен на чтение всем, так как даже статистическая информация о содержимом таблицы может считаться конфиденциальной. (Например, довольно интересны могут быть минимальные и максимальные значения в столбце зарплаты.) Поэтому существует `pg_stats` — доступное всем для чтения представление на базе `pg_statistic`, в котором выдаётся информация только по тем таблицам, которые может читать текущий пользователь.

Таблица 51.53. Столбцы `pg_statistic`

Тип столбца	Описание
`starelid` `oid` (ссылается на `pg_class`.`oid`)	Таблица (или индекс), к которой принадлежит описываемый столбец
`staattnum` `int2` (ссылается на `pg_attribute`.`attnum`)	Номер описываемого столбца
`stainherit` `bool`	Если true, в статистике учитываются значения из дочерних таблиц, а не только в указанном отношении
`stanullfrac` `float4`	Доля записей, в которых этот столбец содержит NULL
`stawidth` `int4`	Средний размер хранения не NULL-элементов, в байтах
`stadistinct` `float4`	Число различных и отличных от NULL значений в столбце. Число больше нуля представляет фактическое количество различных значений. Если это число меньше нуля, его модуль представляет множитель для общего количества строк в таблице; например, для столбца, в котором примерно 80% значений не NULL, и каждое отличное от NULL значение в среднем повторяется дважды, может быть представлено значение `stadistinct` = -0.4. Ноль означает, что количество различных значений неизвестно.
`stakind`*N* `int2`	Кодовое число, определяющее род статистики, хранящейся в *N*-ом «слоте» строки `pg_statistic` row.
`staop`*N* `oid` (ссылается на `pg_operator`.`oid`)	Оператор, с которым была получена статистика, хранящаяся в *N*-ом «слоте». Например, для слота гистограммы это будет оператор <, определяющий порядок сортировки данных. Ноль, если для получения статистики оператор не нужен.
`stacoll`*N* `oid` (ссылается на `pg_collation`.`oid`)	Правило сортировки, с которым была получена статистика, хранящаяся в *N*-ом «слоте». Например, для слота гистограммы, построенной для сортируемого столбца, это будет правило сортировки, определяющее порядок сортировки данных. Для несортируемых данных это поле содержит 0.
`stanumbers`*N* `float4[]`	Численная статистика соответствующего рода для *N*-ного «слота», либо NULL, если с этим родом слота не связаны числовые значения
`stavalues`*N* `anyarray`	Значения столбцов соответствующего рода для *N*-го «слота», либо NULL, если для этого рода слота не хранятся никакие значения. Все значения элементов массива фактически имеют тип данных столбца или связанный тип, например, тип элемента массива, так что определить типы эти столбцов более конкретно, чем `anyarray`, нельзя.

51.54. `pg_statistic_ext`

Каталог `pg_statistic_ext` содержит определения расширенной статистики планировщика. Каждая строка в этом каталоге соответствует *объекту статистики*, созданному командой `CREATE STATISTICS`.

Таблица 51.54. Столбцы `pg_statistic_ext`

Тип столбца **Описание**
`oid` `oid` Идентификатор строки
`stxrelid` `oid` (ссылается на `pg_class` `.oid`) Таблица, содержащая столбцы, описываемые этим объектом
`stxname` `name` Имя объекта статистики
`stxnamespace` `oid` (ссылается на `pg_namespace` `.oid`) OID пространства имён, содержащего этот объект статистики
`stxowner` `oid` (ссылается на `pg_authid` `.oid`) Владелец объекта статистики
`stxkeys` `int2vector` (ссылается на `pg_attribute` `.attnum`) Массив номеров атрибутов, показывающий, какие столбцы таблицы покрываются данным объектом статистики; например, значение `1 3` показывает, что статистика покрывает первый и третий столбцы таблицы
`stxstattarget` `int2` Значение `stxstattarget` управляет детализацией статистики, собираемой для этого объекта командой `ANALYZE`. Нулевое значение указывает, что статистика не должна собираться. При значении NULL используется максимум из ориентиров статистики, заданных для целевых столбцов, а если таковые не заданы, используется системный ориентир статистики по умолчанию. Положительное значение `stxstattarget` определяет ориентировочное количество собираемых «самых частых значений».
`stxkind` `char[]` Массив, содержащий коды для включённых видов статистики; допустимые значения: `d` для статистики по количеству различных значений (n-distinct), `e` для статистики по выражениям (expression statistics), `f` для статистики по функциональным зависимостям (functional dependency) и `m` для списков самых частых значений (MCV)
`stxexprs` `pg_node_tree` Деревья выражений (в представлении `nodeToString()`) для атрибутов объекта статистики, не являющихся простыми ссылками на столбцы. Этот список содержит один элемент для каждого выражения. Значением может быть NULL, если все атрибуты объекта статистики представляют собой простые ссылки.

Поле `pg_statistic_ext` заполняется при выполнении команды `CREATE STATISTICS`, но собственно значения статистики на этом этапе не вычисляются. Статистические данные вычисляются при последующих выполнениях команды `ANALYZE` и сохраняются в соответствующей записи в каталоге `pg_statistic_ext_data`.

51.55. `pg_statistic_ext_data`

Каталог `pg_statistic_ext_data` содержит данные для расширенной статистики планировщика, определённой в `pg_statistic_ext`. Каждая строка в этом каталоге соответствует *объекту статистики*, созданному командой `CREATE STATISTICS`.

Обычно для каждого проанализированного объекта статистики в этом каталоге есть одна запись со значением `stxdinherit` = `false`. Если у таблицы имеются секции или потомки в иерархии наследования, также создаётся вторая запись с `stxdinherit` = `true`. Эта строка представляет объект статистики в дереве наследования, то есть статистику по данным, которые возвратит запрос `SELECT * FROM` *`таблица`*`*`, тогда как строка с `stxdinherit` = `false` представляет результаты запроса `SELECT * FROM ONLY` *`таблица`*.

Как и `pg_statistic`, каталог `pg_statistic_ext_data` не должен быть доступен на чтение для всех, потому что и его содержимое может считаться конфиденциальным. (Например, интерес могут представлять наиболее распространённые сочетания значений столбцов.) Поэтому существует `pg_stats_ext` — доступное всем представление на базе таблицы `pg_statistic_ext_data` (и присоединённой к ней `pg_statistic_ext`), в котором выдаётся информация только по тем таблицам, владельцем которых является текущий пользователь.

Таблица 51.55. Столбцы `pg_statistic_ext_data`

Тип столбца Описание
`stxoid` `oid` (ссылается на `pg_statistic_ext`.`oid`) Объект расширенной статистики, содержащий определение этих данных
`stxdinherit` `bool` Если true, в статистике учитываются значения из дочерних таблиц, а не только в указанном отношении
`stxdndistinct` `pg_ndistinct` Количество различных значений, сериализованное в типе `pg_ndistinct`
`stxddependencies` `pg_dependencies` Статистика по функциональным зависимостям, сериализованная в типе `pg_dependencies`
`stxdmcv` `pg_mcv_list` Статистика по самым частым значениям (Most-Common Values, MCV), сериализованная в типе `pg_mcv_list`
`stxdexpr` `pg_statistic[]` Статистика по каждому выражению, сериализованная в виде массива типа `pg_statistic`

51.56. pg_subscription

В каталоге `pg_subscription` содержатся все существующие подписки логической репликации. Подробнее логическая репликация описана в Главе 28.

В отличие от большинства системных каталогов, `pg_subscription` разделяется всеми базами данных кластера: есть только один экземпляр `pg_subscription` в кластере, а не отдельные в каждой базе данных.

Обычные пользователи не имеют доступа к столбцу `subconninfo`, так как он может содержать пароль в открытом виде.

Таблица 51.56. Столбцы `pg_subscription`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`subdbid` `oid` (ссылается на `pg_database`.`oid`) OID базы данных, в которой располагается эта подписка
`subskiplsn` `pg_lsn` LSN конца транзакции, изменения которой должны быть пропущены, если это действительный LSN; в противном случае `0/0`.
`subname` `name` Имя подписки
`subowner` `oid` (ссылается на `pg_authid`.`oid`) Владелец подписки

Тип столбца **Описание**
`subenabled` `bool` Если true, подписка включена и должна реплицироваться
`subbinary` `bool` Если true, данные для подписки будут запрашиваться с публикующего сервера в двоичном виде
`substream` `char` Управляет потоковой передачей выполняющихся транзакций: `f` = запретить потоковую передачу выполняемых транзакций, `t` = вытеснить изменения выполняемых транзакций на диск и применить сразу после того, как транзакция зафиксирована на публикующем сервере и получена подписчиком, `p` = применить изменения напрямую, используя параллельный рабочий процесс применения, если он доступен (то же, что и `t`, если рабочий процесс недоступен)
`subtwophasestate` `char` Коды состояния для двухфазного режима: `d` = отключено, `p` = ожидается включение, `e` = включено
`subdisableonerr` `bool` Если true, подписка будет отключена в случае обнаружения ошибки одним из её рабочих процессов
`subpasswordrequired` `bool` Если true, подписке требуется указывать пароль для аутентификации
`subrunasowner` `bool` Если true, применение изменений подписки будет выполняться с правами доступа владельца подписки
`subfailover` `bool` При значении true связанные слоты репликации (т. е. основной слот и слоты синхронизации таблиц) в вышестоящей базе данных могут быть синхронизированы с резервными
`subconninfo` `text` Строка подключения к вышестоящей базе данных
`subslotname` `name` Имя слота репликации в вышестоящей базе данных (также применяется в качестве локального имени источника репликации); значение null соответствует имени `NONE`
`subsynccommit` `text` Значение параметра `synchronous_commit` для рабочих процессов подписки
`subpublications` `text[]` Массив имён публикаций, на которые оформлена подписка. Подписки с этими именами определяются в вышестоящей базе данных. Подробнее публикации описаны в Разделе 28.1.
`suborigin` `text` Значение источника должно быть либо `none`, либо `any`. По умолчанию используется `any`. При значении `none` подписка будет запрашивать у публикующего сервера только те изменения, которые не имеют источника. При значении `any` публикующий сервер отправляет изменения независимо от их источника.

51.57. `pg_subscription_rel`

Каталог `pg_subscription_rel` содержит состояние каждого реплицируемого отношения в каждой подписке (это связь вида многие-ко-многим).

Этот каталог содержит только таблицы, известные в подписке после выполнения `CREATE SUBSCRIPTION` или `ALTER SUBSCRIPTION ... REFRESH PUBLICATION`.

Таблица 51.57. Столбцы `pg_subscription_rel`

Тип столбца	Описание
`srsubid oid` (ссылается на `pg_subscription` `.oid`)	Ссылка на подписку
`srrelid oid` (ссылается на `pg_class` `.oid`)	Ссылка на отношение
`srsubstate char`	Код состояния: `i` = инициализация, `d` = копирование данных, `f` = копирование таблицы завершено, `s` = синхронизация выполнена, `r` = готовность (обычная репликация)
`srsublsn pg_lsn`	LSN изменения состояния на удалённой стороне, который используются для координации синхронизации в состояниях `s` или `r`; в других случаях — null

51.58. `pg_tablespace`

В каталоге `pg_tablespace` хранится информация о доступных табличных пространствах. Таблицы могут быть распределены по разным табличным пространствам для эффективного использования дискового хранилища.

В отличие от большинства системных каталогов, `pg_tablespace` разделяется всеми базами данных кластера: есть только один экземпляр `pg_tablespace` в кластере, а не отдельные для каждой базы данных.

Таблица 51.58. Столбцы `pg_tablespace`

Тип столбца	Описание
`oid oid`	Идентификатор строки
`spcname name`	Имя табличного пространства
`spcowner oid` (ссылается на `pg_authid` `.oid`)	Владелец табличного пространства, обычно пользователь, создавший его
`spcacl aclitem[]`	Права доступа; за подробностями обратитесь к Разделу 5.8.
`spcoptions text[]`	Параметры уровня табличного пространства, в виде строк «ключ=значение»

51.59. `pg_transform`

В каталоге `pg_transform` хранятся сведения о трансформациях, которые представляют собой механизм адаптирования типов данных для процедурных языков. За дополнительной информацией обратитесь к CREATE TRANSFORM.

Таблица 51.59. Столбцы `pg_transform`

Тип столбца	Описание
`oid oid`	Идентификатор строки
`trftype oid` (ссылается на `pg_type` `.oid`)	OID типа данных, для которого предназначена трансформация

Тип столбца **Описание**
`trflang` `oid` (ссылается на `pg_language` .`oid`) OID языка, для которого предназначена трансформация
`trffromsql` `regproc` (ссылается на `pg_proc` .`oid`) OID функции, вызываемой для преобразования типа данных, подаваемого на вход процедурному языку (например, в параметрах функции). Ноль, если должно выполняться преобразование по умолчанию.
`trftosql` `regproc` (ссылается на `pg_proc` .`oid`) OID функции, которая будет приводить значение, выдаваемое процедурным языком (например, результаты функций), к данному типу. Ноль, если должно выполняться преобразование по умолчанию.

51.60. pg_trigger

В каталоге `pg_trigger` хранятся триггеры для таблиц и представлений. За дополнительными сведениями обратитесь к описанию CREATE TRIGGER.

Таблица 51.60. Столбцы `pg_trigger`

Тип столбца **Описание**
`oid` `oid` Идентификатор строки
`tgrelid` `oid` (ссылается на `pg_class` .`oid`) Таблица, к которой относится этот триггер
`tgparentid` `oid` (ссылается на `pg_trigger` .`oid`) Родительский триггер, из которого был скопирован данный (это поле заполняется, когда секция создаётся или присоединяется к секционированной таблице); ноль, если данный триггер — не копия
`tgname` `name` Имя триггера (должно быть уникальным среди триггеров одной таблицы)
`tgfoid` `oid` (ссылается на `pg_proc` .`oid`) Вызываемая функция
`tgtype` `int2` Битовая маска, задающая условия срабатывания триггера
`tgenabled` `char` Устанавливает, в каких режимах session_replication_role срабатывает триггер: `O` = триггер срабатывает в режимах «origin» (источник) и «local» (локально), `D` = триггер отключён, `R` = триггер срабатывает в режиме «replica» (реплика), `A` = триггер срабатывает всегда.
`tgisinternal` `bool` True, если триггер создан внутри системы (обычно, для реализации ограничения, заданного в `tgconstraint`)
`tgconstrrelid` `oid` (ссылается на `pg_class` .`oid`) Таблица, задействованная в ограничении ссылочной целостности (ноль, если триггер предназначен не для реализации ограничения ссылочной целостности)
`tgconstrindid` `oid` (ссылается на `pg_class` .`oid`) Индекс, поддерживающий ограничение уникальности, первичного ключа или ссылочной целостности, либо ограничение-исключение (ноль, если триггер предназначен не для реализации таких ограничений)
`tgconstraint` `oid` (ссылается на `pg_constraint` .`oid`)

Тип столбца **Описание**
Запись в `pg_constraint` , связанная с этим триггером (ноль, если триггер предназначен не для реализации ограничения)
`tgdeferrable` `bool` True, если триггер ограничения является откладываемым
`tginitdeferred` `bool` True, если триггер ограничения изначально отложенный
`tgnargs` `int2` Число аргументов, передаваемых функции триггера
`tgattr` `int2vector` (ссылается на `pg_attribute` .`attnum`) Номера столбцов, если триггер привязан к столбцам; в противном случае пустой массив
`tgargs` `bytea` Аргументы строкового типа, передаваемые триггеру, с NULL в конце каждого
`tgqual` `pg_node_tree` Дерево выражения (в представлении `nodeToString()`) для условия триггера `WHEN`, либо NULL, если оно отсутствует
`tgoldtable` `name` Предложение `REFERENCING` для `OLD TABLE` или NULL в случае его отсутствия
`tgnewtable` `name` Предложение `REFERENCING` для `NEW TABLE` или NULL в случае его отсутствия

В настоящее время триггеры, привязанные к столбцам, поддерживаются только для событий `UPDATE`, так что `tgattr` применимо только к событиям такого типа. Поле `tgtype` может содержать биты и для других типов событий, но они распространяются только на таблицы, вне зависимости от значения `tgattr`.

Примечание

Когда `tgconstraint` содержит не ноль, то есть ссылается на запись в `pg_constraint`, поля `tgconstrrelid`, `tgconstrindid`, `tgdeferrable` и `tginitdeferred` по большому счёту избыточны, они повторяют значения в этой записи. Однако возможно связать неоткладываемый триггер с откладываемым ограничением: с ограничениями внешнего ключа могут быть связаны и откладываемые, и неоткладываемые триггеры.

Примечание

Если для отношения есть какие-либо триггеры в этом каталоге, значением `pg_class.relhastriggers` для неё должно быть true.

51.61. `pg_ts_config`

Каталог `pg_ts_config` содержит записи, представляющие конфигурации текстового поиска. Конфигурация задаёт определённый анализатор текстового поиска и список словарей, которые будут использоваться для каждого из фрагментов, выдаваемых анализатором. Анализатор задаётся в записи в `pg_ts_config`, а сопоставления фрагментов со словарями определяются в подчинённых записях в `pg_ts_config_map`.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 51.61. Столбцы `pg_ts_config`

Тип столбца **Описание**
`oid` `oid` Идентификатор строки
`cfgname` `name` Имя конфигурации текстового поиска
`cfgnamespace` `oid` (ссылается на `pg_namespace` `.oid`) OID пространства имён, содержащего эту конфигурацию
`cfgowner` `oid` (ссылается на `pg_authid` `.oid`) Владелец конфигурации
`cfgparser` `oid` (ссылается на `pg_ts_parser` `.oid`) OID анализатора текстового поиска для этой конфигурации

51.62. `pg_ts_config_map`

В каталоге `pg_ts_config_map` содержатся записи, показывающие, к каким словарям текстового поиска и в каком порядке следует обращаться для каждого типа фрагмента, выдаваемого каждым анализатором текстового поиска.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 51.62. Столбцы `pg_ts_config_map`

Тип столбца **Описание**
`mapcfg` `oid` (ссылается на `pg_ts_config` `.oid`) OID записи в `pg_ts_config` , к которой относится эта запись сопоставления
`maptokentype` `int4` Тип фрагмента, выдаваемый анализатором текстового поиска
`mapseqno` `int4` Порядок, в котором нужно просматривать эту запись (меньшие `mapseqno` просматриваются сначала)
`mapdict` `oid` (ссылается на `pg_ts_dict` `.oid`) OID словаря текстового поиска, к которому нужно обратиться

51.63. `pg_ts_dict`

В каталоге `pg_ts_dict` содержатся записи, определяющие словари текстового поиска. Словарь зависит от шаблона текстового поиска, в котором задаются все требуемые функции реализации; сам словарь предоставляет значения для настраиваемых параметров, поддерживаемых шаблонов. Такое разделение позволяет создавать словари непривилегированным пользователям. Параметры задаются текстовой строкой `dictinitoption`, их формат и значение зависят от шаблона.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 51.63. Столбцы `pg_ts_dict`

Тип столбца **Описание**
`oid` `oid` Идентификатор строки
`dictname` `name` Имя словаря текстового поиска
`dictnamespace` `oid` (ссылается на `pg_namespace` `.oid`)

Тип столбца Описание
OID пространства имён, содержащего этот словарь
`dictowner` `oid` (ссылается на `pg_authid` .`oid`) Владелец словаря
`dicttemplate` `oid` (ссылается на `pg_ts_template` .`oid`) OID шаблона текстового поиска для этого словаря
`dictinitoption` `text` Строка с параметрами инициализации для шаблона

51.64. pg_ts_parser

В каталоге `pg_ts_parser` содержатся записи, определяющие анализаторы текстового поиска. Анализатор отвечает за разделение входного текста на лексемы и назначение типа фрагмента каждой лексеме. Так как анализатор должен быть реализован в функции на языке уровня C, создавать новые анализаторы разрешено только суперпользователям баз данных.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 51.64. Столбцы `pg_ts_parser`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`prsname` `name` Имя анализатора текстового поиска
`prsnamespace` `oid` (ссылается на `pg_namespace` .`oid`) OID пространства имён, содержащего этот анализатор
`prsstart` `regproc` (ссылается на `pg_proc` .`oid`) OID функции запуска анализатора
`prstoken` `regproc` (ссылается на `pg_proc` .`oid`) OID функции анализатора, выдающей следующий фрагмент
`prsend` `regproc` (ссылается на `pg_proc` .`oid`) OID функции анализатора, оканчивающей разбор
`prsheadline` `regproc` (ссылается на `pg_proc` .`oid`) OID функции анализатора, выдающей выдержки (ноль, если функции нет)
`prslextype` `regproc` (ссылается на `pg_proc` .`oid`) OID функции анализатора лексических типов

51.65. pg_ts_template

В каталоге `pg_ts_template` содержатся записи, определяющие шаблоны текстового поиска. Шаблон представляет собой заготовку для класса словарей текстового поиска. Так как шаблон должен быть реализован в функциях на уровне языка C, создавать новые шаблоны разрешено только суперпользователям базы.

Возможности текстового поиска Postgres Pro углублённо рассматриваются в Главе 12.

Таблица 51.65. Столбцы `pg_ts_template`

Тип столбца Описание
`oid` `oid` Идентификатор строки

Тип столбца **Описание**
`tmplname` `name` Имя шаблона текстового поиска
`tmplnamespace` `oid` (ссылается на `pg_namespace` `.oid`) OID пространства имён, содержащего этот шаблон
`tmplinit` `regproc` (ссылается на `pg_proc` `.oid`) OID функции инициализации шаблона (ноль, если функции нет)
`tmpllexize` `regproc` (ссылается на `pg_proc` `.oid`) OID функции выделения лексем

51.66. pg_type

В каталоге `pg_type` хранится информация о типах данных. Базовые типы и типы-перечисления (скалярные типы) создаются командой `CREATE TYPE`, а домены — командой `CREATE DOMAIN`. При добавлении любой таблицы в базу данных автоматически создаётся составной тип, представляющий структуру строки таблицы. Также возможно создавать составные типы с помощью команды `CREATE TYPE AS`.

Таблица 51.66. Столбцы pg_type

Тип столбца **Описание**
`oid` `oid` Идентификатор строки
`typname` `name` Имя типа данных
`typnamespace` `oid` (ссылается на `pg_namespace` `.oid`) OID пространства имён, содержащего этот тип
`typowner` `oid` (ссылается на `pg_authid` `.oid`) Владелец типа
`typlen` `int2` Для типа фиксированного размера в `typlen` задаётся число байт во внутреннем представлении типа. Но для типов переменной длины, `typlen` будет отрицательным. Значение -1 обозначает тип «varlena» (он содержит машинное слово, определяющее длину), а -2 обозначает строку в стиле C, оканчивающуюся нулём.
`typbyval` `bool` Поле `typbyval` определяет, будут ли внутренние процедуры передавать переменные этого типа по значению или по ссылке. Полю `typbyval` лучше присвоить false, если длина `typlen` не равна 1, 2 или 4 (либо 8, на 64-битных машинах). Типы переменной длины всегда передаются по ссылке. Заметьте, что `typbyval` может быть false, даже если размер типа позволяет передачу по значению.
`typtype` `char` Поле `typtype` принимает значение `b` для базового типа (base), `c` для составного (composite), то есть типа строки таблицы, `d` для домена (domain), `e` для перечисления (enum), `p` для псевдотипа (pseudo-type), `r` для диапазона (range) или `m` для мультидиапазона (multirange). См. также `typrelid` и `typbasetype`.
`typcategory` `char` В поле `typcategory` задаётся произвольная классификация типов данных, на основе которой анализатор запросов может определить, какие неявные приведения будут «предпочитаемыми». См. Таблицу 51.67.
`typispreferred` `bool`

Тип столбца	**Описание**
	True, если этот тип является предпочитаемым целевым типом в своей категории (`typcategory`)
`typisdefined bool`	True, если тип определён, и false, если это тип-заготовка для ещё не определённого типа. Когда значение `typisdefined` — false, можно полагаться только на заданное имя, пространство имён и OID типа.
`typdelim char`	Символ, разделяющий два значения этого типа при разборе вводимого массива. Заметьте, что этот разделитель связывается с типом данных элемента массива, а не с типом самого массива.
`typrelid oid` (ссылается на `pg_class.oid`)	Если это составной тип (см. `typtype`), этот столбец указывает на запись `pg_class`, определяющую соответствующую таблицу. (Для независимого составного типа запись в `pg_class` на самом деле не представляет таблицу, но она всё равно нужна для связывания с записями `pg_attribute` этого типа.) Для не составных типов содержит ноль.
`typsubscript regproc` (ссылается на `pg_proc.oid`)	OID функции обработки индексов элементов или ноль, если этот тип не поддерживает обращение к элементам по индексу. Для «настоящих» типов массивов `typsubscript` = `array_subscript_handler`, но другие типы могут реализовывать обращение к элементам по индексу особым образом.
`typelem oid` (ссылается на `pg_type.oid`)	Если значение `typelem` ненулевое, оно указывает на другую строку в `pg_type`, определяя тип, возвращаемый при обращении по индексу. Если значение `typsubscript` нулевое, то и значение `typelem` должно быть нулевым. Однако в некоторых случаях оно может быть нулевым и когда значение `typsubscript` ненулевое, если обработчик не учитывает `typelem` при определении типа результата. Обратите внимание, что зависимость `typelem` подразумевает физическое включение типа элемента в целевой тип; поэтому возможности изменения DDL типа элемента могут быть ограничены существованием данного типа.
`typarray oid` (ссылается на `pg_type.oid`)	Если значение `typarray` ненулевое, оно указывает на другую запись в `pg_type`, описывающую «настоящий» тип массива, в которой этот тип будет элементом
`typinput regproc` (ссылается на `pg_proc.oid`)	Функция преобразования ввода (из текстового формата)
`typoutput regproc` (ссылается на `pg_proc.oid`)	Функция преобразования вывода (в текстовый формат)
`typreceive regproc` (ссылается на `pg_proc.oid`)	Функция преобразования ввода (из двоичного формата), либо ноль, если её нет
`typsend regproc` (ссылается на `pg_proc.oid`)	Функция преобразования вывода (в двоичный формат), либо ноль, если её нет
`typmodin regproc` (ссылается на `pg_proc.oid`)	Функция ввода модификатора типа, либо ноль, если тип не поддерживает модификаторы
`typmodout regproc` (ссылается на `pg_proc.oid`)	Функция вывода модификатора типа, либо ноль для использования стандартного формата
`typanalyze regproc` (ссылается на `pg_proc.oid`)	Нестандартная функция `ANALYZE`, либо ноль для использования стандартной функции
`typalign char`	

Тип столбца

Описание

Переменная `typalign` определяет выравнивание, требуемое при хранении значения этого типа. Эта величина применяется при хранении на диске, а также для большинства представлений значений внутри Postgres Pro. Когда последовательно хранятся несколько значений, как например в представлении полной строки на диске, дополнительные байты добавляются перед значением этого типа, чтобы оно начиналось с указанной границы. Заданное выравнивание определяет смещение первого элемента последовательности. Возможные значения:

- `c` = выравнивание по символам (`char`), то есть выравнивание не требуется.
- `s` = выравнивание по коротким словам (`short`), 2 байта для большинства машин.
- `i` = выравнивание по целым (`int`), 4 байта для большинства машин.
- `d` = выравнивание по двойным словам (`double`), 8 байт для большинства машин, но не для всех.

`typstorage` `char`

Значение `typstorage` для типов varlena (типов с `typlen` = -1) говорит, готов ли тип для помещения в TOAST, и какова стратегия по умолчанию для хранения атрибутов этого типа. Возможные значения:

- `p` (plain, простое): Значение всегда должно храниться простым образом (этот вариант используется для всех типов постоянной длины).
- `e` (external, внешнее): Значения могут храниться во вторичном отношении «TOAST» (если оно есть, см. `pg_class.reltoastrelid`).
- `m` (main, основное): Значения могут сжиматься и храниться внутри кортежа.
- `x` (extended, расширенное): Значения могут сжиматься и/или перемещаться во вторичное хранилище.

Для типов, которые могут быть помещены в TOAST, обычно применяется стратегия `x`. Заметьте, что значения `m` тоже могут быть помещены во вторичное хранилище, но только в качестве последней меры (в первую очередь в него помещаются значения `e` и `x`).

`typnotnull` `bool`

Поле `typnotnull` представляет ограничение «не NULL» для типа. Применяется только для доменов.

`typbasetype` `oid` (ссылается на `pg_type`.`oid`)

Если это домен (см. `typtype`), то `typbasetype` указывает на тип, на котором он основан. Ноль, если это не домен.

`typtypmod` `int4`

Домены используют `typtypmod` для записи модификатора (`typmod`), применяемого к их базовому типу (-1, если базовый тип не использует `typmod`). Если тип не является доменом, принимает значение -1.

`typndims` `int4`

Значение `typndims` задаёт число размерностей массива для домена, определённого поверх массива (то есть когда `typbasetype` — тип массива). Для типов, отличных от доменов поверх типов массивов, принимает значение 0.

`typcollation` `oid` (ссылается на `pg_collation`.`oid`)

Значение `typcollation` задаёт правило сортировки для типа. Если тип не является сортируемым, оно будет нулевым. У базового типа, поддерживающего правила сортировки, в этом поле будет ненулевое значение, обычно `DEFAULT_COLLATION_OID`. Домен на базе сортируемого типа может иметь другой OID правила сортировки, если оно было изменено для домена.

`typdefaultbin` `pg_node_tree`

Тип столбца Описание
Если поле `typdefaultbin` не NULL, в нём содержится представление выражения по умолчанию для этого типа (совместимое с `nodeToString()` . Это поле используется только для доменов.
`typdefault text` Поле `typdefault` содержит NULL, если с типом не связано значение по умолчанию. Если `typdefaultbin` не NULL, `typdefault` должно содержать понятную человеку версию выражения значения по умолчанию, записанного в `typdefaultbin`. Если `typdefaultbin` содержит NULL, а `typdefault` нет, то в `typdefault` находится внешнее представление значения по умолчанию, которое можно передать функции преобразования ввода и получить константу.
`typacl aclitem[]` Права доступа; за подробностями обратитесь к Разделу 5.8.

Примечание

Для типов фиксированного размера, используемых в системных таблицах, важно, чтобы размер и выравнивание, определённые в `pg_type`, согласовывались с тем, как компилятор располагает этот столбец в структуре, представляющей строку таблицы.

В Таблице 51.67 перечисляются определённые в системе значения `typcategory`. Если этот список будет дополняться в будущем, в него будут добавляться тоже буквы ASCII в верхнем регистре. Все другие символы ASCII зарезервированы для категорий, определяемых пользователями.

Таблица 51.67. Коды `typcategory`

Код	Категория
`A`	Типы массивов
`B`	Логические типы
`C`	Составные типы
`D`	Типы даты/времени
`E`	Типы-перечисления
`G`	Геометрические типы
`I`	Типы, описывающие сетевые адреса
`N`	Числовые типы
`P`	Псевдотипы
`R`	Диапазонные типы
`S`	Строковые типы
`T`	Интервальные типы
`U`	Пользовательские типы
`V`	Типы битовых строк
`X`	Неизвестный тип (`unknown`)
`Z`	Типы для внутреннего пользования

51.67. pg_user_mapping

В каталоге `pg_user_mapping` хранятся сопоставления локальных пользователей с удалёнными. Обычные пользователи не имеют доступа к этому каталогу, они должны использовать представление `pg_user_mappings`.

Таблица 51.68. Столбцы `pg_user_mapping`

Тип столбца Описание
`oid` `oid` Идентификатор строки
`umuser` `oid` (ссылается на `pg_authid`.`oid`) OID сопоставляемой локальной роли, либо ноль, если сопоставление задаётся для всех
`umserver` `oid` (ссылается на `pg_foreign_server`.`oid`) OID стороннего сервера, содержащего это сопоставление
`umoptions` `text[]` Специальные параметры сопоставления пользователей, в виде строк «ключ=значение»

Глава 52. Системные представления

В дополнение к системным каталогам, в Postgres Pro есть набор встроенных представлений. Некоторые системные представления содержат в себе некоторые популярные запросы к системным каталогам, а другие дают доступ к внутреннему состоянию сервера.

Информационная схема (см. Главу 35) содержит другой набор представлений, пересекающихся по функциональности с системными представлениям. Так как информационная схема соответствует стандарту SQL, тогда как описанные здесь представления свойственны только для Postgres Pro, обычно лучше использовать информационную схему, если через неё можно получить всю требуемую информацию.

В Таблице 52.1 перечислены описываемые здесь системные представления. Подробное описание каждого представления следует далее. Есть также дополнительные представления, показывающие накапливаемую статистику; они перечисляются в Таблице 26.2.

52.1. Обзор

В Таблице 52.1 перечислены системные представления. Подробное описание каждого представления следует далее. Кроме явно отмеченных исключений, все описанные здесь представления доступны только для чтения.

Таблица 52.1. Системные представления

Имя представления	Предназначение
`pg_available_extensions`	доступные расширения
`pg_available_extension_versions`	доступные версии расширений
`pg_backend_memory_contexts`	контекст памяти обслуживающего процесса
`pg_config`	параметры конфигурации времени компиляции
`pg_cursors`	открытые курсоры
`pg_file_settings`	сводка содержимого файла конфигурации
`pg_group`	группы пользователей баз данных
`pg_hba_file_rules`	сводка содержимого файла конфигурации аутентификации клиентов
`pg_ident_file_mappings`	сводное содержимое файла конфигурации с сопоставлениями имён пользователей
`pg_indexes`	индексы
`pg_locks`	блокировки, установленные или ожидаемые в данный момент
`pg_matviews`	материализованные представления
`pg_policies`	policies
`pg_prepared_statements`	подготовленные операторы
`pg_prepared_xacts`	подготовленные транзакции
`pg_publication_tables`	публикации и информация о связанных с ними таблицах
`pg_replication_origin_status`	информация об источниках репликации, включая данные прогресса репликации
`pg_replication_slots`	информация о слотах репликации
`pg_roles`	роли баз данных
`pg_rules`	правила

Имя представления	Предназначение
pg_seclabels	метки безопасности
pg_sequences	последовательности
pg_settings	значения параметров
pg_shadow	пользователи базы данных
pg_shmem_allocations	блоки, выделенные в общей памяти
pg_stats	статистика планировщика
pg_stats_ext	расширенная статистика планировщика
pg_stats_ext_exprs	расширенная статистика планировщика по выражениям
pg_stats_vacuum_database	статистика по очистке базы данных
pg_stats_vacuum_indexes	статистика по очистке индекса
pg_stats_vacuum_tables	статистика по очистке таблицы
pg_tables	таблицы
pg_timezone_abbrevs	аббревиатуры часовых поясов
pg_timezone_names	имена часовых поясов
pg_user	пользователи базы данных
pg_user_mappings	сопоставления пользователей
pg_views	представления
pg_wait_events	события ожидания

52.2. `pg_available_extensions`

В представлении `pg_available_extensions` перечисляются расширения, доступные для установки. Также обратите внимание на каталог `pg_extension`, в котором перечисляются уже установленные расширения.

Таблица 52.2. Столбцы `pg_available_extensions`

Тип столбца Описание
`name` `name` Имя расширения
`default_version` `text` Имя версии по умолчанию, либо `NULL`, если это имя не определено
`installed_version` `text` Текущая установленная версия расширения, либо `NULL`, если расширение не установлено
`comment` `text` Строка комментария из управляющего файла расширения

Представление `pg_available_extensions` доступно только для чтения.

52.3. `pg_available_extension_versions`

В представлении `pg_available_extension_versions` перечислены определённые версии расширений, доступные для установки. Также обратите внимание на каталог `pg_extension`, в котором перечисляются уже установленные расширения.

Таблица 52.3. Столбцы `pg_available_extension_versions`

Тип столбца Описание
`name name` Имя расширения
`version text` Имя версии
`installed bool` True, если эта версия расширения уже установлена
`superuser bool` True, если устанавливать это расширение разрешено только суперпользователям (но см. `trusted`)
`trusted bool` True, если это расширение может быть установлено обычными пользователями, имеющими необходимые права
`relocatable bool` True, если расширение можно переместить в другую схему
`schema name` Имя схемы, в которую должно быть установлено расширение, либо `NULL`, если оно частично или полностью переместимо
`requires name[]` Имена расширений, требующихся для данного, либо `NULL`, если таких нет
`comment text` Строка комментария из управляющего файла расширения

Представление `pg_available_extension_versions` доступно только для чтения.

52.4. `pg_backend_memory_contexts`

В представлении `pg_backend_memory_contexts` показываются все контексты памяти серверного процесса, обслуживающего текущий сеанс.

Представление `pg_backend_memory_contexts` содержит одну строку для каждого контекста памяти.

Таблица 52.4. Столбцы `pg_backend_memory_contexts`

Тип столбца Описание
`name text` Имя контекста памяти
`ident text` Информация для идентификации контекста памяти. Размер поля ограничен 1024 байтами
`parent text` Имя родителя данного контекста памяти
`level int4` Расстояние от верхнего контекста памяти (TopMemoryContext) в дереве контекстов
`total_bytes int8` Общий объём памяти, выделенный для данного контекста, в байтах
`total_nblocks int8`

Тип столбца Описание
Общее количество блоков памяти, выделенных для данного контекста
`free_bytes int8` Объём свободного пространства в байтах
`free_chunks int8` Общее количество свободных участков
`used_bytes int8` Объём использованного пространства в байтах

По умолчанию представление `pg_backend_memory_contexts` могут читать только суперпользователи или пользователи с правами роли `pg_read_all_stats`.

52.5. pg_config

В представлении `pg_config` описываются конфигурационные параметры времени компиляции для текущей установленной версии Postgres Pro. Оно предназначено, например, для программных средств, которым может потребоваться узнать у Postgres Pro расположение требуемых заголовочных файлов и библиотек. Оно выдаёт ту же базовую информацию, что и клиентская утилита Postgres Pro pg_config.

По умолчанию представление `pg_config` доступно только суперпользователям и только для чтения.

Таблица 52.5. Столбцы pg_config

Тип столбца Описание
`name text` Имя параметра
`setting text` Значение параметра

52.6. pg_cursors

В представлении `pg_cursors` перечисляются курсоры, доступные в данный момент. Курсоры могут быть созданы несколькими способами:

- через оператор `DECLARE` в SQL
- через сообщение Bind в клиент-серверном протоколе, как описано в Подразделе 53.2.3
- через интерфейс программирования сервера (SPI, Server Programming Interface), как описано в Разделе 45.1

В представлении `pg_cursors` показываются курсоры, полученные любым способом. Курсор существует только на протяжении транзакции, в которой он определён, если только он не был объявлен с указанием `WITH HOLD`. Поэтому не удерживаемые курсоры показываются в этом представлении только пока не завершится создавшая их транзакция.

> **Примечание**
>
> Курсоры используются внутри системы для реализации некоторых компонентов Postgres Pro, таких как процедурные языки. Таким образом, в представлении `pg_cursors` могут быть курсоры, которые пользователи не создавали явно.

Таблица 52.6. Столбцы `pg_cursors`

Тип столбца	Описание
`name text`	Имя курсора
`statement text`	Дословная строка запроса, создавшего данный курсор
`is_holdable bool`	`True`, если курсор удерживаемый (то есть, к нему можно обращаться после того, как будет зафиксирована транзакция, в которой он объявлен); в противном случае — `false`
`is_binary bool`	`True`, если курсор был объявлен с указанием `BINARY`; в противном случае — `false`
`is_scrollable bool`	`True`, если курсор прокручиваемый (то есть, позволяет получать строки непоследовательным образом); в противном случае — `false`
`creation_time timestamptz`	Время, в которое был объявлен курсор

Представление `pg_cursors` доступно только для чтения.

52.7. `pg_file_settings`

В представлении `pg_file_settings` показывается сводное содержимое файлов конфигурации сервера. Для каждой имеющейся в этих файлах записи «имя = значение» это представление содержит строку с отметкой, показывающей, может ли это значение быть успешно применено. Также это представление может содержать дополнительные строки, говорящие о проблемах, не связанных с записями «имя = значение», например, синтаксических ошибках в этих файлах.

Это представление полезно для проверки, будут ли работать планируемые изменения в файлах конфигурации, или для диагностики возникшей ранее проблемы. Заметьте, что в этом представлении отражается *текущее* содержимое файлов, а не то, что было применено сервером в последний раз. (Чтобы получить то состояние, обычно достаточно обратиться к представлению `pg_settings`.)

По умолчанию представление `pg_file_settings` доступно только суперпользователям и только для чтения.

Таблица 52.7. Столбцы `pg_file_settings`

Тип столбца	Описание
`sourcefile text`	Полный путь и имя файла конфигурации
`sourceline int4`	Номер строки в файле конфигурации, из которой получена эта запись
`seqno int4`	Порядок, в котором обрабатываются записи (1..n)
`name text`	Имя параметра конфигурации
`setting text`	Значение, присваиваемое параметру
`applied bool`	True, если значение может быть применено успешно
`error text`	

Тип столбца Описание
Сообщение об ошибке, говорящее, почему эта запись не может быть применена, либо NULL

Если файл конфигурации содержит синтаксические ошибки или недопустимые имена параметров, сервер не будет пытаться применять никакие параметры из него, так что все поля `applied` будут равны False. В этом случае представление будет содержать одну или несколько строк, в которых поле `error` описывает проблему. Иначе отдельные записи этого файла будут применяться по возможности. Если заданное в некоторой записи присваивание выполнить нельзя (например, из-за неверного значения или если параметр нельзя изменять после запуска сервера), в поле `error` для неё будет записано соответствующее сообщение. Поле `applied` также может содержать False, если данная запись переопределяется последующей записью с тем же именем параметра; это не считается ошибкой, так что поле `error` будет пустым.

Чтобы узнать больше о различных способах изменения параметров времени выполнения, обратитесь к Разделу 18.1.

52.8. pg_group

Представление `pg_group` существует для обратной совместимости: оно эмулирует каталог, существовавший в PostgreSQL до версии 8.1. В нём показываются имена и члены всех ролей без признака `rolcanlogin`, что приблизительно соответствует списку ролей, которые использовались как группы.

Таблица 52.8. Столбцы pg_group

Тип столбца Описание
`groname name` (ссылается на `pg_authid .rolname`) Имя группы
`grosysid oid` (ссылается на `pg_authid .oid`) ID этой группы
`grolist oid[]` (ссылается на `pg_authid .oid`) Массив, содержащий идентификаторы ролей в этой группе

52.9. pg_hba_file_rules

В представлении `pg_hba_file_rules` показывается сводное содержимое файла конфигурации аутентификации клиентов, `pg_hba.conf`. Для каждой непустой и незакомментированной строки в этом файле данное представление содержит одну строку с отметкой, показывающей, может ли это правило быть успешно применено.

Это представление может быть полезно для проверки, будут ли работать планируемые изменения в файле конфигурации аутентификации, или для диагностики возникшей проблемы. Заметьте, что в этом представлении отражается *текущее* содержимое файла, а не то, что было загружено сервером в последний раз.

По умолчанию представление `pg_hba_file_rules` доступно только суперпользователям и только для чтения.

Таблица 52.9. Столбцы pg_hba_file_rules

Тип столбца Описание
`rule_number int4`

Тип столбца **Описание**
Номер правила, если оно рабочее. В противном случае — `NULL`. Таким образом указывается порядок, в котором рассматривается каждое правило во время аутентификации до того, пока не будет найдено соответствие
`file_name text` Имя файла, содержащего это правило
`line_number int4` Номер строки этого правила в `file_name`
`type text` Тип подключения
`database text[]` Список имён баз данных, к которым применяется это правило
`user_name text[]` Список имён пользователей и групп, к которым применяется это правило
`address text` Имя или IP-адрес узла либо одно из значений: `all`, `samehost` или `samenet`, либо NULL для локальных подключений
`netmask text` Маска IP-адреса либо NULL, если это неприменимо
`auth_method text` Метод аутентификации
`options text[]` Параметры, задаваемые для метода аутентификации (если они есть)
`error text` Сообщение об ошибке, говорящее, почему эта строка не может быть обработана, либо NULL

Обычно строка, отражающая некорректную запись, будет содержать значения только в полях `line_number` и `error`.

Чтобы узнать больше о конфигурации аутентификации клиентов, обратитесь к Главе 19.

52.10. `pg_ident_file_mappings`

В представлении `pg_ident_file_mappings` показывается сводное содержимое файла конфигурации, в котором описываются сопоставления имён пользователей, `pg_ident.conf`. Для каждой непустой и незакомментированной строки в этом файле данное представление содержит строку, дополняемую текстом ошибки, если сопоставление не может быть применено.

Это представление может быть полезно для проверки, будут ли работать планируемые изменения в файле конфигурации аутентификации, или для диагностики возникшей проблемы. Заметьте, что в этом представлении отражается *текущее* содержимое файла, а не то, что было загружено сервером в последний раз.

По умолчанию представление `pg_ident_file_mappings` могут читать только суперпользователи.

Таблица 52.10. Столбцы `pg_ident_file_mappings`

Тип столбца **Описание**
`map_number int4`

Тип столбца Описание
Номер этого сопоставления в порядке приоритета, если рабочее; в противном случае — `NULL`
`file_name` `text` Имя файла, содержащего это сопоставление
`line_number` `int4` Номер строки этого сопоставления в `file_name`
`map_name` `text` Имя сопоставления
`sys_name` `text` Полученное имя пользователя клиента
`pg_username` `text` Сопоставленное имя пользователя Postgres Pro
`error` `text` Сообщение об ошибке, говорящее, почему эта строка не может быть обработана, либо `NULL`

Обычно строка, отражающая некорректную запись, будет содержать значения только в полях `line_number` и `error`.

Чтобы узнать больше о конфигурации аутентификации клиентов, обратитесь к Главе 19.

52.11. pg_indexes

Представление `pg_indexes` даёт доступ к полезной информации обо всех индексах в базе данных.

Таблица 52.11. Столбцы pg_indexes

Тип столбца Описание
`schemaname` `name` (ссылается на `pg_namespace` `.nspname`) Имя схемы, содержащей таблицу и индекс
`tablename` `name` (ссылается на `pg_class` `.relname`) Имя таблицы, для которой создан индекс
`indexname` `name` (ссылается на `pg_class` `.relname`) Имя индекса
`tablespace` `name` (ссылается на `pg_tablespace` `.spcname`) Имя табличного пространства, содержащего индекс (NULL, если это пространство по умолчанию)
`indexdef` `text` Определение индекса (реконструированная команда CREATE INDEX)

52.12. pg_locks

Представление `pg_locks` даёт доступ к информации о блокировках, удерживаемых активными процессами на сервере баз данных. Подробнее блокировки рассматриваются в Главе 13.

Представление `pg_locks` содержит одну строку для каждого активного блокируемого объекта, запрошенного режима блокировки и блокирующего процесса. Таким образом, один и тот же блокируемый объект может фигурировать в этом представлении неоднократно, если его блокируют или ожидают блокировки несколько процессов. Однако объекты, свободные от блокировок, в этом представлении отсутствуют вовсе.

Существует несколько различных типов блокируемых объектов: отношения целиком (например, таблицы), отдельные страницы отношений, отдельные кортежи отношений, идентификаторы транзакций (виртуальные и постоянные) и произвольные объекты баз данных (идентифицируемые по OID класса и OID объекта, так же как в `pg_description` или `pg_depend`). Кроме того, в виде отдельного блокируемого объекта представлено право расширения отношения, как и право изменения значения `pg_database.datfrozenxid`. Также могут быть установлены «рекомендательные» блокировки, не имеющие предопределённого значения.

Таблица 52.12. Столбцы `pg_locks`

Тип столбца **Описание**
`locktype text` Тип блокируемого объекта: `relation` (отношение), `extend` (расширение отношения), `frozenid` (замороженный идентификатор), `page` (страница), `tuple` (кортеж), `transactionid` (идентификатор транзакции), `virtualxid` (виртуальный идентификатор), `spectoken` (спекулятивный маркер), `object` (объект), `userlock` (пользовательская блокировка), `advisory` (рекомендательная) или `applytransaction` (применение транзакции). (См. также Таблицу 26.11.)
`database oid` (ссылается на `pg_database.oid`) OID базы данных, к которой относится цель блокировки, ноль, если это разделяемый объект, либо NULL, если целью является идентификатор транзакции
`relation oid` (ссылается на `pg_class.oid`) OID отношения, являющегося целью блокировки, либо NULL, если цель блокировки — не отношение или часть отношения
`page int4` Номер страницы в отношении, являющейся целью блокировки, либо NULL, если цель блокировки — не страница или кортеж отношения
`tuple int2` Номер кортежа на странице, являющегося целью блокировки, либо NULL, если цель блокировки — не кортеж
`virtualxid text` Виртуальный идентификатор транзакции, являющийся целью блокировки, либо `NULL`, если цель блокировки — другой объект; см. Главу 64
`transactionid xid` Идентификатор транзакции, являющийся целью блокировки, либо `NULL`, если цель блокировки — другой объект; см. Главу 64
`classid oid` (ссылается на `pg_class.oid`) OID системного каталога, содержащего цель блокировки, либо NULL, если цель блокировки — не обычный объект базы данных
`objid oid` (ссылается на какой-либо столбец OID) OID цели блокировки в соответствующем системном каталоге, либо NULL, если цель блокировки — не обычный объект базы данных
`objsubid int2` Номер столбца, являющегося целью блокировки (на саму таблицу указывают `classid` и `objid`), ноль, если это некоторый другой обычный объект базы данных, либо NULL, если цель не обычный объект
`virtualtransaction text` Виртуальный идентификатор транзакции, удерживающей или ожидающей блокировку
`pid int4` Идентификатор серверного процесса (PID, Process ID), удерживающего или ожидающего эту блокировку, либо NULL, если блокировка удерживается подготовленной транзакцией

Тип столбца **Описание**
`mode` `text` Название режима блокировки, которая удерживается или запрашивается этим процессом (см. Подраздел 13.3.1 и Подраздел 13.2.3)
`granted` `bool` True, если блокировка получена, и false, если она ожидается
`fastpath` `bool` True, если блокировка получена по быстрому пути, и false, если она получена через основную таблицу блокировок
`waitstart` `timestamptz` Время, когда серверный процесс начал ожидать блокировку, или NULL, если блокировка получена. Обратите внимание, что сразу после начала ожидания в этом поле может кратковременно наблюдаться NULL, при том что в `granted` будет `false`.

Признак `granted` устанавливается в строке, представляющей блокировку, удерживаемую указанным процессом. Если он сброшен, этот процесс ждёт блокировки, из чего следует что как минимум один другой процесс удерживает или ожидает блокировку того же объекта в конфликтующем режиме. Ожидающий процесс будет приостановлен до освобождения другой блокировки (или выявления ситуации взаимоблокировки). Один процесс в один момент времени может ожидать получения максимум одной блокировки.

На протяжении транзакции серверный процесс удерживает исключительную блокировку виртуального идентификатора транзакции. Если транзакции назначается постоянный идентификатор (что обычно происходит, только если транзакция изменяет состояние базы данных), он также удерживает до её завершения блокировку этого постоянного идентификатора. Когда процесс находит необходимым ожидать именно какую-то другую транзакцию, он делает это, запрашивая разделяемую блокировку для идентификатора этой транзакции (виртуального или постоянного, в зависимости от ситуации). Этот запрос будет выполнен, только когда другая транзакция завершится и освободит свои блокировки.

Хотя кортежи тоже представляют собой блокируемый объект, информация о блокировках строк хранится на диске, а не в памяти, поэтому такие блокировки обычно не показываются в этом представлении. Если процесс ожидает блокировки на уровне строки, он обычно виден в нём как ожидающий постоянного идентификатора транзакции текущего владельца этой блокировки.

Блокировка спекулятивного добавления состоит из идентификатора транзакции и маркера спекулятивного добавления. Маркер спекулятивного добавления отображается в столбце `objid`.

Рекомендательные блокировки могут быть получены по ключам, состоящим из одного значения `bigint` или из двух значений integer. Старшая половина `bigint` выводится в столбце `classid`, а младшая половина в столбце `objid`, и `objsubid` равен 1. Исходное значение `bigint` может быть восстановлено выражением `(classid::bigint << 32) | objid::bigint`. Для ключей integer первая часть ключа находится в `classid`, а вторая часть в `objid`, и `objsubid` равна 2. Конкретное предназначение этих ключей определяет пользователь. Рекомендательные блокировки существуют в рамках базы данных, поэтому столбец `database` имеет значение для таких блокировок.

В параллельном режиме используются блокировки применения транзакции при логической репликации. Идентификатор удалённой транзакции отображается в столбце `transactionid`. Столбец `objsubid` показывает подтип блокировки: 0 — блокировка для синхронизации набора изменений, 1 — блокировка для ожидания завершения транзакции, чтобы обеспечить порядок фиксации.

Представление `pg_locks` даёт общую информацию по всем блокировкам в кластере баз данных, а не только по тем, что относятся к текущей базе. Хотя соединив `relation` с `pg_class.oid`, можно получить заблокированные отношения, это будет работать корректно только для отношений в текущей базе данных (для тех, в блокировках которых столбец `database` содержит OID текущей базы данных или ноль).

Соединив столбец `pid` со столбцом `pid` представления `pg_stat_activity`, можно получить дополнительную информацию о сеансах, удерживающих или ожидающих каждую блокировку, например так:

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
    ON pl.pid = psa.pid;
```

Также, если вы используете подготовленные транзакции, столбец `virtualtransaction` можно соединить со столбцом `transaction` представления `pg_prepared_xacts` для получения дополнительной информации о подготовленных транзакциях, удерживающих блокировки. (Подготовленная транзакция не может ожидать блокировок, но она может продолжать удерживать блокировки, полученные ей в процессе выполнения.) Например:

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
    ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

Хотя в принципе возможно получить информацию о процессах, которые блокируют другие процессы, соединив представление `pg_locks` с ним же, очень трудно сделать это правильно во всех деталях. В частности потому, что такой запрос должен будет знать, какие режимы блокировки конфликтуют с другими. Мало того, представление `pg_locks` не показывает, какие процессы стоят перед какими в очередях ожидания блокировок, а также какие процессы являются параллельными рабочими процессами и к каким клиентским сеансам они относятся. Чтобы узнать, каким процессом или процессами блокируется ожидающий процесс, лучше использовать функцию `pg_blocking_pids()` (см. Таблицу 9.69).

В представлении `pg_locks` показываются данные и из менеджера обычных блокировок, и из менеджера предикатных блокировок, которые являются отдельными механизмами; кроме того, менеджер обычных блокировок подразделяет свои блокировки на обычные и полученные *быстрым путём*. Абсолютная согласованность всех этих данных не гарантируется. При обращении к этому представлению данные блокировок по быстрому пути (с `fastpath` = `true`) собираются по очереди с каждого серверного процесса, без замораживания состояния всего менеджера блокировок, так что существует возможность, что в процессе сбора этой информации блокировки будут освобождены или получены. Заметьте однако, что эти блокировки не должны конфликтовать с любыми другими актуальными блокировками. После того как от всех процессов получены блокировки по быстрому пути, менеджер обычных блокировок замораживается целиком и информация обо всех оставшихся блокировках собирается в атомарной операции. После размораживания этого менеджера, также замораживается менеджер предикатных блокировок, и информация об этих блокировках собирается атомарно. Таким образом, за исключением блокировок по быстрому пути, каждый менеджер блокировок выдаёт согласованный набор результатов, но так как мы не блокируем оба этих менеджера одновременно, блокировки могут быть получены или освобождены после того, как опрашивается менеджер обычных блокировок, и до того, как опрашивается менеджер предикатных блокировок.

Блокировка менеджера обычных или предикатных блокировок может отразиться на производительности базы данных, если обращаться к этому представлению часто. Эта блокировка удерживается не дольше, чем необходимо для получения данных от менеджеров, но это не исключает возможность снижения производительности.

52.13. pg_matviews

Представление `pg_matviews` даёт доступ к полезной информации обо всех материализованных представлениях в базе данных.

Таблица 52.13. Столбцы `pg_matviews`

Тип столбца Описание
`schemaname` `name` (ссылается на `pg_namespace`.`nspname`) Имя схемы, содержащей материализованное представление
`matviewname` `name` (ссылается на `pg_class`.`relname`)

Тип столбца	**Описание**
	Имя материализованного представления
`matviewowner name` (ссылается на `pg_authid` .`rolname`)	Имя владельца материализованного представления
`tablespace name` (ссылается на `pg_tablespace` .`spcname`)	Имя табличного пространства, содержащего материализованное представление (NULL, если это пространство по умолчанию)
`hasindexes bool`	True, если материализованное представление имеет (или недавно имело) индексы
`ispopulated bool`	True, если материализованное представление в данный момент наполнено
`definition text`	Определение материализованного представления (реконструированный запрос SELECT)

52.14. pg_policies

Представление `pg_policies` даёт доступ к полезной информации обо всех политиках защиты на уровне строк в базе данных.

Таблица 52.14. Столбцы `pg_policies`

Тип столбца	**Описание**
`schemaname name` (ссылается на `pg_namespace` .`nspname`)	Имя схемы, содержащей таблицу с этой политикой
`tablename name` (ссылается на `pg_class` .`relname`)	Имя таблицы с этой политикой
`policyname name` (ссылается на `pg_policy` .`polname`)	Имя политики
`permissive text`	Является ли политика разрешительной или ограничительной?
`roles name[]`	Роли, к которым применяется политика
`cmd text`	Тип команды, к которому применяется политика
`qual text`	Выражение, добавляемое к условиям барьера безопасности в запросы, к которым применяется политика
`with_check text`	Выражение, добавляемое к условиям WITH CHECK в запросы, которые пытаются добавлять строки в таблицу

52.15. pg_prepared_statements

В представлении `pg_prepared_statements` отображаются все подготовленные операторы, существующие в текущем сеансе. За дополнительными сведениями о подготовленных операторах обратитесь к PREPARE.

Представление `pg_prepared_statements` содержит отдельную строку для каждого подготовленного оператора. Строки добавляются в него, когда создаётся новый подготовленный оператор, и удаляются, когда подготовленный оператор освобождается (например командой `DEALLOCATE`).

Таблица 52.15. Столбцы `pg_prepared_statements`

Тип столбца	Описание
`name` `text`	Идентификатор подготовленного оператора
`statement` `text`	Строка запроса, переданного клиентом и создавшего этот подготовленный оператор. Для подготовленных операторов, создаваемых через SQL, это оператор `PREPARE`, переданный клиентом. Для подготовленных операторов, созданных через клиент-серверный протокол, этот текст представляет сам подготовленный оператор.
`prepare_time` `timestamptz`	Время, в которое был создан подготовленный оператор
`parameter_types` `regtype[]`	Ожидаемые типы параметров для подготовленного оператора в форме массива `regtype`. OID, соответствующий элементу этого массива, может быть получен в результате приведения значения `regtype` к `oid`.
`result_types` `regtype[]`	Типы столбцов, возвращаемых подготовленным оператором в форме массива `regtype`. OID, соответствующий элементу этого массива, может быть получен в результате приведения значения `regtype` к `oid`. Если подготовленный оператор не выдаёт результат (как, например, оператор DML), это поле может содержать NULL.
`from_sql` `bool`	Значение `true`, если подготовленный оператор был создан SQL-командой `PREPARE`; `false`, если оператор был подготовлен через клиент-серверный протокол
`generic_plans` `int8`	Сколько раз был выбран общий план
`custom_plans` `int8`	Сколько раз был выбран специализированный план

Представление `pg_prepared_statements` доступно только для чтения.

52.16. `pg_prepared_xacts`

Представление `pg_prepared_xacts` содержит информацию о транзакциях, которые в настоящее время подготовлены для двухфазной фиксации (за подробностями обратитесь к PREPARE TRANSACTION).

Представление `pg_prepared_xacts` содержит отдельную запись для каждой подготовленной транзакции. Эта запись удаляется, когда транзакция фиксируется или откатывается.

Таблица 52.16. Столбцы `pg_prepared_xacts`

Тип столбца	Описание
`transaction` `xid`	Числовой идентификатор подготовленной транзакции
`gid` `text`	Глобальный идентификатор транзакции, назначаемый транзакции
`prepared` `timestamptz`	Время, в которое транзакция была подготовлена для фиксации
`owner` `name` (ссылается на `pg_authid` `.rolname`)	Имя пользователя, выполнявшего транзакцию
`database` `name` (ссылается на `pg_database` `.datname`)	

Тип столбца Описание
Имя базы данных, в которой выполнялась транзакция

Когда запрашивается представление `pg_prepared_xacts`, внутренние структуры данных менеджера транзакций на мгновение блокируются и создаётся их копия для вывода через это представление. Это гарантирует, что представление выдаёт согласованный набор результатов, при этом не задерживая обычные операции на более продолжительное время, чем необходимо. Тем не менее это может отрицательно сказаться на производительности базы данных при частых обращениях к представлению.

52.17. pg_publication_tables

Представление `pg_publication_tables` содержит информацию о сопоставлении публикаций с информацией о связанных с ними таблицах. В отличие от нижележащего каталога `pg_publication_rel`, в это представление добавляются публикации вида `FOR ALL TABLES` и `FOR TABLES IN SCHEMA`, и для такой публикации в нём будет присутствовать строка для каждой входящей в неё таблицы.

Таблица 52.17. Столбцы `pg_publication_tables`

Тип столбца Описание
`pubname` `name` (ссылается на `pg_publication`.`pubname`) Имя публикации
`schemaname` `name` (ссылается на `pg_namespace`.`nspname`) Имя схемы, содержащей таблицу
`tablename` `name` (ссылается на `pg_class`.`relname`) Имя таблицы
`attnames` `name[]` (ссылается на `pg_attribute`.`attname`) Имена столбцов таблицы, включённых в публикацию. Если список столбцов таблицы не указан в публикации, перечисляются все её столбцы.
`rowfilter` `text` Выражение для условия, ограничивающего публикацию таблицы

52.18. pg_replication_origin_status

Представление `pg_replication_origin_status` содержит информацию о позиции воспроизведения записей репликации, достигнутой для определённого источника. Подробно источники репликации описаны в Главе 48.

Таблица 52.18. Столбцы `pg_replication_origin_status`

Тип столбца Описание
`local_id` `oid` (ссылается на `pg_replication_origin`.`roident`) внутренний идентификатор узла
`external_id` `text` (ссылается на `pg_replication_origin`.`roname`) внешний идентификатор узла
`remote_lsn` `pg_lsn` LSN исходного узла, до которого были реплицированы данные.
`local_lsn` `pg_lsn` LSN данного узла, с которым был реплицирован `remote_lsn`. Применяется при асинхронной фиксации для сброса на диск записей фиксации до сохранения данных.

52.19. `pg_replication_slots`

Представление `pg_replication_slots` содержит список всех слотов репликации, существующих в данный момент в кластере баз данных, а также их текущее состояние.

За дополнительной информацией о слотах репликации обратитесь к Подразделу 25.2.6 и Главе 47.

Таблица 52.19. Столбцы `pg_replication_slots`

Тип столбца	**Описание**
`slot_name name`	Уникальный в рамках кластера идентификатор для слота репликации
`plugin name`	Базовое имя разделяемого объекта, содержащего модуль вывода, который используется этим логическим слотом, либо NULL для физических слотов.
`slot_type text`	Тип слота: `physical` (физический) или `logical` (логический)
`datoid oid` (ссылается на `pg_database.oid`)	OID базы данных, с которой связан этот слот, либо NULL. С базой данных могут быть связаны только логические слоты.
`database name` (ссылается на `pg_database.datname`)	Имя базы данных, с которой связан этот слот, либо NULL. С базой данных могут быть связаны только логические слоты.
`temporary bool`	True, если это временный слот репликации. Временные слоты не сохраняются на диск и автоматически удаляются при ошибке или завершении сеанса.
`active bool`	True, если в данный момент слот используется для передачи данных
`active_pid int4`	ID процесса сеанса, передающего данные для этого слота. `NULL`, если он не используется.
`xmin xid`	Старейшая транзакция, которая должна сохраняться в базе данных для этого слота. `VACUUM` не сможет удалять кортежи, удалённые более поздними транзакциями.
`catalog_xmin xid`	Старейшая транзакция, затрагивающая системные каталоги, которая должна сохраняться в базе данных для этого слота. `VACUUM` не сможет удалять кортежи, удалённые более поздними транзакциями.
`restart_lsn pg_lsn`	Адрес (`LSN`) старейшей записи в WAL, которая по-прежнему может быть нужна пользователям этого слота и поэтому не будет автоматически удаляться во время контрольной точки (при условии соблюдения ограничения max_slot_wal_keep_size). Принимает значение `NULL`, если `LSN` данного слота ещё не резервировался.
`confirmed_flush_lsn pg_lsn`	Адрес (`LSN`), до которого потребитель логического слота подтвердил получение данных. Данные, которые относятся к транзакциям, зафиксированным до этого `LSN`, уже недоступны. Для физических слотов — `NULL`.
`wal_status text`	Доступность файлов WAL, которые требуются для этого слота. Может принимать следующие значения: • `reserved` (резерв) означает, что объём требуемых файлов вписывается в `max_wal_size`.

Тип столбца **Описание**
• `extended` (превышение) означает, что предел `max_wal_size` превышен, но файлы всё ещё сохраняются, потому что вписываются в `wal_keep_size` или нужны слоту репликации. • `unreserved` (нет резерва) означает, что для слота больше не сохраняются требуемые файлы WAL и некоторые из них будут удалены при следующей контрольной точке. Обычно это происходит, если для max_slot_wal_keep_size задано неотрицательное значение. Это состояние может смениться на `reserved` или `extended`. • `lost` (потеря) означает, что данный слот уже нельзя использовать.
`safe_wal_size` `int8` Объём, который может быть записан в WAL, чтобы этот слот не оказался в состоянии «lost». Задаётся в байтах. Для потерянных слотов равен NULL, как и при `max_slot_wal_keep_size` равном `-1`.
`two_phase` `bool` True, если слот используется для декодирования подготовленных транзакций. Для физических слотов всегда false.
`inactive_since` `timestamptz` Время, когда слот перестал быть активным. Если данные по слоту передаются прямо сейчас, отобразится `NULL`. Обратите внимание, что для слотов на резервном сервере, которые синхронизируются с ведущего сервера (для которых поле `synced` содержит значение `true`), столбец `inactive_since` показывает время последней остановки синхронизации (см. Подраздел 47.2.3). `NULL`, если слот всегда синхронизировался.
`conflicting` `bool` True, если этот логический слот конфликтует с восстановлением (и поэтому аннулируется). При значении true в этом столбце обратитесь к столбцу `invalidation_reason`, чтобы узнать причину аннулирования. Для физических слотов всегда `NULL`.
`invalidation_reason` `text` Причина аннулирования слота. Устанавливается как для логических, так и для физических слотов. `NULL`, если слот не аннулирован. Допустимые значения: • `wal_removed` означает, что необходимый файл WAL был удалён. • `rows_removed` означает, что необходимые строки были удалены. Устанавливается только для логических слотов. • `wal_level_insufficient` означает неподходящее значение параметра конфигурации wal_level на ведущем сервере для выполнения логического декодирования. Устанавливается только для логических слотов.
`failover` `bool` True, если для это логического слота включена возможность синхронизации с резервными серверами так, чтобы логическая репликация могла быть возобновлена с нового ведущего сервера после аварийного переключения. Для физических слотов всегда false.
`synced` `bool` True, если этот логический слот был синхронизирован с ведущего сервера. На сервере горячего резерва слоты, для которых столбец `synced` содержит значение true, не могут использоваться для логического декодирования или быть удалены вручную. Для ведущего сервера значение в этом столбце роли не играет и по умолчанию false для всех слотов, однако может быть и true (если ранее этот узел был резервным, и после его повышения до ведущего значение в столбце осталось true).

52.20. `pg_roles`

Представление `pg_roles` открывает доступ к информации о ролях в базах данных. Это просто доступное для всех отображение каталога `pg_authid`, в котором очищено поле пароля.

Таблица 52.20. Столбцы `pg_roles`

Тип столбца **Описание**
`rolname` `name` Имя роли
`rolsuper` `bool` Роли имеет права суперпользователя
`rolinherit` `bool` Роль автоматически наследует права ролей, в которые она включена
`rolcreaterole` `bool` Роль может создавать другие роли
`rolcreatedb` `bool` Роль может создавать базы данных
`rolcanlogin` `bool` Роль может подключаться к серверу. То есть эта роль может быть указана в качестве начального идентификатора авторизации сеанса
`rolreplication` `bool` Роль является ролью репликации. Такие роли могут устанавливать соединения для репликации и создавать/удалять слоты репликации.
`rolconnlimit` `int4` Для ролей, которые могут подключаться к серверу, это значение задаёт максимально разрешённое для этой роли число одновременных подключений. При значении -1 ограничения нет.
`rolprofile` `regprofile` (ссылается на `pg_profile`.`oid`) OID профиля пользователя
`rolloginattempts` `int4` Число следующих подряд неудачных попыток входа пользователя. Это число равно `0`, если параметр `FAILED_LOGIN_ATTEMPTS` соответствующего профиля равен `UNLIMITED`. После успешного входа значение `rolloginattempts` также сбрасывается в `0`.
`rollastlogin` `timestamptz` Время последнего подключения пользователя роли
`rolfirstfailedauth` `timestamptz` Время первой неудачной попытки подключения пользователя роли
`rolstatus` `int2` Состояние роли: `0` — роль активна, `1` — роль заблокирована вручную (см. параметр `ACCOUNT LOCK` в ALTER ROLE), `2` — роль заблокирована из-за бездействия (см. параметр `USER_INACTIVE_TIME` в CREATE PROFILE), `4` — роль заблокирована из-за превышения предельного числа неудачных попыток входа (см. параметр `FAILED_LOGIN_ATTEMPTS` в CREATE PROFILE)
`rolpassword` `text` Не пароль (всегда выводится как `********`)
`rolvaliduntil` `timestamptz` Срок действия пароля (используется только при аутентификации по паролю); NULL, если срок действия не ограничен
`rolpasssetat` `timestamptz` Время установления текущего пароля (используется только при аутентификации по паролю); null, если пароль не установлен.
`rolbypassrls` `bool` Роль не подчиняется никаким политикам защиты на уровне строк; за подробностями обратитесь к Разделу 5.9.

Тип столбца Описание
`rolconfig text[]` Заданные для роли значения по умолчанию переменных времени конфигурации
`oid oid` (ссылается на `pg_authid .oid`) Идентификатор роли

52.21. pg_rules

Представление `pg_rules` открывает доступ к полезной информации о правилах перезаписи запросов.

Таблица 52.21. Столбцы `pg_rules`

Тип столбца Описание
`schemaname name` (ссылается на `pg_namespace .nspname`) Имя схемы, содержащей таблицу
`tablename name` (ссылается на `pg_class .relname`) Имя таблицы, с которой связано это правило
`rulename name` (ссылается на `pg_rewrite .rulename`) Имя правила
`definition text` Определение правила (реконструированная команда создания)

Из представления `pg_rules` исключены правила `ON SELECT` для представлений и материализованных представлений; их можно увидеть в `pg_views` и `pg_matviews`.

52.22. pg_seclabels

Представление `pg_seclabels` содержит информацию о метках безопасности. Это более простая в использовании версия каталога `pg_seclabel`.

Таблица 52.22. Столбцы `pg_seclabels`

Тип столбца Описание
`objoid oid` (ссылается на какой-либо столбец OID) OID объекта, к которому относится эта метка безопасности
`classoid oid` (ссылается на `pg_class .oid`) OID системного каталога, к которому относится этот объект
`objsubid int4` Для метки безопасности, связанной со столбцом таблицы, это номер столбца (`objoid` и `classoid` указывают на саму таблицу). Для всех других типов объектов это поле содержит ноль.
`objtype text` Тип объекта, к которому применяется эта метка, в текстовом виде.
`objnamespace oid` (ссылается на `pg_namespace .oid`) OID пространства имён объекта, если применимо; иначе NULL.
`objname text` Имя объекта, к которому применяется эта метка, в текстовом виде.
`provider text` (ссылается на `pg_seclabel .provider`)

Тип столбца **Описание**
Поставщик меток безопасности, связанный с этой меткой.
`label text` (ссылается на `pg_seclabel .label`) Метка безопасности, применённая к этому объекту.

52.23. pg_sequences

Представление `pg_sequences` даёт доступ к полезной информации обо всех последовательностях в базе данных.

Таблица 52.23. Столбцы pg_sequences

Тип столбца **Описание**
`schemaname name` (ссылается на `pg_namespace .nspname`) Имя схемы, содержащей последовательность
`sequencename name` (ссылается на `pg_class .relname`) Имя последовательности
`sequenceowner name` (ссылается на `pg_authid .rolname`) Имя владельца последовательности
`data_type regtype` (ссылается на `pg_type .oid`) Тип данных последовательности
`start_value int8` Начальное значение последовательности
`min_value int8` Минимальное значение последовательности
`max_value int8` Максимальное значение последовательности
`increment_by int8` Шаг увеличения последовательности
`cycle bool` Зацикливается ли последовательность
`cache_size int8` Размер кеша последовательности
`last_value int8` Последнее значение последовательности, записанное на диск. Если используется кеширование, это значение может быть больше последнего значения, полученного из последовательности.

Столбец `last_value` будет считаться равным NULL, если выполняется любое из следующих условий:

- Чтение из последовательности ещё не выполнялось.
- Текущий пользователь не имеет права `USAGE` или `SELECT` для этой последовательности.
- Последовательность — нежурналируемая, а сервер — резервный.

52.24. pg_settings

Представление `pg_settings` открывает доступ к параметрам времени выполнения сервера. По сути оно представляет собой альтернативный интерфейс для команд `SHOW` и `SET`. Оно также позволяет получить некоторые свойства каждого параметра, которые нельзя получить непосредственно, используя команду `SHOW`, например минимальные и максимальные значения.

Таблица 52.24. Столбцы `pg_settings`

Тип столбца	**Описание**
`name text`	Имя параметра конфигурации времени выполнения
`setting text`	Текущее значение параметра
`unit text`	Неявно подразумеваемая единица измерения параметра
`category text`	Логическая группа параметра
`short_desc text`	Краткое описание параметра
`extra_desc text`	Дополнительное, более подробное, описание параметра
`context text`	Контекст, в котором может задаваться значение параметра (см. ниже)
`vartype text`	Тип параметра (`bool`, `enum`, `integer`, `real` или `string`)
`source text`	Источник текущего значения параметра
`min_val text`	Минимальное допустимое значение параметра (NULL для нечисловых значений)
`max_val text`	Максимально допустимое значение параметра (NULL для нечисловых значений)
`enumvals text[]`	Допустимые значения параметра-перечисления (NULL для значений не перечислений)
`boot_val text`	Значение параметра, устанавливаемое при запуске сервера, если параметр не устанавливается другим образом
`reset_val text`	Значение, к которому будет сбрасывать параметр команда `RESET` в текущем сеансе
`sourcefile text`	Файл конфигурации, в котором было задано текущее значение (NULL для значений, полученных не из файлов конфигурации, или при чтении этого поля не суперпользователем и не пользователем с правами роли `pg_read_all_settings`); полезно при использовании указаний `include` в файлах конфигурации
`sourceline int4`	Номер строки в файле конфигурации, в которой было задано текущее значение (NULL для значений, полученных не из файлов конфигурации, или при чтении этого поля не суперпользователем и не пользователем с правами роли `pg_read_all_settings`).
`pending_restart bool`	`true`, если значение изменено в файле конфигурации, но требуется перезапуск; в противном случае — `false`.

Поле `context` может содержать одно из следующих значений (они перечислены в порядке уменьшения сложности изменения параметров):

`internal`

Эти параметры нельзя изменить непосредственно; они отражают значения, определяемые внутри системы. Некоторые из них можно изменить, пересобрав сервер с другими параметрами конфигурации, либо передав другие аргументы initdb.

`postmaster`

Эти параметры могут быть применены только при запуске сервера, так что любое изменение требует перезапуска сервера. Значения этих параметров обычно задаются в `postgresql.conf`, либо передаются в командной строке при запуске сервера. Разумеется, параметры более низкого уровня `context` также можно задать в момент запуска сервера.

`sighup`

Внесённые в `postgresql.conf` изменения этих параметров можно применить, не перезапуская сервер. Если передать управляющему процессу сигнал SIGHUP, он перечитает `postgresql.conf` и применит изменения. Управляющий процесс также перешлёт сигнал SIGHUP всем своим дочерним процессам, чтобы они тоже приняли новое значение.

`superuser-backend`

Внесённые в `postgresql.conf` изменения этих параметров можно применить без перезапуска сервера; их также можно задать для определённого сеанса в пакете запроса соединения (например, через переменную окружения `PGOPTIONS`, учитываемую библиотекой libpq), но сделать это может только суперпользователь или пользователь с соответствующими правами `SET`. Однако эти параметры никогда не меняются в сеансе, когда он уже начат. Если вы измените их в `postgresql.conf`, отправьте сигнал SIGHUP управляющему процессу, чтобы он перечитал `postgresql.conf`. Новые значения подействуют только на сеансы, запускаемые после этого.

`backend`

Внесённые в `postgresql.conf` изменения этих параметров можно применить без перезапуска сервера; их также можно задать для определённого сеанса в пакете запроса соединения (например, через переменную окружения `PGOPTIONS`, учитываемую библиотекой libpq); это может сделать любой пользователь в своём сеансе. Однако эти параметры никогда не меняются в сеансе, когда он уже начат. Если вы измените их в `postgresql.conf`, отправьте сигнал SIGHUP управляющему процессу, чтобы он перечитал `postgresql.conf`. Новые значения подействуют только на сеансы, запускаемые после этого.

`superuser`

Эти параметры можно изменить в `postgresql.conf`, либо в рамках сеанса, командой `SET`; но только суперпользователи и пользователи с соответствующими правами `SET` могут менять их, используя `SET`. Изменения `postgresql.conf` не повлияют на существующие сеансы, если в них командой `SET` были заданы локальные значения.

`user`

Эти параметры можно задать в `postgresql.conf`, либо в рамках сеанса, командой `SET`. В рамках сеанса изменять их разрешено всем пользователям. Изменения в `postgresql.conf` будут отражены в существующих сеансах, только если в них командой `SET` не были заданы локальные значения.

Чтобы узнать больше о различных способах изменения этих параметров, обратитесь к Разделу 18.1.

Это представление не допускает добавление и удаление строк, но допускает изменение. Команда `UPDATE`, применённая к строке `pg_settings`, равнозначна выполнению команды `SET` для этого параметра. Изменение повлияет только на значение в текущем сеансе. Если `UPDATE` выполняется в транзакции, которая затем прерывается, эффект `UPDATE` пропадает, когда транзакция откатывается. После фиксирования окружающей транзакции этот эффект сохраняется до завершения сеанса, если он не будет переопределён другой командой `UPDATE` или `SET`.

В этом представлении не отображаются внесистемные параметры, если определяющий их модуль расширения не будет загружен к моменту обращения к представлению (например, он может загружаться в shared_preload_libraries или при вызове функции уровня C в данном расширении или при выполнении команды `LOAD`). Так, например, модули архивирования обычно загружаются процессом архиватора, а не обычными сеансами, поэтому в данном представлении не будут отображаться никакие внесистемные параметры, определённые в таких модулях, если специально не загрузить их в обслуживающий процесс, выполняющий запрос.

52.25. pg_shadow

Представление `pg_shadow` существует для обратной совместимости: оно эмулирует каталог, существовавший в PostgreSQL до версии 8.1. В нём показываются свойства всех ролей с признаком `rolcanlogin` в `pg_authid`.

Такое имя («тень») объясняется тем фактом, что эта таблица не должна быть доступна на чтение всем, так как она содержит пароли. Представление `pg_user` является доступным всем отображением `pg_shadow`, в котором очищено поле пароля.

Таблица 52.25. Столбцы pg_shadow

Тип столбца Описание
`usename name` (ссылается на `pg_authid .rolname`) Имя пользователя
`usesysid oid` (ссылается на `pg_authid .oid`) ID этого пользователя
`usecreatedb bool` Пользователь может создавать базы данных
`usesuper bool` Пользователь является суперпользователем
`userepl bool` Пользователь может инициировать потоковую репликацию, включать и отключать режим резервного копирования.
`usebypassrls bool` Пользователь не подчиняется никаким политикам защиты на уровне строк; за подробностями обратитесь к Разделу 5.9.
`passwd text` Зашифрованный пароль; NULL, если он не задан. Подробнее хранение зашифрованных паролей описано в `pg_authid` .
`valuntil timestamptz` Срок действия пароля (используется только при аутентификации по паролю)
`useconfig text[]` Сеансовые значения по умолчанию для переменных конфигурации времени выполнения

52.26. pg_shmem_allocations

В представлении `pg_shmem_allocations` показываются блоки памяти, выделенные в основном сегменте общей памяти сервера. Сюда входят блоки, выделенные с использованием описанных в Подразделе 36.10.10 механизмов как для самого процесса Postgres Pro, так и для расширений.

Заметьте, что в этом представлении не показываются блоки, выделенные с использованием инфраструктуры динамической общей памяти.

Таблица 52.26. Столбцы `pg_shmem_allocations`

Тип столбца	Описание
`name text`	Имя блока в общей памяти. NULL, если этот блок памяти не используется, и `<anonymous>`, если это анонимный блок.
`off int8`	Смещение, с которого начинается выделенный блок. NULL, если это анонимный блок, так как дополнительной информации о таких блоках нет.
`size int8`	Размер блока в байтах
`allocated_size int8`	В размер блока в байтах включается размер выравнивающего дополнения. Для анонимных блоков информация о дополнении недоступна, поэтому значения в столбцах `size` и `allocated_size` всегда равны. Для освобождённой памяти объём дополнения не имеет смысла, так что эти столбцы тоже будут содержать одинаковые значения.

Анонимными считаются блоки, выделенные непосредственно функцией `ShmemAlloc()`, а не функцией `ShmemInitStruct()` или `ShmemInitHash()`.

По умолчанию представление `pg_shmem_allocations` могут читать только суперпользователи или пользователи с правами роли `pg_read_all_stats`.

52.27. `pg_stats`

Представление `pg_stats` открывает доступ к информации, хранящейся в каталоге `pg_statistic`. Это представление даёт доступ только к тем строкам каталога `pg_statistic`, что соответствуют таблицам, которые пользователь может читать; таким образом, это представление можно без опасений разрешить читать всем.

Кроме того, представление `pg_stats` специально разработано для подачи информации в более понятном виде, чем нижележащий каталог — ценой того, что схему представления приходится расширять всякий раз, когда для `pg_statistic` определяются новые типы слотов.

Таблица 52.27. Столбцы `pg_stats`

Тип столбца	Описание
`schemaname name` (ссылается на `pg_namespace.nspname`)	Имя схемы, содержащей таблицу
`tablename name` (ссылается на `pg_class.relname`)	Имя таблицы
`attname name` (ссылается на `pg_attribute.attname`)	Имя столбца, описываемого этой строкой
`inherited bool`	Если true, в данных этой строки учитываются значения в дочерних таблицах, а не только в указанной таблице
`null_frac float4`	Доля записей, в которых этот столбец содержит NULL
`avg_width int4`	Средний размер элементов в столбце, в байтах
`n_distinct float4`	Число больше нуля представляет примерное количество различных значений в столбце. Если это число меньше нуля, его модуль представляет количество различных значений,

Тип столбца **Описание**
делённое на количество строк. (Отрицательная форма применяется, когда `ANALYZE` полагает, что число различных значений, скорее всего, будет расти по мере роста таблицы; положительная, когда в столбце, вероятно, будет фиксированное количество возможных значений.) Например, -1 указывает на столбец с уникальным содержимым, в котором количество различных значений совпадает с количеством строк.
`most_common_vals` `anyarray` Список самых частых значений в столбце. (NULL, если не находятся значения, встречающиеся чаще других.)
`most_common_freqs` `float4[]` Список частот самых частых значений, то есть число их вхождений, делённое на общее количество строк. (NULL, когда `most_common_vals` — NULL.)
`histogram_bounds` `anyarray` Список значений, разделяющих значения столбца на примерно одинаковые популяции. Значения `most_common_vals`, если они присутствуют, не рассматриваются при вычислении этой гистограммы. (Этот столбец содержит NULL, если для типа данных столбца не определён оператор `<`, либо если в `most_common_vals` перечисляется вся популяция.)
`correlation` `float4` Статистическая корреляция между физическим порядком строк и логическим порядком значений столбца. Допустимые значения лежат в диапазоне -1 .. +1. Когда значение около -1 или +1, сканирование индекса по столбцу будет считаться дешевле, чем когда это значение около нуля, как результат уменьшения случайного доступа к диску. (Этот столбец содержит NULL, если для типа данных столбца не определён оператор `<`.)
`most_common_elems` `anyarray` Список элементов, отличных от NULL, наиболее часто присутствующих в значениях столбца. (NULL для скалярных типов.)
`most_common_elem_freqs` `float4[]` Список частот самых частых элементов, то есть доля строк, содержащих минимум один экземпляр данного значения. За частотами по элементам следуют два или три дополнительных значения: минимум и максимум предшествующих частот по элементам и дополнительно частота элементов NULL. (Принимает значение NULL, когда `most_common_elems` — NULL.)
`elem_count_histogram` `float4[]` Гистограмма количеств различных и отличных от NULL элементов в значениях этого столбца, за которой следует среднее количество элементов, отличных от NULL. (Принимает значение NULL для скалярных типов.)
`range_length_histogram` `anyarray` Гистограмма длин непустых и отличных от NULL диапазонов столбцов диапазонных типов. (NULL для недиапазонных типов.) Эта гистограмма вычисляется при помощи функции `subtype_diff` вне зависимости от того, включены границы в диапазон или нет.
`range_empty_frac` `float4` Доля записей, в которых этот столбец содержит пустой диапазон. (NULL для недиапазонных типов.)
`range_bounds_histogram` `anyarray` Гистограмма нижних и верхних границ непустых и отличных от NULL диапазонов. (NULL для недиапазонных типов.) Эти две гистограммы представлены единым массивом диапазонов, чьи нижние границы представляют гистограмму нижних границ, а верхние границы — гистограмму верхних границ.

Максимальным числом записей в полях-массивах можно управлять на уровне столбцов, используя команду `ALTER TABLE SET STATISTICS`, или глобально, задав параметр времени выполнения default_statistics_target.

52.28. `pg_stats_ext`

Представление `pg_stats_ext` открывает доступ к информации о каждом объекте расширенной статистики, объединяя данные из каталогов `pg_statistic_ext` и `pg_statistic_ext_data`. Это представление даёт доступ только к тем строкам `pg_statistic_ext` и `pg_statistic_ext_data`, что соответствуют таблицам, владельцем которых является пользователь: таким образом, это представление можно без опасений разрешить читать всем.

Кроме того, представление `pg_stats_ext` специально разработано для подачи информации в более понятном виде, чем нижележащие каталоги — ценой того, что его схему приходится расширять всякий раз, когда в `pg_statistic_ext` добавляются новые типы расширенной статистики.

Таблица 52.28. Столбцы `pg_stats_ext`

Тип столбца	Описание
`schemaname` `name` (ссылается на `pg_namespace`.`nspname`)	Имя схемы, содержащей таблицу
`tablename` `name` (ссылается на `pg_class`.`relname`)	Имя таблицы
`statistics_schemaname` `name` (ссылается на `pg_namespace`.`nspname`)	Имя схемы, содержащей объект расширенной статистики
`statistics_name` `name` (ссылается на `pg_statistic_ext`.`stxname`)	Имя объекта расширенной статистики
`statistics_owner` `name` (ссылается на `pg_authid`.`rolname`)	Владелец объекта расширенной статистики
`attnames` `name[]` (ссылается на `pg_attribute`.`attname`)	Имена столбцов, включённых в объект расширенной статистики
`exprs` `text[]`	Выражения, включённые в объект расширенной статистики
`kinds` `char[]`	Типы объектов расширенной статистики, включённые для данной записи
`inherited` `bool` (ссылается на `pg_statistic_ext_data`.`stxdinherit`)	Если true, в статистике учитываются значения в дочерних таблицах, а не только в указанном отношении
`n_distinct` `pg_ndistinct`	Количество различных комбинаций значений столбцов. Число больше нуля представляет примерное количество различных скомбинированных значений. Если это число меньше нуля, его модуль представляет количество различных значений, делённое на количество строк. (Отрицательная форма применяется, когда `ANALYZE` полагает, что число различных значений, скорее всего, будет расти по мере роста таблицы; положительная, когда в столбце, вероятно, будет фиксированное количество возможных значений.) Например, -1 указывает на такую комбинацию столбцов, в которой количество различных значений совпадает с количеством строк.
`dependencies` `pg_dependencies`	Статистика по функциональным зависимостям
`most_common_vals` `text[]`	Список самых частых комбинаций значений в столбцах. (NULL, если не находятся комбинации, встречающиеся чаще других.)

Тип столбца **Описание**
`most_common_val_nulls bool[]` Список флагов NULL для самых частых комбинаций значений. (NULL, когда `most_common_vals` — NULL.)
`most_common_freqs float8[]` Список частот самых частых комбинаций, то есть число их вхождений, делённое на общее количество строк. (NULL, когда `most_common_vals` — NULL.)
`most_common_base_freqs float8[]` Список базовых частот самых частых комбинаций, то есть произведение частот отдельных значений. (NULL, когда `most_common_vals` — NULL.)

Максимальным числом записей в полях-массивах можно управлять на уровне столбцов, используя команду `ALTER TABLE SET STATISTICS`, или глобально, задав параметр времени выполнения default_statistics_target.

52.29. `pg_stats_ext_exprs`

Представление `pg_stats_ext_exprs` открывает доступ к информации обо всех выражениях, включённых в объекты расширенной статистики, объединяя данные из каталогов `pg_statistic_ext` и `pg_statistic_ext_data`. Это представление даёт доступ только к тем строкам `pg_statistic_ext` и `pg_statistic_ext_data`, что соответствуют таблицам, владельцем которых является пользователь: таким образом, это представление можно без опасений разрешить читать всем.

Кроме того, представление `pg_stats_ext_exprs` специально разработано для подачи информации в более понятном виде, чем нижележащие каталоги — ценой того, что его схему приходится расширять всякий раз, когда изменяется структура содержимого `pg_statistic_ext`.

Таблица 52.29. Столбцы `pg_stats_ext_exprs`

Тип столбца **Описание**
`schemaname name` (ссылается на `pg_namespace.nspname`) Имя схемы, содержащей таблицу
`tablename name` (ссылается на `pg_class.relname`) Имя таблицы, для которой определён объект статистики
`statistics_schemaname name` (ссылается на `pg_namespace.nspname`) Имя схемы, содержащей объект расширенной статистики
`statistics_name name` (ссылается на `pg_statistic_ext.stxname`) Имя объекта расширенной статистики
`statistics_owner name` (ссылается на `pg_authid.rolname`) Владелец объекта расширенной статистики
`expr text` Выражение, включённое в объект расширенной статистики
`inherited bool` (ссылается на `pg_statistic_ext_data.stxdinherit`) Если true, в статистике учитываются значения в дочерних таблицах, а не только в указанном отношении
`null_frac float4` Доля записей, в которых это выражение содержит NULL
`avg_width int4` Средний размер элементов в выражении, в байтах
`n_distinct float4` Число больше нуля представляет примерное количество различных значений в выражении. Если это число меньше нуля, его модуль представляет количество различных

Тип столбца Описание
значений, делённое на количество строк. (Отрицательная форма применяется, когда `ANALYZE` полагает, что число различных значений, скорее всего, будет расти по мере роста таблицы; положительная, когда в выражении, вероятно, будет фиксированное количество возможных значений.) Например, -1 указывает на выражение с уникальным содержимым, в котором количество различных значений совпадает с количеством строк.
`most_common_vals` `anyarray` Список самых частых значений в выражении. (NULL, если не находятся значения, встречающиеся чаще других.)
`most_common_freqs` `float4[]` Список частот самых частых значений, то есть число их вхождений, делённое на общее количество строк. (NULL, когда `most_common_vals` — NULL.)
`histogram_bounds` `anyarray` Список значений, разделяющих значения выражения на примерно одинаковые популяции. Значения `most_common_vals`, если они присутствуют, не рассматриваются при вычислении этой гистограммы. (Это выражение содержит NULL, если для типа данных выражения не определён оператор `<`, либо если в `most_common_vals` перечисляется вся популяция.)
`correlation` `float4` Статистическая корреляция между физическим порядком строк и логическим порядком значений выражения. Допустимые значения лежат в диапазоне -1 .. +1. Когда значение около -1 или +1, сканирование индекса по выражению будет считаться дешевле, чем когда это значение около нуля, как результат уменьшения случайного доступа к диску. (Это выражение содержит NULL, если для типа данных выражения не определён оператор `<`.)
`most_common_elems` `anyarray` Список элементов, отличных от NULL, наиболее часто присутствующих в значениях выражения. (NULL для скалярных типов.)
`most_common_elem_freqs` `float4[]` Список частот самых частых элементов, то есть доля строк, содержащих минимум один экземпляр данного значения. За частотами по элементам следуют два или три дополнительных значения: минимум и максимум предшествующих частот по элементам и дополнительно частота элементов NULL. (Принимает значение NULL, когда `most_common_elems` — NULL.)
`elem_count_histogram` `float4[]` Гистограмма количеств различных и отличных от NULL элементов в значениях этого выражения, за которой следует среднее количество элементов, отличных от NULL. (Принимает значение NULL для скалярных типов.)

Максимальным числом записей в полях-массивах можно управлять на уровне столбцов, используя команду `ALTER TABLE SET STATISTICS`, или глобально, задав параметр времени выполнения default_statistics_target.

52.30. `pg_stats_vacuum_database`

Представление `pg_stats_vacuum_database` будет содержать одну строку для каждой базы данных в этом кластере, показывая статистику очистки этой базы данных.

Таблица 52.30. Столбцы `pg_stats_vacuum_database`

Тип столбца Описание
`dboid` `oid` OID базы данных

Тип столбца	Описание
`db_blks_read` `int8`	Количество блоков базы данных, прочитанных операциями очистки для этой БД
`db_blks_hit` `int8`	Количество попаданий в буферный кеш блоков базы данных в процессе операций очистки, выполненных для этой БД
`total_blks_dirtied` `int8`	Количество блоков базы данных, загрязнённых операциями очистки для этой БД
`total_blks_written` `int8`	Количество блоков базы данных, записанных операциями очистки для этой БД
`wal_records` `int8`	Общее число записей WAL, сгенерированных операциями очистки для этой базы данных
`wal_fpi` `int8`	Общее число образов полных страниц в WAL, сгенерированных операциями очистки для этой базы данных
`wal_bytes` `numeric`	Общий объём WAL (в байтах), сгенерированный операциями очистки для этой базы данных
`blk_read_time` `float8`	Общее время, затраченное на чтение блоков базы данных операциями очистки, выполненными для этой БД, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`blk_write_time` `float8`	Общее время, затраченное на запись блоков базы данных операциями очистки, выполненными для этой БД, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`delay_time` `float8`	Время простоя в точке задержки при выполнении операций очистки для этой базы данных, в миллисекундах (за подробностями обратитесь к Подразделу 18.4.4).
`system_time` `float8`	Время использования процессора в режиме ядра при очистке этой базы данных, в миллисекундах
`user_time` `float8`	Время использования процессора в пользовательском режиме при очистке этой базы данных, в миллисекундах
`total_time` `float8`	Общее время, затраченное на очистку этой базы данных, в миллисекундах
`interrupts` `int4`	Сколько раз операции очистки для этой базы данных были прерваны из-за каких-либо ошибок

Примечание

При обновлении сервера путём замены исполняемых файлов обновите представление `pg_stats_vacuum_database` от имени суперпользователя, запускавшего `initdb`:

- Для каждой базы данных, включая `template1`, но исключая `template0`, выполните:

```
CREATE OR REPLACE VIEW pg_stats_vacuum_database AS
SELECT
  db.oid as dboid,  stats.db_blks_read,
```

```
    stats.db_blks_hit,
    stats.total_blks_dirtied,
    stats.total_blks_written,  stats.wal_records,
    stats.wal_fpi,
    stats.wal_bytes,  stats.blk_read_time,
    stats.blk_write_time,  stats.delay_time,
    stats.system_time,
    stats.user_time,
    stats.total_time,  stats.interrupts
  FROM
    pg_database db LEFT JOIN pg_stats_vacuum_database(db.oid) stats
  ON
    db.oid = stats.dboid;
```

- Для базы данных `template0` выполните:

```
\c template1
ALTER DATABASE template0 ALLOW_CONNECTIONS on;
\c template0
CREATE OR REPLACE VIEW pg_stats_vacuum_database AS
SELECT
  db.oid as dboid,  stats.db_blks_read,
  stats.db_blks_hit,
  stats.total_blks_dirtied,
  stats.total_blks_written,  stats.wal_records,
  stats.wal_fpi,
  stats.wal_bytes,  stats.blk_read_time,
  stats.blk_write_time,  stats.delay_time,
  stats.system_time,
  stats.user_time,
  stats.total_time,  stats.interrupts
FROM
  pg_database db LEFT JOIN pg_stats_vacuum_database(db.oid) stats
ON
  db.oid = stats.dboid;
\c template1
ALTER DATABASE template0 ALLOW_CONNECTIONS off;
```

52.31. pg_stats_vacuum_indexes

Представление `pg_stats_vacuum_indexes` будет содержать одну строку для каждого индекса в текущей базе данных (включая индексы таблиц TOAST), показывая статистику очистки этого конкретного индекса.

Таблица 52.31. Столбцы `pg_stats_vacuum_indexes`

Тип столбца Описание
`relid` `oid` OID индекса
`schema` `name` Имя схемы, в которой расположен индекс
`relname` `name` Имя этого индекса
`total_blks_read` `int8` Количество блоков БД, прочитанных операциями очистки для этого индекса
`total_blks_hit` `int8`

Тип столбца	**Описание**
	Количество попаданий в буферный кеш блоков базы данных в процессе операций очистки, выполненных для этого индекса
`total_blks_dirtied int8`	Количество блоков базы данных, загрязнённых операциями очистки для этого индекса
`total_blks_written int8`	Количество блоков базы данных, записанных операциями очистки для этого индекса
`rel_blks_read int8`	Количество блоков, прочитанных для этого индекса операциями очистки
`rel_blks_hit int8`	Сколько раз блоки этого индекса уже обнаруживались в буферном кеше операциями очистки, так что чтение с диска не потребовалось (здесь учитываются только случаи обнаружения блоков в кеше Postgres Pro, а не в кеше операционной системы)
`pages_deleted int8`	Количество страниц, которые были удалены операциями очистки, выполненными для этого индекса
`tuples_deleted int8`	Количество «мёртвых» кортежей, удалённых операциями очистки для этого индекса
`wal_records int8`	Общее число записей WAL, сгенерированных операциями очистки для этого индекса
`wal_fpi int8`	Общее число образов полных страниц в WAL, сгенерированных операциями очистки для этого индекса
`wal_bytes numeric`	Общий объём WAL (в байтах), сгенерированный операциями очистки для этого индекса
`blk_read_time int8`	Общее время, затраченное на чтение блоков БД операциями очистки, выполненными для этого индекса, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`blk_write_time int8`	Общее время, затраченное на запись блоков БД операциями очистки, выполненными для этой таблицы, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`delay_time float8`	Время простоя в точке задержки при выполнении операций очистки для этого индекса, в миллисекундах (за подробностями обратитесь к Подразделу 18.4.4).
`system_time float8`	Время использования процессора в режиме ядра при очистке этого индекса, в миллисекундах
`user_time float8`	Время использования процессора в пользовательском режиме при очистке этого индекса, в миллисекундах
`total_time float8`	Общее время, затраченное на очистку этого индекса, в миллисекундах
`interrupts integer`	Сколько раз операции очистки для этого индекса были прерваны из-за каких-либо ошибок

52.32. `pg_stats_vacuum_tables`

Представление `pg_stats_vacuum_tables` будет содержать одну строку для каждой таблицы в текущей базе данных (включая таблицы TOAST), показывая статистику очистки этой конкретной таблицы.

Таблица 52.32. Столбцы `pg_stats_vacuum_tables`

Тип столбца Описание
`relid` `oid` OID таблицы
`schema` `name` Имя схемы, в которой расположена эта таблица
`relname` `name` Имя этой таблицы
`total_blks_read` `int8` Количество блоков БД, прочитанных операциями очистки для этой таблицы
`total_blks_hit` `int8` Количество попаданий в буферный кеш блоков базы данных в процессе операций очистки, выполненных для этой таблицы
`total_blks_dirtied` `int8` Количество блоков БД, загрязнённых операциями очистки для этой таблицы
`total_blks_written` `int8` Количество блоков базы данных, записанных операциями очистки для этой таблицы
`rel_blks_read` `int8` Количество блоков, прочитанных из этой таблицы операциями очистки
`rel_blks_hit` `int8` Сколько раз блоки этой таблицы уже обнаруживались в буферном кеше операциями очистки, так что чтение с диска не потребовалось (здесь учитываются только случаи обнаружения блоков в кеше Postgres Pro, а не в кеше операционной системы)
`pages_scanned` `int8` Количество страниц, к которым обращались операции очистки, выполненные для этой таблицы
`pages_removed` `int8` Количество страниц, удалённых из физического хранилища операциями очистки для этой таблицы
`pages_frozen` `int8` Сколько раз страницы этой таблицы были помечены операциями очистки в карте видимости как полностью замороженные
`pages_all_visible` `int8` Сколько раз страницы этой таблицы были помечены операциями очистки в карте видимости как полностью видимые
`tuples_deleted` `int8` Количество «мёртвых» кортежей, удалённых операциями очистки из этой таблицы
`tuples_frozen` `int8` Количество кортежей этой таблицы, помеченных операциями очистки как замороженные
`dead_tuples` `int8` Количество «мёртвых» кортежей, оставленных операциями очистки в этой таблице из-за видимости этих кортежей в транзакциях
`index_vacuum_count` `int8` Сколько раз была выполнена очистка индексов этой таблицы

Тип столбца **Описание**
`rev_all_frozen_pages` `int8` Сколько раз для страниц данной таблицы снималась метка полной заморозки в карте видимости
`rev_all_visible_pages` `int8` Сколько раз для страниц данной таблицы снималась метка полной видимости в карте видимости
`wal_records` `int8` Общее число записей WAL, сгенерированных операциями очистки для этой таблицы
`wal_fpi` `int8` Общее число образов полных страниц в WAL, сгенерированных операциями очистки для этой таблицы
`wal_bytes` `numeric` Общий объём WAL (в байтах), сгенерированный операциями очистки для этой таблицы
`blk_read_time` `int8` Общее время, затраченное на чтение блоков БД операциями очистки, выполненными для этой таблицы, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`blk_write_time` `int8` Общее время, затраченное на запись блоков БД операциями очистки, выполненными для этой таблицы, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`delay_time` `float8` Время простоя в точке задержки при выполнении операций очистки для этой таблицы, в миллисекундах (за подробностями обратитесь к Подразделу 18.4.4).
`system_time` `float8` Время использования процессора в режиме ядра при очистке этой таблицы, в миллисекундах
`user_time` `float8` Время использования процессора в пользовательском режиме при очистке этой таблицы, в миллисекундах
`total_time` `float8` Общее время, затраченное на очистку этой таблицы, в миллисекундах
`interrupts` `integer` Сколько раз операции очистки для этой таблицы были прерваны из-за каких-либо ошибок

Столбцы `total_*`, `wal_*` и `blk_*` содержат данные об очистке индексов этой таблицы, а столбцы `system_time` и `user_time` содержат только данные об очистке кучи.

52.33. `pg_tables`

Представление `pg_tables` даёт доступ к полезной информации обо всех таблицах в базе данных.

Таблица 52.33. Столбцы `pg_tables`

Тип столбца **Описание**
`schemaname` `name` (ссылается на `pg_namespace` `.nspname`) Имя схемы, содержащей таблицу
`tablename` `name` (ссылается на `pg_class` `.relname`)

Тип столбца Описание
Имя таблицы
`tableowner name` (ссылается на `pg_authid .rolname`) Имя владельца таблицы
`tablespace name` (ссылается на `pg_tablespace .spcname`) Имя табличного пространства, содержащего таблицу (NULL, если это пространство по умолчанию)
`hasindexes bool` (ссылается на `pg_class .relhasindex`) True, если эта таблица имеет (или недавно имела) индексы
`hasrules bool` (ссылается на `pg_class .relhasrules`) True, если для таблицы определены (или были определены) правила
`hastriggers bool` (ссылается на `pg_class .relhastriggers`) True, если для таблицы определены (или были определены) триггеры
`rowsecurity bool` (ссылается на `pg_class .relrowsecurity`) True, если для таблицы включена защита строк

52.34. pg_timezone_abbrevs

Представление `pg_timezone_abbrevs` содержит список аббревиатур часовых поясов, которые в настоящее время распознаются процедурами ввода даты/времени. Содержимое этого представления меняется при изменении параметра времени выполнения timezone_abbreviations.

Таблица 52.34. Столбцы `pg_timezone_abbrevs`

Тип столбца Описание
`abbrev text` Сокращение часового пояса
`utc_offset interval` Смещение от UTC (положительное — к востоку от Гринвича)
`is_dst bool` True, если эта аббревиатура задаёт летнее время

Большинство аббревиатур часовых поясов представляют фиксированные смещения от UTC, но в некоторые поясах смещение менялось в ходе истории (за подробностями обратитесь к Разделу B.4). В таких случаях данное представление отображает текущее значение.

52.35. pg_timezone_names

В представлении `pg_timezone_names` содержится список имён часовых поясов, распознаваемых командой `SET TIMEZONE`, вместе с соответствующими аббревиатурами, смещением UTC и статусом летнего времени. (Говоря технически строго, в Postgres Pro используется не UTC, так как не учитываются секунды координации.) В отличие от аббревиатур, представленных в `pg_timezone_abbrevs`, многие из этих имён подразумевают некоторый набор правил перехода на летнее время. Поэтому сопутствующая информация меняется при пересечении границ перехода на летнее время. Отображаемая информация вычисляется, исходя из текущего значения `CURRENT_TIMESTAMP`.

Таблица 52.35. Столбцы `pg_timezone_names`

Тип столбца Описание
`name text` Название часового пояса

Тип столбца Описание
`abbrev` `text` Сокращение часового пояса
`utc_offset` `interval` Смещение от UTC (положительное — к востоку от Гринвича)
`is_dst` `bool` True, если текущие данные отражают летнее время

52.36. pg_user

Представление `pg_user` открывает доступ к информации о пользователях базы данных. Это просто доступное для всех отображение представления `pg_shadow`, в котором очищено поле пароля.

Таблица 52.36. Столбцы pg_user

Тип столбца Описание
`usename` `name` Имя пользователя
`usesysid` `oid` ID этого пользователя
`usecreatedb` `bool` Пользователь может создавать базы данных
`usesuper` `bool` Пользователь является суперпользователем
`userepl` `bool` Пользователь может инициировать потоковую репликацию, включать и отключать режим резервного копирования.
`usebypassrls` `bool` Пользователь не подчиняется никаким политикам защиты на уровне строк; за подробностями обратитесь к Разделу 5.9.
`passwd` `text` Не пароль (всегда выводится как `********`)
`valuntil` `timestamptz` Срок действия пароля (используется только при аутентификации по паролю)
`useconfig` `text[]` Сеансовые значения по умолчанию для переменных конфигурации времени выполнения

52.37. pg_user_mappings

Представление `pg_user_mappings` открывает доступ к информации о сопоставлениях пользователей. По сути это просто доступное для чтения всеми отображение `pg_user_mapping`, в котором поле параметров показывается, только если у пользователя есть права читать его.

Таблица 52.37. Столбцы pg_user_mappings

Тип столбца Описание
`umid` `oid` (ссылается на `pg_user_mapping`.`oid`) OID сопоставления пользователей
`srvid` `oid` (ссылается на `pg_foreign_server`.`oid`) OID стороннего сервера, содержащего это сопоставление

Тип столбца Описание
`srvname` `name` (ссылается на `pg_foreign_server` .`srvname`) Имя стороннего сервера
`umuser` `oid` (ссылается на `pg_authid` .`oid`) OID сопоставляемой локальной роли, либо ноль, если сопоставление задаётся для всех
`usename` `name` Имя локального пользователя, для которого задано сопоставление
`umoptions` `text[]` Специальные параметры сопоставления пользователей, в виде строк «ключ=значение»

Для защиты информации о паролях, хранящейся в свойствах сопоставления пользователей, столбец `umoptions` при чтении будет содержать не NULL, если только выполняется одно из этих условий:

- текущий пользователь является объектом сопоставления и владельцем сервера или имеет право `USAGE` для него
- текущий пользователь является владельцем сервера и прочитывается сопоставление для `PUBLIC`
- текущий пользователь является суперпользователем

52.38. pg_views

Представление `pg_views` даёт доступ к полезной информации обо всех представлениях в базе данных.

Таблица 52.38. Столбцы `pg_views`

Тип столбца Описание
`schemaname` `name` (ссылается на `pg_namespace` .`nspname`) Имя схемы, содержащей представление
`viewname` `name` (ссылается на `pg_class` .`relname`) Имя представления
`viewowner` `name` (ссылается на `pg_authid` .`rolname`) Имя владельца представления
`definition` `text` Определение представления (реконструированный запрос SELECT)

52.39. pg_wait_events

Представление `pg_wait_events` содержит информацию о событиях ожидания.

Таблица 52.39. Столбцы `pg_wait_events`

Тип столбца Описание
`type` `text` Тип события ожидания
`name` `text` Название события ожидания
`description` `text` Описание события ожидания

Глава 53. Клиент-серверный протокол

Клиенты и серверы Postgres Pro взаимодействуют друг с другом, используя специальный протокол, основанный на сообщениях. Этот протокол поддерживается для соединений по TCP/IP и через Unix-сокеты. Для серверов, поддерживающих этот протокол, в IANA зарезервирован номер TCP-порта 5432, но на практике можно задействовать любой порт, не требующий особых привилегий.

В этой документации описана версия 3.0 этого протокола, реализованная в PostgreSQL версии 7.4 и новее. За описанием предыдущих версий протокола обратитесь к документации более ранних выпусков PostgreSQL. Один сервер способен поддерживать несколько версий протокола. Какую версию протокола пытается использовать клиент, сервер узнаёт из стартового сообщения при установлении соединения. Если старшая версия, запрашиваемая клиентом, не поддерживается сервером, соединение будет разорвано (например, это будет иметь место, если клиент запросит протокол версии 4.0, несуществующий на момент написания этого текста). Если младшая версия, запрашиваемая клиентом, не поддерживается сервером (например, клиент запросил версию 3.1, а сервер поддерживает только 3.0), сервер может либо разорвать соединение, либо ответить сообщением NegotiateProtocolVersion с указанием наибольшей младшей версии, которую он поддерживает. Затем клиент может решить либо продолжить установление соединения с указанной версией протокола, либо разорвать соединение.

Чтобы эффективно обслуживать множество клиентов, сервер запускает отдельный «обслуживающий» процесс для каждого клиента. В текущей реализации новый дочерний процесс запускается немедленно после обнаружения входящего подключения. Однако это происходит прозрачно для протокола. С точки зрения протокола, термины «обслуживающий процесс», «процесс заднего плана» и «сервер» взаимозаменяемы, как и «приложение переднего плана» и «клиент».

53.1. Обзор

В протоколе выделены отдельные фазы для запуска и обычной работы. На стадии запуска клиент устанавливает подключение к серверу и должен удовлетворить сервер, подтвердив свою подлинность. (Для этого может потребоваться одно или несколько сообщений, в зависимости от применяемого метода проверки подлинности.) Если всё проходит успешно, сервер сообщает клиенту о текущем состоянии, а затем переходит к обычной работе. Не считая начального стартового сообщения, в этой фазе протокола ведущую роль играет сервер.

В ходе обычной работы клиент передаёт запросы и другие команды серверу, а сервер возвращает результаты запросов и другие ответы. В некоторых случаях (например, с `NOTIFY`) сервер передаёт клиенту сообщения по своей инициативе, но по большей части эта фаза сеанса управляется запросами клиента.

Завершение сеанса обычно происходит по желанию клиента, но в некоторых случаях и сервер может принудительно завершить сеанс. В любом случае, когда сервер закрывает соединение, он предварительно откатывает любую открытую (незавершённую) транзакцию.

В процессе обычной работы команды SQL могут выполняться по одному из двух разделов протокола. При «простом» выполнении запросов клиент посылает текстовую строку запроса, которую сервер сразу же разбирает и выполняет. При «расширенном» выполнении запросов обработка разделяется на несколько этапов: разбор, привязывание значений параметров и исполнение. Это даёт дополнительную гибкость и может увеличить быстродействие ценой большей сложности.

При обычной работе также поддерживаются дополнительные разделы протокола для специальных операций, например `COPY`.

53.1.1. Обзор обмена сообщениями

Всё взаимодействие представляет собой поток сообщений. Первый байт сообщения определяет тип сообщения, а следующие четыре байта задают длину остального сообщения (эта длина включает размер самого поля длины, но не байт с типом сообщения). Остальное содержимое сообщения определяется его типом. По историческим причинам в самом первом сообщении, передаваемом клиентом, (стартовом сообщении) первый байт с типом сообщения отсутствует.

Чтобы не потерять синхронизацию в потоке сообщений, и серверы, и клиенты обычно считывают всё сообщение в буфер (его размер определяется счётчиком байт), прежде чем обрабатывать его содержимое. Это позволяет без труда продолжить работу, если возникает ошибка при разборе сообщения. В исключительных случаях (например, при нехватке памяти для помещения сообщения в буфер), счётчик байт помогает получателю определить, сколько поступающих байт нужно пропустить, прежде чем продолжать получать сообщения.

С другой стороны, и клиенты, и серверы, ни при каких условиях не должны передавать неполные сообщения. Чтобы этого не допустить, обычно всё сообщение сначала размещается в буфере, и только потом передаётся. Если в процессе отправки или получения сообщения происходит сбой передачи, единственным разумным вариантом продолжения будет прерывание соединения, так как вероятность восстановления синхронизации по границам сообщений в этой ситуации минимальна.

53.1.2. Обзор расширенного выполнения запросов

В расширенном протоколе запросов исполнение команд SQL разделяется на несколько этапов. Состояние между этапами представляется объектами двух типов: *подготовленные операторы* и *порталы*. Подготовленный оператор представляет собой результат разбора и семантического анализа текстовой строки запроса. Подготовленный оператор сам по себе не готов для исполнения, так как он может не иметь конкретных значений для *параметров*. Портал представляет собой готовый к исполнению или уже частично выполненный оператор, в котором заданы все недостающие значения параметров. (Для операторов `SELECT` портал равнозначен открытому курсору, но мы выбрали другой термин, так как курсоры неприменимы к операторам, отличным от `SELECT`.)

Общий цикл выполнения состоит из этапа *разбора*, на котором из текстовой строки запроса создаётся подготовленный оператор; этапа *привязки*, на котором из подготовленного оператора и значений для необходимых параметров создаётся портал; и этапа *выполнения*, на котором исполняется запрос портала. В случае запроса, возвращающего строки (`SELECT`, `SHOW` и т. д.), можно указать, чтобы за один шаг выполнения возвращалось только ограниченное число строк, так что для завершения операции понадобятся несколько шагов выполнения.

Сервер может контролировать одновременно несколько подготовленных операторов и порталов (но учтите, что они существуют только в рамках сеанса и никогда не разделяются между сеансами). Обращаться к подготовленным операторам и порталам можно по именам, которые назначаются им при создании. Кроме того, существуют и «безымянные» подготовленные операторы и порталы. Хотя они практически не отличаются от именованных объектов, операции с ними оптимизированы для разового выполнения запроса с последующим освобождением объекта, тогда как операции с именованными объектами оптимизируются в расчёте на многоразовое использование.

53.1.3. Форматы и коды форматов

Данные определённого типа могут передаваться в одном из нескольких различных *форматов*. С версии 7.4 PostgreSQL поддерживаются только текстовый («text») и двоичный («binary») форматы, но в протоколе предусмотрены возможности для расширения в будущем. Ожидаемый формат для любого значения задаётся *кодом формата*. Клиенты могут указывать код формата для каждого передаваемого значения параметра и для каждого столбца результата запроса. Текстовый формат имеет код ноль, двоичный — код один, а другие коды оставлены для определения в будущем.

Текстовым представлением значений будут строки, которые выдаются и принимаются функциями ввода-вывода определённого типа данных. В передаваемом представлении завершающий нулевой символ отсутствует, клиент должен добавить его сам, если хочет обрабатывать такое представление в виде строки C. (Собственно, данные в текстовом формате не могут содержать нулевые символы.)

В двоичном представлении целых чисел применяется сетевой порядок байтов (старший байт идёт первым). Какое именно двоичное представление имеют другие типы данных, можно узнать из документации. Но учтите, что двоичное представление сложных типов данных может меняться от версии к версии сервера; с точки зрения портируемости обычно лучше текстовый формат.

53.2. Поток сообщений

В этом разделе описывается поток сообщений и семантика каждого типа сообщений. (Подробнее точное представление каждого сообщения описывается в Разделе 53.7.) В зависимости от состояния соединения выделяются несколько различных разделов протокола: запуск, выполнение запросов, вызов функции, копирование (`COPY`) и завершение. Есть также специальные средства для асинхронных операций (в частности, для уведомлений и отмены команд), которые могут выполняться в любой момент после этапа запуска.

53.2.1. Запуск

Чтобы начать сеанс, клиент открывает подключение к серверу и передаёт стартовое сообщение. В этом сообщении содержатся имена пользователя и базы данных, к которой пользователь хочет подключиться; в нём также определяется, какая именно версия протокола будет использоваться. (Стартовое сообщение также может содержать дополнительные значения для параметров времени выполнения.) Проанализировав эту информацию и содержимое своих файлов конфигурации (в частности, `pg_hba.conf`), сервер определяет, можно ли предварительно разрешить это подключение, и какая дополнительная проверка подлинности требуется.

Затем сервер отправляет соответствующее сообщение с запросом аутентификации, на которое клиент должен ответить сообщением, подтверждающим его подлинность (например, по паролю). Для всех методов аутентификации, за исключением GSSAPI, SSPI и SASL, может быть максимум один запрос и один ответ. Для некоторых методов ответ клиента вообще не требуется, так что запрос аутентификации также не передаётся. Методы GSSAPI, SSPI и SASL для прохождения проверки подлинности могут потребовать выполнить серию обменов пакетами.

Цикл аутентификации заканчивает сервер, либо запрещая соединение (ErrorResponse), либо принимая его (отправляя AuthenticationOk).

Сервер может передавать в этой фазе следующие сообщения:

ErrorResponse (Ошибочный ответ)

Попытка соединения была отвергнута. Сразу после этого сервер закрывает соединение.

AuthenticationOk (Аутентификация пройдена)

Обмен сообщениями для проверки подлинности завершён успешно.

AuthenticationKerberosV5 (Аутентификация Kerberos V5)

Клиент должен теперь принять участие в диалоге аутентификации по протоколу Kerberos V5 (здесь его детали не описывается, так как они относятся к спецификации Kerberos) с сервером. Если этот диалог завершается успешно, сервер отвечает AuthenticationOk, иначе — ErrorResponse. Этот вариант аутентификации больше не поддерживается.

AuthenticationCleartextPassword (Аутентификация с открытым паролем)

Клиент должен передать в ответ сообщение PasswordMessage, содержащее пароль в открытом виде. Если пароль правильный, сервер отвечает ему AuthenticationOk, иначе — ErrorResponse.

AuthenticationMD5Password (Аутентификация с паролем MD5)

Клиент должен передать в ответ сообщение PasswordMessage с результатом преобразования пароля (и имени пользователя) в хеш MD5 с последующим хешированием с четырёхбайтовым случайным значением соли, переданным в сообщении AuthenticationMD5Password. Если пароль правильный, сервер отвечает AuthenticationOk, иначе — ErrorResponse. Содержимое сообщения PasswordMessage можно вычислить в SQL как `concat('md5', md5(concat(md5(concat(password, username)), random-salt)))`. (Учтите, что функция `md5()` возвращает результат в виде шестнадцатеричной строки.)

AuthenticationGSS (Аутентификация GSS)

Клиент должен начать согласование GSSAPI. В ответ на это сообщение клиент отправляет GSSResponse с первой частью потока данных GSSAPI. Если потребуются дополнительные сообщения, сервер передаст в ответ AuthenticationGSSContinue.

AuthenticationSSPI (Аутентификация SSPI)

Клиент должен начать согласование SSPI. В ответ на это сообщение клиент отправляет GSSResponse с первой частью потока данных SSPI. Если потребуются дополнительные сообщения, сервер передаст в ответ AuthenticationGSSContinue.

AuthenticationGSSContinue (Продолжение аутентификации GSS)

Это сообщение содержит данные ответа на предыдущий шаг согласования GSSAPI или SSPI (AuthenticationGSS, AuthenticationSSPI или предыдущего AuthenticationGSSContinue). Если в структуре GSSAPI или SSPI в этом сообщении указывается, что для завершения аутентификации требуются дополнительные данные, клиент должен передать их в очередном сообщении GSSResponse. Если этим сообщением завершается проверка подлинности GSSAPI или SSPI, сервер затем передаёт AuthenticationOk, сообщая об успешной проверке подлинности, либо ErrorResponse, сообщая об ошибке.

AuthenticationSASL (Аутентификация SASL)

Клиент должен начать согласование SASL, используя один из механизмов SASL, перечисленных в сообщении. В ответ на это сообщение клиент отправляет SASLInitialResponse с именем выбранного механизма и первой частью потока данных SASL. Если потребуются дополнительные сообщения, сервер передаст в ответ AuthenticationSASLContinue. За подробностями обратитесь к Разделу 53.3.

AuthenticationSASLContinue (Продолжение аутентификации SASL)

Это сообщение содержит данные вызова с предыдущего шага согласования SASL (AuthenticationSASL или предыдущего AuthenticationSASLContinue). Клиент должен передать в ответ сообщение SASLResponse.

AuthenticationSASLFinal (Окончание аутентификации SASL)

Аутентификация SASL завершена с дополнительными данными для клиента, специфичными для механизма. Затем сервер передаст сообщение AuthenticationOk, говорящее об успешной аутентификации, или ErrorResponse, говорящее об ошибке. Данное сообщение передаётся, только если механизм SASL должен в завершение передать с сервера клиенту дополнительные специфичные данные.

NegotiateProtocolVersion

Сервер не поддерживает младшую версию протокола, запрошенную клиентом, но поддерживает более раннюю версию протокола; в этом сообщении указывается наибольшая поддерживаемая младшая версия. Это сообщение будет также передаваться, если клиент запросил в стартовом пакете неподдерживаемые параметры протокола (то есть, начинающиеся с `_pq_.`). За этим сообщением должен последовать или ответ ErrorResponse, или сообщение, говорящее об успехе или неудаче проверки подлинности.

Если клиент не поддерживает метод проверки подлинности, запрошенный сервером, он должен немедленно закрыть соединение.

Получив сообщение AuthenticationOk, клиент должен ждать дальнейших сообщений от сервера. В этой фазе запускается обслуживающий процесс, а клиент представляет собой просто заинтересованного наблюдателя. Попытка запуска может быть неудачной (и клиент получит ErrorResponse) либо сервер может отказать в поддержке запрошенной младшей версии протокола (NegotiateProtocolVersion), но в обычной ситуации обслуживающий процесс передаёт несколько сообщений ParameterStatus, BackendKeyData и, наконец, ReadyForQuery.

В ходе этой фазы обслуживающий процесс попытается применить все параметры времени выполнения, полученные в стартовом сообщении. Если это удастся, эти значения становятся сеансовыми значениями по умолчанию. При ошибке он передаёт ErrorResponse и завершается.

Обслуживающий процесс может передавать в этой фазе следующие сообщения:

BackendKeyData (Данные ключа сервера)

В этом сообщении передаётся секретный ключ, который клиент должен сохранить, чтобы впоследствии иметь возможность выполнять запросы. Клиент не должен отвечать на это сообщение, он должен дожидаться сообщения ReadyForQuery.

ParameterStatus (Состояние параметров)

Это сообщение говорит клиенту о текущих (начальных) значениях параметров обслуживающего процесса, например, client_encoding или DateStyle. Клиент может проигнорировать это сообщение или сохранить значения для дальнейшего использования; за дополнительными подробностями обратитесь к Подразделу 53.2.7. Клиент не должен отвечать на это сообщение, он должен дожидаться сообщения ReadyForQuery.

ReadyForQuery (Готов к запросам)

Запуск завершён. Теперь клиент может выполнять команды.

ErrorResponse (Ошибочный ответ)

Запуск не удался. Соединение закрывается после передачи этого сообщения.

NoticeResponse (Ответ с замечанием)

Выдаётся предупреждающее сообщение. Клиент должен вывести это сообщение, но продолжать ожидать сообщения ReadyForQuery или ErrorResponse.

Сообщение ReadyForQuery в данной фазе ничем не отличается от сообщений, который передаёт сервер после каждого цикла команд. В зависимости от условий реализации клиента, можно воспринимать сообщение ReadyForQuery как начинающее цикл команд, либо как завершающее фазу запуска и каждый последующий цикл команд.

53.2.2. Простое выполнение запросов

Цикл простого выполнения запросов начинает клиент, передавая серверу сообщение Query. Это сообщение включает команду (или команды) SQL, выраженную в виде текстовой строки. В ответ сервер передаёт одно или несколько сообщений, в зависимости от строки запроса, и завершает цикл сообщением ReadyForQuery. ReadyForQuery говорит клиенту, что он может безопасно передавать новую команду. (На самом деле клиент может передать следующую команду, не дожидаясь ReadyForQuery, но тогда он сам должен разобраться в ситуации, когда первая команда завершается ошибкой, а последующая выполняется успешно.)

Сервер может передавать в этой фазе следующие ответные сообщения:

CommandComplete (Команда завершена)

Команда SQL выполнена нормально.

CopyInResponse (Ответ входящего копирования)

Сервер готов копировать данные, получаемые от клиента, в таблицу; см. Подраздел 53.2.6.

CopyOutResponse (Ответ исходящего копирования)

Сервер готов копировать данные из таблицы клиенту; см. Подраздел 53.2.6.

RowDescription (Описание строк)

Показывает, что в ответ на запрос `SELECT`, `FETCH` и т. п. будут возвращены строки. В содержимом этого сообщения описывается структура столбцов этих строк. За ним для каждой строки, возвращаемой клиенту, следует сообщение DataRow.

DataRow (Строка данных)

Одна строка из набора, возвращаемого запросом `SELECT`, `FETCH` и т. п.

EmptyQueryResponse (Ответ на пустой запрос)

Была принята пустая строка запроса.

ErrorResponse (Ошибочный ответ)

Произошла ошибка.

ReadyForQuery (Готов к запросам)

Обработка строки запроса завершена. Чтобы отметить это, отправляется отдельное сообщение, так как строка запроса может содержать несколько команд SQL. (Сообщение CommandComplete говорит о завершении обработки одной команды SQL, а не всей строки.) ReadyForQuery передаётся всегда, и при успешном завершении обработки, и при ошибке.

NoticeResponse (Ответ с замечанием)

Выдаётся предупреждение, связанное с запросом. Эти замечания дополняют другие ответы, то есть сервер, выдавая их, продолжает обрабатывать команду.

Ответ на запрос `SELECT` (или другие запросы, возвращающие наборы строк, такие как `EXPLAIN` и `SHOW`) обычно состоит из RowDescription, нуля или нескольких сообщений DataRow, и завершающего CommandComplete. Для команды `COPY` с вводом или выводом данных через клиента, применяется специальный протокол, описанный в Подразделе 53.2.6. Со всеми другими типами запросами обычно выдаётся только сообщение CommandComplete.

Так как строка запроса может содержать несколько запросов (разделённых точкой с запятой), до завершения обработки всей строки сервер может передать несколько серий таких ответов. Когда сервер завершает обработку всей строки и готов принять следующую строку запроса, он выдаёт сообщение ReadyForQuery.

Если получена полностью пустая строка запроса (не содержащая ничего, кроме пробельных символов), ответом будет EmptyQueryResponse с последующим ReadyForQuery.

В случае ошибки выдаётся ErrorResponse с последующим ReadyForQuery. Сообщение ErrorResponse прерывает дальнейшую обработку строки запроса (даже если в ней остались другие запросы). Заметьте, что оно может быть выдано и в середине последовательности сообщений, выдаваемых в ответ на отдельный запрос.

В случае простого выполнения запросов получаемые значения всегда передаются в текстовом формате, за исключением результатов команды `FETCH` для курсора, объявленного с атрибутом `BINARY`. С такой командой значения передаются в двоичном формате. Какой именно формат используется, определяют коды формата, передаваемые в сообщении RowDescription.

Клиент должен быть готов принять сообщения ErrorResponse и NoticeResponse, ожидая любой другой тип сообщений. Также обратитесь к Подразделу 53.2.7 за информацией о сообщениях, которые сервер может выдавать в ответ на внешние события.

Код клиента рекомендуется реализовывать в виде конечного автомата, который в любой момент будет принимать сообщения всех типов, имеющих смысл на данном этапе, но не программировать жёстко обработку точной последовательности сообщений.

53.2.2.1. Несколько операторов в простом протоколе запросов

Когда простое сообщение Query содержит несколько SQL-операторов (разделённых точкой с запятой), эти операторы выполняются в одной транзакции, если только среди них нет явных команд управления транзакциями, меняющих это поведение. Например, если сообщение содержит

```
INSERT INTO mytable VALUES(1);
SELECT 1/0;
```

```
INSERT INTO mytable VALUES(2);
```

, то ошибка деления на ноль в `SELECT` приведёт к откату результата первого `INSERT`. Более того, вследствие прерывания обработки сообщения на первой ошибке, второй `INSERT` не будет выполняться вовсе.

Если же сообщение содержит:

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELECT 1/0;
```

результат первого `INSERT` фиксируется явной командой `COMMIT`. Второй `INSERT` и последующий `SELECT` будут так же обрабатываться в одной транзакции, поэтому ошибка деления на ноль приведёт к откату второго `INSERT`, и не затронет первый.

Для реализации этого поведения операторы в составном сообщении запроса выполняются в *неявном блоке транзакции*, если только в сообщении нет явного блока транзакции, в котором они должны выполняться. Основное отличие неявного блока транзакции от обычного состоит в том, что неявный блок автоматически закрывается в конце сообщения Query — либо неявно фиксируется при отсутствии ошибок, либо неявно откатывается в противном случае. Подобная неявная фиксация или отмена транзакции имеет место, когда оператор выполняется отдельно (вне блока транзакции).

Если в рамках сеанса уже начат блок транзакции (в результате выполнения оператора `BEGIN` из некоторого предыдущего сообщения), сообщение Query просто продолжает этот блок независимо от того, содержится ли в нём один оператор или несколько. Однако если сообщение Query содержит команду `COMMIT` или `ROLLBACK`, закрывающую существующий блок транзакций, то все последующие команды в нём выполняются в неявном блоке транзакции. И напротив, если составное сообщение Query содержит команду `BEGIN`, она начинает обычный блок транзакции, который будет закончен только явными командами `COMMIT` или `ROLLBACK`, в каком бы сообщении Query, текущем или последующих, они ни содержались. Если `BEGIN` следует за операторами, которые выполнялись в неявном блоке транзакции, эти операторы не фиксируются немедленно; они задним числом включаются в новый обычный блок транзакции.

Операторы `COMMIT` и `ROLLBACK`, фигурирующие в неявном блоке транзакции, выполняются как обычно, закрывая неявный блок; однако при этом будет выдано предупреждение, так как `COMMIT` или `ROLLBACK` без предшествующего `BEGIN` могут выполняться по ошибке. Если за этими операторами следуют другие, для них будет начат новый неявный блок транзакции.

Точки сохранения в неявных блоках транзакций не допускаются, так как они будут конфликтовать с правилом автоматического закрытия блока при любой ошибке.

Помните, что, вне зависимости от наличия любых команд управления транзакциями, выполнение сообщения Query останавливается при первой же ошибке. Таким образом, с данными командами:

```
BEGIN;
SELECT 1/0;
ROLLBACK;
```

в одном сообщении Query сеанс останется внутри прерванного обычного блока транзакции, так как команда `ROLLBACK` не достигается после ошибки деления на ноль. Для приведения сеанса в порядок потребуется выполнить ещё один `ROLLBACK`.

Также следует заметить, что первоначальный лексический и синтаксический анализ производится для всей строки запроса, прежде чем какая-либо её часть будет выполняться. Таким образом, простые ошибки (например, опечатка в ключевом слове) в последующих операторах могут привести к тому, что не будет выполнен и ни один из предшествующих операторов. Это обычно незаметно для пользователей, так как эти операторы откатились бы всё равно при выполнении в неявном блоке транзакции. Однако эта особенность может проявиться при попытке выполнить несколько транзакций в одном составном запросе. Например, если из-за опечатки предыдущий пример превратился в:

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELCT 1/0;
```

ни один из операторов не будет выполняться, и отличие проявится в том, что первый `INSERT` не будет зафиксирован. Ошибки, выявленные на стадии семантического анализа или позже, например, опечатки в имени таблиц или столбца, такого влияния на выполнение не оказывают.

Наконец, обратите внимание, что все операторы в сообщении Query учитывают одно и то же значение функции `statement_timestamp()`, поскольку данная временная метка обновляется только при получении сообщения Query. Это приводит к тому, что также учитывается одно и то же значение функции `transaction_timestamp()`, за исключением случаев, когда строка запроса завершает ранее начатую транзакцию и начинает новую.

53.2.3. Расширенное выполнение запросов

Расширенный протокол запросов разбивает вышеописанный простой протокол на несколько шагов. Результаты подготовительных шагов можно многократно использовать повторно для улучшения эффективности. Кроме того, он открывает дополнительные возможности, в частности, возможность передавать значения данных в отдельных параметрах вместо того, чтобы внедрять их непосредственно в строку запроса.

В расширенном протоколе клиент сначала передаёт сообщение Parse с текстовой строкой запроса и, возможно, некоторыми сведениями о типах параметров и именем целевого объекта подготовленного оператора (если имя пустое, создаётся безымянный подготовленный оператор). Ответом на это сообщение будет ParseComplete или ErrorResponse. Типы параметров указываются по OID; при отсутствии явного указания анализатор запроса пытается определить типы данных так же, как он делал бы для нетипизированных строковых констант.

Примечание

Тип данных параметра можно оставить неопределённым, задав для него значение ноль, либо сделав массив с OID типов параметров короче, чем набор символов параметров (`$`*`n`*), используемых в строке запроса. Другой особый случай — передача типа параметра как `void` (то есть передача OID псевдотипа `void`). Это предусмотрено для того, чтобы символы параметров можно было использовать для параметров функций, на самом деле представляющих собой параметры OUT. Обычно параметр `void` нельзя использовать ни в каком контексте, но если такой параметр фигурирует в списке параметров функции, он фактически игнорируется. Например, вызову функции `foo($1,$2,$3,$4)` может соответствовать функция с аргументами IN и двумя OUT, если аргументы `$3` и `$4` объявлены как имеющие тип `void`.

Примечание

Строка запроса, содержащаяся в сообщении Parse, не может содержать больше одного оператора SQL; иначе выдаётся синтаксическая ошибка. Это ограничение отсутствует в простом протоколе запросов, но присутствует в расширенном протоколе, так как добавление поддержки подготовленных операторов или порталов, содержащих несколько команд, неоправданно усложнило бы протокол.

В случае успешного создания именованный подготовленный оператор продолжает существовать до завершения текущего сеанса, если только он не будет уничтожен явно. Безымянный подготовленный оператор сохраняется только до следующей команды Parse, в которой целевым является безымянный оператор. (Заметьте, что простое сообщение Query также уничтожает безымянный оператор.) Именованные операторы должны явно закрываться, прежде чем их можно будет переопределить другим сообщением Parse, но для безымянных операторов это не требуется. Именованные подготовленные операторы также можно создавать и вызывать на уровне команд SQL, используя команды `PREPARE` и `EXECUTE`.

Когда подготовленный оператор существует, его можно подготовить к выполнению сообщением Bind. В сообщении Bind задаётся имя исходного подготовленного оператора (пустая строка подразумевает безымянный подготовленный оператор), имя целевого портала (пустая строка подразумевает безымянный портал) и значения для любых шаблонов параметров, представленных в подготовленном операторе. Набор передаваемых значений должен соответствовать набору параметров, требующихся для подготовленного оператора. (Если вы объявили параметры `void` в сообщении Parse, передайте для них значения NULL в сообщении Bind.) Bind также принимает указание формата для данных, возвращаемых в результате запроса; формат можно указать для всех данных, либо для отдельных столбцов. Ответом на это сообщение будет BindComplete или ErrorResponse.

Примечание

Выбор между текстовым и двоичным форматом вывода определяется кодами формата, передаваемыми в Bind, вне зависимости от команды SQL. При использовании расширенного протокола запросов атрибут `BINARY` в объявлении курсоров не имеет значения.

Планирование запроса обычно имеет место при обработке сообщения Bind. Если подготовленный оператор не имеет параметров, либо он выполняется многократно, сервер может сохранить созданный план и использовать его повторно при последующих сообщениях Bind для того же подготовленного оператора. Однако он будет делать это, только если решит, что можно получить универсальный план, который не будет значительно неэффективнее планов, зависящих от конкретных значений параметров. С точки зрения протокола это происходит незаметно.

В случае успешного создания объект именованного портала продолжает существование до конца текущей транзакции, если только он не будет уничтожен явно. Безымянный портал уничтожается в конце транзакции или при выполнении следующей команды Bind, в которой в качестве целевого выбирается безымянный портал. (Заметьте, что простое сообщение Query также уничтожает безымянный портал.) Именованные порталы должны явно закрываться, прежде чем их можно будет явно переопределить другим сообщением Bind, но это не требуется для безымянных порталов. Именованные порталы также можно создавать и вызывать на уровне команд SQL, используя команды `DECLARE CURSOR` и `FETCH`.

Когда портал существует, его можно запустить на выполнение сообщением Execute. В сообщении Execute указывается имя портала (пустая строка подразумевает безымянный портал) и максимальное число результирующих строк (ноль означает «выбрать все строки»). Число результирующих строк имеет значение только для порталов, которые содержат команды, возвращающие наборы строк; в других случаях команда всегда выполняется до завершения и число строк игнорируется. В ответ на Execute могут быть получены те же сообщения, что описаны выше для запросов, выполняемых через простой протокол, за исключением того, что после Execute не выдаются сообщения ReadyForQuery и RowDescription.

Если операция Execute оканчивается до завершения выполнения портала (из-за достижения ненулевого ограничения на число строк), сервер отправляет сообщение PortalSuspended; появление этого сообщения говорит клиенту о том, что для завершения операции с данным порталом нужно выдать ещё одно сообщение Execute. Сообщение CommandComplete, говорящее о завершении исходной команды SQL, не передаётся до завершения выполнения портала. Таким образом, фаза Execute всегда заканчивается при появлении одного из сообщений: CommandComplete, EmptyQueryResponse (если портал был создан из пустой строки запроса), ErrorResponse или PortalSuspended.

В конце каждой серии сообщений расширенного протокола запросов клиент должен выдать сообщение Sync. Получив это сообщение без параметров, сервер закрывает текущую транзакцию, если команды выполняются не внутри блока транзакции `BEGIN`/`COMMIT` (под «закрытием» понимается фиксация при отсутствии ошибок или откат в противном случае). Затем он выдаёт ответ ReadyForQuery. Целью сообщения Sync является обозначение точки синхронизации для восстановления в случае ошибок. Если при обработке сообщений расширенного протокола запросов происходит ошибка, сервер выдаёт ErrorResponse, затем считывает и пропускает сообщения до Sync, после чего выдаёт ReadyForQuery и возвращается к обычной обработке сообщений. (Но заметьте, что он не будет пропускать следующие сообщения, если ошибка происходит *в процессе* обработки Sync — это гарантирует, что для каждого Sync будет передаваться в точности одно сообщение ReadyForQuery.)

Примечание

Сообщение Sync не приводит к закрытию блока транзакции, открытого командой `BEGIN`. Выявить эту ситуацию можно, используя информацию о состоянии транзакции, содержащуюся в сообщении ReadyForQuery.

В дополнение к этим фундаментальным и обязательным операциям, расширенный протокол запросов позволяет выполнить и несколько дополнительных операций.

В сообщении Describe (в вариации для портала) задаётся имя существующего портала (пустая строка обозначает безымянный портал). В ответ передаётся сообщение RowDescription, описывающее строки, которые будут возвращены при выполнении портала; либо сообщение NoData, если портал не содержит запроса, возвращающего строки; либо ErrorResponse, если такого портала нет.

В сообщении Describe (в вариации для оператора) задаётся имя существующего подготовленного оператора (пустая строка обозначает безымянный подготовленный оператор). В ответ передаётся сообщение ParameterDescription, описывающее параметры, требующиеся для оператора, за которым следует сообщение RowDescription, описывающее строки, которые будут возвращены, когда оператор будет собственно выполнен (или сообщение NoData, если оператор не возвратит строки). ErrorResponse выдаётся, если такой подготовленный оператор отсутствует. Заметьте, что так как команда Bind не выполнялась, сервер ещё не знает, в каком формате будут возвращаться столбцы; в этом случае поля кодов формата в сообщении RowDescription будут содержать нули.

Подсказка

В большинстве случаев клиент должен выдать ту или иную вариацию Describe, прежде чем выдавать Execute, чтобы понять, как интерпретировать результаты, которые он получит.

Сообщение Close закрывает существующий подготовленный оператор или портал и освобождает связанные ресурсы. При попытке выполнить Close для имени несуществующего портала или оператора ошибки не будет. Ответ на это сообщение обычно CloseComplete, но может быть и ErrorResponse, если при освобождении ресурсов возникают проблемы. Заметьте, что при закрытии подготовленного оператора неявно закрываются все открытые порталы, которые были получены из этого оператора.

Сообщение Flush не приводит к генерации каких-либо данных, а указывает серверу передать все данные, находящиеся в очереди в его буферах вывода. Сообщение Flush клиент должен отправлять после любой команды расширенного протокола запросов, кроме Sync, если он желает проанализировать результаты этой команды, прежде чем выдавать следующие команды. Без Flush сообщения, возвращаемые сервером, будут объединяться вместе в минимальное количество пакетов с целью уменьшения сетевого трафика.

Примечание

Простое сообщение Query примерно равнозначно последовательности сообщений Parse, Bind, Describe (для портала), Execute, Close, Sync, с использованием объектов подготовленного оператора и портала без имён и без параметров. Одно из отличий состоит в том, что такое сообщение может содержать в строке запроса несколько операторов SQL, для каждого из которых по очереди автоматически выполняется последовательность Bind/Describe/Execute. Другое отличие заключается в том, что в ответ на него не приходят сообщения ParseComplete, BindComplete, CloseComplete или NoData.

53.2.4. Конвейеризация

Расширенный протокол запросов позволяет использовать *конвейеризацию*, то есть отправлять серию запросов, не ожидая завершения каждого предыдущего. При этом уменьшается количество сетевых операций, необходимых для выполнения данной последовательности запросов. Однако пользователь должен тщательно продумывать поведение в случае сбоя одного из шагов, поскольку последующие запросы уже будут отправлены на сервер.

Например, можно объединить всю последовательность запросов в одну транзакцию, заключив её в `BEGIN` ... `COMMIT`. Однако если какие-то команды должны выполняться независимо от остальных, этот способ не подходит.

Расширенный протокол запросов позволяет реализовать поведение в случае ошибки по-другому, исключив передачу сообщений Sync между зависимыми шагами. Поскольку после ошибки сервер пропускает командные сообщения до тех пор, пока не получит Sync, более поздние команды в конвейере автоматически пропускаются при сбое более ранней, при этом клиенту не нужно явно управлять этим с помощью `BEGIN` и `COMMIT`. Сегменты конвейера, фиксируемые независимо, можно разделять сообщениями Sync.

Если клиент не передаёт команду `BEGIN` явно, то каждое сообщение Sync обычно вызывает неявное выполнение `COMMIT` в случае успешного завершения предыдущих шагов или `ROLLBACK` в случае сбоя. Однако есть несколько команд DDL (например, `CREATE DATABASE`), которые нельзя выполнить внутри блока транзакции. Если такая команда выполняется в конвейере, она успешно выполнится, только если будет первой. Кроме того, в случае успешного выполнения она вызовет немедленную фиксацию для сохранения согласованности базы данных. Таким образом, если сразу за ней отправить Sync, никаких дополнительных действий это сообщение не вызовет, кроме ответа ReadyForQuery.

Применяя этот метод, факт завершения конвейера нужно определять, подсчитывая сообщения ReadyForQuery и ожидая достижения количества отправленных сообщений Sync. Вариант с подсчётом сообщений о завершении команд ненадёжен, так как некоторые команды могут пропускаться, так что такие сообщения выдаваться не будут.

53.2.5. Вызов функций

Раздел протокола «Вызов функций» позволяет клиенту запросить непосредственный вызов любой функции, существующей в системном каталоге `pg_proc`. При этом клиент должен иметь право на выполнение этой функции.

Примечание

Этот раздел протокола считается устаревшим и в новом коде использовать его не следует. Примерно тот же результат можно получить, подготовив оператор с командой `SELECT function($1, ...)`. При таком подходе цикл вызова функции заменяется последовательностью Bind/Execute.

Цикл вызова функции начинает клиент, передавая серверу сообщение FunctionCall. Сервер возвращает одно или несколько сообщений ответа, в зависимости от результата вызова функции, и завершающее сообщение ReadyForQuery. ReadyForQuery говорит клиенту, что он может свободно передавать новый запрос или вызов функции.

Сервер может передавать в этой фазе следующие ответные сообщения:

ErrorResponse (Ошибочный ответ)

Произошла ошибка.

FunctionCallResponse (Ответ на вызов функции)

Вызов функции завершён и в этом сообщении передаётся её результат. (Заметьте, что протокол вызова функций позволяет выдать только один скалярный результат, но не кортеж или набор результатов.)

ReadyForQuery (Готов к запросам)

Обработка вызова функции завершена. В ответ всегда передаётся ReadyForQuery, независимо от того, была ли функция выполнена успешно или с ошибкой.

NoticeResponse (Ответ с замечанием)

Выдаётся предупреждение, связанное с вызовом функции. Эти замечания дополняют другие ответы, то есть сервер, выдавая их, продолжает обрабатывать вызов.

53.2.6. Операции COPY

Команда `COPY` позволяет обеспечить скоростную передачу данных на сервер или с сервера. Операции входящего и исходящего копирования переключают соединение на использование отдельного раздела протокола, которое завершается только в конце операции.

Режим входящего копирования (передача данных на сервер) включается, когда клиент выполняет SQL-оператор `COPY FROM STDIN`. Переходя в этот режим, сервер передаёт клиенту сообщение CopyInResponse. После этого клиент должен передать ноль или более сообщений CopyData, образующих поток входных данных. (При этом границы сообщений не обязательно должны совпадать с границами строк данных, хотя часто имеет смысл выровнять их.) Клиент может завершить режим входящего копирования, передав либо сообщение CopyDone (говорящее об успешном завершении), либо CopyFail (которое приведёт к завершению SQL-оператора `COPY` с ошибкой). При этом сервер вернётся в обычный режим обработки, в котором он находился до выполнения команды `COPY` (это может быть простой или расширенный протокол запросов). Затем он отправит сообщение CommandComplete (в случае успешного завершения) или ErrorResponse (в противном случае).

В случае возникновения ошибки в режиме входящего копирования (включая получение сообщения CopyFail), сервер выдаёт сообщение ErrorResponse. Если команда `COPY` была получена в сообщении расширенного запроса, сервер не будет обрабатывать последующие сообщения клиента, пока не получит сообщение Sync, после которого он выдаст ReadyForQuery и вернётся в обычный режим работы. Если команда `COPY` была получена в сообщении простого запроса, остальная часть сообщения игнорируется и сразу выдаётся ReadyForQuery. В любом случае все последующие сообщения CopyData, CopyDone или CopyFail, поступающие от клиента, будут просто игнорироваться.

В режиме входящего копирования сервер игнорирует поступающие сообщения Flush и Sync. При поступлении сообщений любого другого типа, не связанного с копированием, возникает ошибка, приводящая к прерыванию режима входящего копирования, как описано выше. (Исключение для сообщений Flush и Sync сделано для удобства клиентских библиотек, которые всегда передают Flush или Sync после сообщения Execute, не проверяя, не запускается ли в нём команда `COPY FROM STDIN`.)

Режим исходящего копирования (передача данных с сервера) включается, когда клиент выполняет SQL-оператор `COPY TO STDOUT`. Переходя в этот режим, сервер передаёт клиенту сообщение CopyOutResponse, за ним ноль или более сообщений CopyData (всегда одно сообщение для каждой строки) и в завершение CopyDone. Затем сервер возвращается в обычный режим обработки, в котором он находился до выполнения команды `COPY`, и передаёт CommandComplete. Клиент не может прервать передачу (кроме как закрыв соединение или выдав запрос Cancel), но он может игнорировать ненужные ему сообщения CopyData и CopyDone.

В случае обнаружения ошибки в режиме исходящего копирования, сервер выдаёт сообщение ErrorResponse и возвращается к обычной обработке. Клиент должен воспринимать поступление ErrorResponse как завершение режима исходящего копирования.

Между сообщениями CopyData могут поступать сообщения NoticeResponse и ParameterStatus; клиенты должны обрабатывать их и быть готовы принимать и другие типы асинхронных сообщений (см. Подраздел 53.2.7). В остальном, сообщения любых типов, кроме CopyData и CopyDone, могут восприниматься как завершающие режим исходящего копирования.

Есть ещё один режим копирования, называемый двусторонним копированием и обеспечивающий высокоскоростную передачу данных на *и* с сервера. Двустороннее копирование запускается, когда клиент в режиме walsender выполняет оператор `START_REPLICATION`. В ответ сервер передаёт клиенту сообщение CopyBothResponse. Затем и сервер, и клиент могут передавать друг другу сообщения CopyData, пока кто-то из них не завершит передачу сообщением CopyDone. Когда сообщение CopyDone передаёт клиент, соединение переходит из режима двустороннего в режим исходящего копирования и клиент больше не может передавать сообщения CopyData. Аналогично, когда сообщение CopyDone передаёт сервер, соединение переходит в режим входящего копирования и сервер больше не может передавать сообщения CopyData. Когда сообщения CopyDone переданы обеими сторонами, режим копирования завершается и сервер возвращается в режим обработки команд. В случае обнаружения ошибки на стороне сервера в режиме двустороннего копирования, сервер выдаёт сообщение ErrorResponse, пропускает следующие сообщения клиента, пока не будет получено сообщение Sync, а затем выдаёт ReadyForQuery и возвращается к обычной обработке. Клиент должен воспринимать получение ErrorResponse как завершение двустороннего копирования; в этом случае сообщение CopyDone посылаться не должно. За дополнительной информацией о разделе протокола, управляющем двусторонним копированием, обратитесь к Разделу 53.4.

Сообщения CopyInResponse, CopyOutResponse и CopyBothResponse содержат поля, из которых клиент может узнать количество столбцов в строке и код формата для каждого столбца. (В текущей реализации для всех столбцов в заданной операции `COPY` устанавливается один формат, но в конструкции сообщения это не заложено.)

53.2.7. Асинхронные операции

Возможны ситуации, в которых сервер будет отправлять клиенту сообщения, не предполагаемые потоком команд в текущем режиме. Клиенты должны быть готовы принять эти сообщения в любой момент, даже не в процессе выполнения запроса. Как минимум, следует проверять такие сообщения, прежде чем начинать читать ответ на запрос.

Сообщения NoticeResponse могут выдаваться вследствие внешней активности; например, если администратор инициирует «быстрое» отключение баз данных, сервер отправит NoticeResponse, сигнализирующее об этом факте, прежде чем закрывать соединение. Соответственно, клиенты должны быть готовы всегда принять и вывести сообщения NoticeResponse, даже когда соединение фактически простаивает.

Сообщения ParameterStatus будут выдаваться всякий раз, когда меняется действующее значение одного из параметров, об изменении которых, по мнению сервера, должен знать клиент. Чаще всего это происходит в ответ на SQL-команду `SET`, выполняемую клиентом и в таком случае это сообщение по сути синхронно — но состояние параметров может меняться и когда администратор изменяет файл конфигурации, а затем посылает серверу сигнал SIGHUP. Также, если действие команды `SET` отменяется, клиенту передаётся сообщение ParameterStatus, в котором отражается текущее значение параметра.

В настоящее время есть жёстко зафиксированный набор параметров, при изменении которых выдаётся ParameterStatus:

`application_name`
`client_encoding`
`DateStyle`
`default_transaction_read_only`
`in_hot_standby`
`is_superuser`
`scram_iterations`
`server_encoding`
`server_version`
`session_authorization`

`integer_datetimes`	`standard_conforming_strings`
`IntervalStyle`	`TimeZone`

(`default_transaction_read_only` и `in_hot_standby` не отслеживались до версии 14; `scram_iterations` — до версии 16.) Заметьте, что `server_version`, `server_encoding` и `integer_datetimes` — это псевдопараметры, которые не могут меняться после запуска сервера. Этот набор может быть изменён в будущем или даже будет настраиваемым. Соответственно, клиент может просто игнорировать сообщения ParameterStatus для параметров, которые ему неизвестны или не представляют интереса.

Если клиент выполняет команду `LISTEN`, сервер будет передавать ему сообщения NotificationResponse (не путайте с NoticeResponse!), когда для канала с тем же именем затем будет выполняться команда `NOTIFY`.

> **Примечание**
>
> В настоящее время, сообщение NotificationResponse может быть передано только вне транзакции, так что оно не может оказаться в середине серии ответов на команду, хотя может поступить сразу после ReadyForQuery. Однако полагаться на это при проектировании логики клиента неразумно. Лучше разработать код так, чтобы NotificationResponse могло быть принято в любой фазе протокола.

53.2.8. Отмена выполняющихся запросов

В процессе обработки запроса клиент может запросить отмену этого запроса. Запрос отмены не передаётся серверу непосредственно через открытое соединение из соображений эффективности: мы не хотим, чтобы сервер постоянно проверял поступающие от клиента сообщения в процессе выполнения запроса. Запросы отмены должны быть относительно редкими, поэтому мы выбрали для них не самый простой путь во избежание негативного влияния на обычный режим работы.

Чтобы запросить отмену запроса, клиент должен установить новое подключение к серверу и отправить ему сообщение CancelRequest, вместо StartupMessage, обычно передаваемого при установлении нового подключения. Сервер обработает полученную команду и закроет это подключение. По соображениям безопасности сервер не отвечает непосредственно на сообщение с запросом отмены.

Сообщение CancelRequest обрабатывается, только если оно содержит те же ключевые данные (PID и секретный ключ), что были переданы клиенту при запуске. Если PID и секретный ключ в запросе соответствуют данным выполняющегося в данный момент обслуживающего процесса, обработка текущего запроса в нём прерывается. (В существующей реализации это осуществляется путём передачи специального сигнала данному обслуживающему процессу.)

Сигнал отмены может подействовать, а может и не подействовать (например, если он поступит после того, как сервер завершит обработку запроса). Если отмена действительно происходит, текущая команда прерывается досрочно с сообщением об ошибке.

Вследствие такой организации, объясняемой соображениями безопасности и эффективности, клиент не может непосредственно определить, был ли запрос отмены успешен. Он должен продолжать ожидать ответа сервера на исходный запрос. Запрос отмены просто увеличивает вероятность того, что текущий запрос завершится быстрее, как и вероятность того, что он будет завершён не успешно, а с ошибкой.

Так как запрос отмены передаётся серверу через новое подключение к серверу, а не через обычный канал связи клиент-сервер, такие запросы могут выдавать любые процессы, а не только клиентский процесс, запрос которого требуется отменить. Это может дать дополнительную гибкость при построении многопроцессных приложений. Это также представляет собой угрозу безопасности, так как попытаться отменить запросы могут и неавторизованные пользователи. Для ликвидации этой угрозы в запросах отмены требуется передавать динамически генерируемый секретный ключ.

53.2.9. Завершение

Обычная процедура мягкого завершения заключается в том, что клиент отправляет сообщение Terminate и немедленно закрывает соединение. Получая это сообщение, обслуживающий процесс закрывает соединение и завершается.

В редких случаях (например, при отключении базы данных по команде администратора) обслуживающий процесс может отключиться, даже если клиент не запрашивает этого. В таких случаях, перед тем, как закрыть соединение, этот процесс пытается передать сообщение с ошибкой или уведомлением, в котором будет указана причина отключения.

Другие сценарии завершения возникают с различными вариантами отказа, например, при критическом сбое с дампом памяти на одной или другой стороне, при потере канала соединения, потере синхронизации по границам сообщений и т. д. Если клиент или сервер обнаруживает, что соединение было неожиданно закрыто, он должен очистить ресурсы и завершиться. Клиент при этом может запустить новый обслуживающий процесс, переподключившись к серверу, если он сам хочет продолжать работу. Закрывать соединение также рекомендуется при получении сообщений нераспознанного типа, так как это может быть признаком потери синхронизации по границам сообщений.

При штатном или нештатном завершении сеанса любая открытая транзакция откатывается, а не фиксируется. Однако следует заметить, что при отключении клиента в процессе обработки запроса, отличного от `SELECT`, обслуживающий процесс вероятнее всего завершит запрос, прежде чем заметит отключение. Если запрос выполняется не в блоке транзакции (вне последовательности `BEGIN` ... `COMMIT`), его результаты могут быть зафиксированы до того, как будет обнаружено отключение.

53.2.10. Защита сеанса с SSL

Если Postgres Pro был собран с поддержкой SSL, взаимодействие клиента с сервером может быть зашифровано с применением SSL. Это обеспечивает защиту на уровне канала в среде, где злоумышленники могут перехватить трафик сеанса. За дополнительными сведениями о шифровании трафика сеансов Postgres Pro с использованием SSL обратитесь к Разделу 17.9.

Чтобы начать сеанс с SSL-шифрованием, клиент передаёт серверу вместо StartupMessage сообщение SSLRequest. В ответ сервер передаёт один байт, содержащий символ `S` или `N`, показывающий, что он желает, либо не желает включать SSL, соответственно. Если клиент не удовлетворён ответом, он может закрыть соединение на этом этапе. Чтобы продолжить установление соединения после получения `S`, он выполняет начальное согласование SSL с сервером (не описывается здесь, так как относится к протоколу SSL). Если эта процедура выполняется успешно, он продолжает соединение, передавая обычное сообщение StartupMessage. При этом StartupMessage и все последующие данные будут защищены SSL-шифрованием. Чтобы продолжить после получения `N`, клиент может передать обычное сообщение StartupMessage и дальше взаимодействовать с сервером без шифрования. (Клиент также может выдать сообщение GSSENCRequest после получения `N` и попытаться использовать шифрование GSSAPI вместо SSL.)

Клиент также должен быть готов обработать сообщение ErrorMessage, полученное от сервера в ответ на SSLRequest. Клиент не должен показывать эту ошибку пользователю/в приложении, поскольку сервер не был аутентифицирован (*CVE-2024-10977*). В этом случае соединение должно быть закрыто, но клиент может решить открыть новое соединение, не запрашивая SSL.

Когда SSL-шифрование может быть включено, ожидается, что сервер передаст только один байт, содержащий символ `S`, а затем будет ожидать от клиента начала согласования SSL. Если на этом этапе доступны для чтения дополнительные байты, это может означать, что злоумышленник пытается выполнить атаку с переполнением буфера (*CVE-2021-23222*). Код на стороне клиента должен либо прочитать ровно один байт, а затем передать сокет библиотеке SSL, либо, если были прочитаны дополнительные байты, считать это нарушением протокола.

Подобным образом, сервер ожидает, что клиент не будет начинать согласование SSL, пока не получит ответ от сервера на SSL-запрос в виде одного байта. Если клиент начнет согласование сразу, не дожидаясь ответа сервера, задержка соединения может сократиться на один цикл обмена данными. Однако мгновенное согласование может привести к невозможности управлять сценарием, когда сервер отправляет отрицательный ответ на SSL-запрос. В этом случае вместо того, чтобы использовать аутентификацию GSSAPI, установить незашифрованное соединение или выдать ошибку протокола, сервер просто отключится.

Начальный запрос SSLRequest может также передаваться при установлении соединения, открываемого для передачи сообщения CancelRequest.

Существует ещё один альтернативный способ инициировать SSL-шифрование. Сервер распознаёт соединения, для которых незамедлительно начинается согласование SSL без обработки пакетов SSLRequest. После установления соединения по протоколу SSL сервер будет ожидать получения обычного пакета на запрос запуска и продолжит согласование по зашифрованному соединению. В этом случае другие запросы на шифрование отклоняются. Этот способ не считается предпочтительным для инструментов общего назначения, поскольку при его использовании не получится согласовать наиболее подходящее шифрование соединения или управлять незашифрованными соединениями. Однако он может быть полезен в средах одновременного управления сервером и клиентом. В этом случае он позволяет избежать дополнительного цикла обмена данными и, соответственно, сократить задержку соединения, а также использовать сетевые инструменты, которые функционируют, используя стандартные SSL-соединения. При использовании таких соединений необходимо, чтобы клиент применял расширение ALPN, описанное в документе *RFC 7301*, для защиты от атак, направленных на нарушение работы протокола. Согласно *реестру идентификаторов протоколов TLS ALPN IANA*, идентификатор протокола Postgres Pro — «postgresql».

Так как в самом протоколе не предусмотрено принудительное включение SSL-шифрования сервером, администратор может настроить сервер так, чтобы в качестве дополнительного условия при проверке подлинности клиента он не принимал незашифрованные сеансы.

53.2.11. Защита сеанса с GSSAPI

Если Postgres Pro был собран с поддержкой GSSAPI, взаимодействие клиента с сервером может быть зашифровано с применением GSSAPI. Это обеспечивает защиту на уровне канала в среде, где злоумышленники могут перехватить трафик сеанса. За дополнительными сведениями о шифровании трафика сеансов Postgres Pro с использованием GSSAPI обратитесь к Разделу 17.10.

Чтобы установить соединение, зашифрованное GSSAPI, клиент вначале посылает сообщение GSSENCRequest, а не StartupMessage. Сервер в ответ передаёт один байт с буквой `G` или `N`, показывающей соответственно, желает ли он использовать шифрование GSSAPI или нет. Клиент может закрыть соединение в этот момент, если ответ его не устраивает. Чтобы продолжить после варианта `G`, используя GSSAPI на уровне C в соответствии со стандартом *RFC 2744* или равнозначным, выполните инициализацию GSSAPI, вызывая `gss_init_sec_context()` в цикле и отправляя результат на сервер, сначала без входных данных, а затем для всех поступающих от сервера данных, пока они не закончатся. Передавая результаты `gss_init_sec_context()` серверу, добавьте перед сообщением его длину в виде четырёхбайтового целого в сетевом порядке байтов. Чтобы продолжить после получения `N`, клиент может передать обычное сообщение StartupMessage и дальше не использовать шифрование. (Клиент также может выдать сообщение SSLRequest после получения `N` и попытаться использовать шифрование SSL вместо GSSAPI.)

Клиент также должен быть готов обработать сообщение ErrorMessage, полученное от сервера в ответ на GSSENCRequest. Клиент не должен показывать эту ошибку пользователю/в приложении, поскольку сервер не был аутентифицирован (*CVE-2024-10977*). В этом случае соединение должно быть закрыто, но клиент может решить открыть новое соединение, не запрашивая шифрование GSSAPI.

Когда GSSAPI-шифрование может быть включено, ожидается, что сервер передаст только один байт, содержащий символ `G`, а затем будет ожидать от клиента согласование GSSAPI. Если на этом этапе доступны для чтения дополнительные байты, это может означать, что злоумышленник пытается выполнить атаку с переполнением буфера (*CVE-2021-23222*). Код на стороне клиента должен либо прочитать ровно один байт, а затем передать сокет библиотеке GSSAPI, либо, если были прочитаны дополнительные байты, считать это нарушением протокола.

Начальный запрос GSSENCRequest может также передаваться при установлении соединения, открываемого для передачи сообщения CancelRequest.

Когда шифрование GSSAPI установлено, используйте `gss_wrap()` для шифрования обычного сообщения StartupMessage и всех последующих данных. Перед собственно зашифрованным результатом добавьте его длину в виде четырёхбайтового целого в сетевом порядке байтов. Учтите, что сервер будет принимать от клиента зашифрованные пакеты, только если они меньше 16 КБ; чтобы определить, укладывается ли зашифрованное сообщение в это ограничение, клиенту следует использовать функцию `gss_wrap_size_limit()`, а чтобы разбить большие сообщения на части — последовательно вызывать `gss_wrap()`. Сегменты незашифрованных данных обычно имеют размер 8 КБ, поэтому зашифрованные пакеты оказываются немного больше 8 КБ, но вполне умещаются в 16 КБ. Можно ожидать, что сервер также не будет передавать клиенту зашифрованные пакеты размером больше 16 КБ.

Так как в самом протоколе не предусмотрено принудительное включение GSSAPI-шифрования сервером, администратор может настроить сервер так, чтобы в качестве дополнительного условия при проверке подлинности клиента он не принимал незашифрованные сеансы.

53.3. Аутентификация SASL

SASL — это инфраструктура аутентификации для протоколов, ориентированных на соединения. На данный момент Postgres Pro реализует два механизма SASL: SCRAM-SHA-256 и SCRAM-SHA-256-PLUS, а в будущем могут появиться и другие. Далее описывается, как в принципе осуществляется аутентификация SASL, а в следующем подразделе более подробно рассматриваются SCRAM-SHA-256 и SCRAM-SHA-256-PLUS.

Поток сообщений аутентификации SASL

1. Чтобы начать обмен по схеме аутентификации SASL, сервер передаёт сообщение AuthenticationSASL. Оно содержит список механизмов аутентификации SASL, с которыми может работать сервер, в порядке предпочтений сервера.
2. Клиент выбирает один из поддерживаемых механизмов из списка и передаёт серверу сообщение SASLInitialResponse. Это сообщение содержит имя выбранного механизма и может содержать «Начальный ответ клиента», если это использует выбранный механизм.
3. За этим следует одно или нескольких сообщений вызова со стороны сервера и ответов со стороны клиента. Все вызовы сервер передаёт в сообщениях AuthenticationSASLContinue, а клиент отвечает на них сообщениями SASLResponse. Частные детали сообщений зависят от конкретного механизма.
4. Наконец, когда обмен аутентификационной информацией заканчивается успешно, сервер передаёт сообщение AuthenticationSASLFinal и сразу за ним сообщение AuthenticationOk. В сообщении AuthenticationSASLFinal передаются дополнительные данные от сервера клиенту, содержимое которых определяется выбранным механизмом аутентификации. Если механизм аутентификации не требует передавать дополнительные данные в завершение обмена, сообщение AuthenticationSASLFinal опускается.

В случае ошибки сервер может прервать процесс аутентификации на любом этапе и передать сообщение ErrorMessage.

53.3.1. Аутентификация SCRAM-SHA-256

На данный момент реализованы два механизма SASL: `SCRAM-SHA-256` и его вариация со связыванием каналов, `SCRAM-SHA-256-PLUS`. Они подробно описываются в *RFC 7677* и *RFC 5802*.

Когда в Postgres Pro задействуется SCRAM-SHA-256, сервер игнорирует имя пользователя, которое клиент передаёт в `client-first-message`. Вместо этого используется имя, переданное ранее в стартовом сообщении. Согласно спецификации SCRAM, имя пользователя должно быть в UTF-8, но Postgres Pro поддерживает разные кодировки символов, и значит, имя пользователя Postgres Pro не всегда будет представимо в UTF-8.

В спецификации SCRAM говорится, что пароль также должен передаваться в UTF-8 и обрабатываться алгоритмом *SASLprep*. Однако Postgres Pro не требует, чтобы пароль представлялся в UTF-8. Когда устанавливается пароль пользователя, он обрабатывается алгоритмом SASLprep как пароль в UTF-8, вне зависимости от фактической кодировки. Однако если он представлен недопустимой для UTF-8 последовательностью байтов либо содержит комбинации байтов UTF-8, которые не принимает алгоритм SASLprep, это не будет считаться ошибкой — при аутентификации будет использоваться исходный пароль, без обработки SASLprep. Это позволяет нормализовать пароли, представленные в UTF-8, и при этом использовать пароли не в UTF-8, а также не требует, чтобы система знала, в какой кодировке задан пароль.

Связывание каналов поддерживается в Postgres Pro при сборке с использованием SSL. Для SCRAM со связыванием каналов в качестве имени механизма SASL выбрано `SCRAM-SHA-256-PLUS`. Postgres Pro использует тип связывания `tls-server-end-point`.

В SCRAM без связывания каналов сервер выбирает случайное число, которое передаётся клиенту для смешивания с введённым пользователем паролем и получения передаваемого в ответ хеша. Хотя это препятствует повторному воспроизведению пароля в последующем сеансе, поддельный сервер между настоящим сервером и клиентом может прозрачно передать случайное число сервера и затем успешно пройти аутентификацию.

SCRAM со связыванием каналов позволяет предотвратить такие атаки посредника, подмешивая подпись сертификата сервера в передаваемый хеш пароля. Хотя поддельный сервер может повторить передачу сертификата настоящего сервера, у него не будет доступа к закрытому ключу, соответствующему этому сертификату, поэтому он не сможет подтвердить, что является его владельцем, и, как следствие, установить SSL-соединение.

Пример

1. Сервер передаёт сообщение AuthenticationSASL. Оно содержит список механизмов аутентификации SASL, с которыми может работать сервер. Этот список будет включать `SCRAM-SHA-256-PLUS` и `SCRAM-SHA-256`, если сервер собран с поддержкой SSL, а иначе — только последнее значение.
2. Клиент в ответ передаёт сообщение SASLInitialResponse, информирующее о выбранном механизме, `SCRAM-SHA-256` или `SCRAM-SHA-256-PLUS`. (Клиент волен выбрать любой механизм, но для большей безопасности следует выбирать вариацию со связыванием каналов, если он это поддерживает.) В поле «Начальный ответ клиента» это сообщение содержит данные SCRAM `client-first-message`. В `client-first-message` содержится тип связывания каналов, выбранный клиентом.
3. Сервер передаёт сообщение AuthenticationSASLContinue, содержащее данные SCRAM `server-first-message`.
4. Клиент передаёт сообщение SASLResponse, содержащее данные SCRAM `client-final-message`.
5. Сервер передаёт сообщение AuthenticationSASLFinal, содержащее данные SCRAM `server-final-message`, и сразу за ним сообщение AuthenticationOk.

53.4. Протокол потоковой репликации

Чтобы инициировать потоковую репликацию, клиент передаёт в стартовом сообщении параметр `replication`. Логическое значение `true` (или `on`, `yes`, `1`) этого параметра указывает обслуживающему процессу перейти в режим передатчика данных физической репликации. В этом режиме вместо SQL-операторов клиент может выдавать только ограниченный набор команд репликации, показанный ниже.

Если параметр `replication` имеет значение `database`, обслуживающий процесс должен перейти в режим передатчика данных логической репликации. При этом выполняется подключение к базе данных, заданной в параметре `dbname`. В режиме логической репликации могут выполняться как команды репликации, показанные ниже, так и обычные SQL-команды.

В режиме передачи данных физической или логической репликации может использоваться только простой протокол запросов.

Для тестирования команд репликации вы можете установить соединение для репликации, запустив psql или другую программу на базе libpq со строкой подключения, включающей параметр `replication`, например так:

```
psql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

Однако часто полезнее использовать pg_receivewal (для физической репликации) или pg_recvlogical (для логической).

Команды репликации записываются в журнал работы сервера, когда включён параметр log_replication_commands.

В режиме репликации принимаются следующие команды:

`IDENTIFY_SYSTEM`

Запрашивает идентификационные данные сервера. Сервер возвращает набор результатов с одной строкой, содержащей четыре поля:

`systemid` (`text`)

Уникальный идентификатор системы, идентифицирующий кластер. По нему можно определить, что базовая резервная копия, из которой инициализировался резервный сервер, получена из того же кластера.

`timeline` (`int8`)

Идентификатор текущей линии времени. Также полезен для того, чтобы убедиться, что резервный сервер согласован с главным.

`xlogpos` (`text`)

Текущее положение сохранённых данных в WAL. Позволяет узнать, с какой позиции в журнале предзаписи может начаться потоковая передача.

`dbname` (`text`)

Подключённая база данных или NULL.

`SHOW` *имя*

Запрашивает у сервера текущее значение параметра времени выполнения. Эта команда подобна SQL-команде SHOW.

имя

Имя параметра времени выполнения. Доступные параметры описаны в Главе 18.

`TIMELINE_HISTORY` *лин_врем*

Запрашивает с сервера файл истории для линии времени *лин_врем*. Сервер возвращает набор результатов в одной строке, содержащей два поля. Эти поля обозначены как имеющие тип `text`, но фактически они содержат просто байты, не в текстовой кодировке:

`filename` (`text`)

Имя файла с историей линии времени, например `00000002.history`.

`content` (`text`)

Содержимое файла с историей линией времени.

`CREATE_REPLICATION_SLOT` *`имя_слота`* [`TEMPORARY`] { `PHYSICAL` | `LOGICAL` *`модуль_вывода`* } [(*`параметр`* [, ...])]

Создаёт слот физической или логической репликации. Слоты репликации описаны подробно в Подразделе 25.2.6.

`имя_слота`

Имя создаваемого слота. Заданное имя должно быть допустимым для слота репликации (см. Подраздел 25.2.6.1).

`модуль_вывода`

Имя модуля вывода, применяемого для логического декодирования (см. Раздел 47.6).

`TEMPORARY`

Это указание отмечает, что данный слот репликации является временным. Временные слоты не сохраняются на диске и автоматически удаляются при ошибке или завершении сеанса.

Поддерживаются следующие параметры:

`TWO_PHASE [` *`boolean`* `]`

Если параметр включён (по умолчанию он отключён), этот слот логической репликации поддерживает декодирование двухфазной фиксации. С этим параметром декодируются и передаются серверу команды, связанные с двухфазной фиксацией: `PREPARE TRANSACTION`, `COMMIT PREPARED` и `ROLLBACK PREPARED`. Транзакция будет декодироваться и передаваться во время `PREPARE TRANSACTION`.

`RESERVE_WAL [` *`boolean`* `]`

Если параметр включён (по умолчанию он отключён), этот слот физической репликации резервирует WAL немедленно. Без этого указания WAL резервируется только при подключении клиента потоковой репликации.

`SNAPSHOT { 'export' | 'use' | 'nothing' }`

Эти указания выбирают, что делать со снимком, создаваемым при инициализации логического слота. С указанием `'export'`, подразумеваемым по умолчанию, этот снимок будет экспортироваться для использования в других сеансах. Это указание нельзя использовать внутри транзакции. С указанием `'use'` снимок будет использоваться для текущей транзакции, в которой выполняется команда. Это указание должно использоваться в транзакции, при этом команда `CREATE_REPLICATION_SLOT` должна быть первой в этой транзакции. Наконец, с `'nothing'` снимок будет использоваться только для логического декодирования в обычном режиме, и ничего больше с ним делать нельзя.

`FAILOVER [` *`boolean`* `]`

Если параметр имеет значение true, слот может синхронизироваться с резервными серверами, что позволяет возобновлять логическую репликацию после аварийного переключения узлов. Значение по умолчанию — false.

В ответ на эту команду сервер передаст набор результатов с одной строкой, содержащей следующие поля:

`slot_name` (`text`)

Имя создаваемого слота репликации.

`consistent_point` (`text`)

Позиция в WAL, в которой слот достиг согласованного состояния. Это самая ранняя позиция, с которой может начаться трансляция через этот слот репликации.

`snapshot_name` (`text`)

Идентификатор снимка, экспортированного командой. Этот снимок действителен до тех пор, пока через это соединение не будет выполнена следующая команда или соединение не будет закрыто. Null, если созданный слот — физический.

`output_plugin` (`text`)

Имя модуля вывода, используемого созданным слотом репликации. Null, если созданный слот — физический.

`CREATE_REPLICATION_SLOT` *имя_слота* [`TEMPORARY`] { `PHYSICAL` [`RESERVE_WAL`] | `LOGICAL` *модуль_вывода* [`EXPORT_SNAPSHOT` | `NOEXPORT_SNAPSHOT` | `USE_SNAPSHOT` | `TWO_PHASE`] }

Такой вариант записи команды `CREATE_REPLICATION_SLOT` по-прежнему поддерживается для совместимости со старыми версиями.

`ALTER_REPLICATION_SLOT` *имя_слота* (*параметр* [, ...])

Изменяет определение слота репликации. Слоты репликации подробно описаны в Подразделе 25.2.6. Эта команда сейчас поддерживается только для слотов логической репликации.

имя_слота

Имя изменяемого слота. Заданное имя должно быть действительным для слота репликации (см. Подраздел 25.2.6.1).

Поддерживается следующий параметр:

`FAILOVER [` *boolean* `]`

Если параметр имеет значение true, слот может синхронизироваться с резервными серверами, что позволяет возобновлять логическую репликацию после аварийного переключения узлов.

`READ_REPLICATION_SLOT` *имя_слота*

Считывает информацию о слоте репликации. Возвращает кортеж со значениями `NULL`, если указанный слот репликации не существует. Эта команда поддерживается только для слотов физической репликации.

В ответ на эту команду сервер вернёт набор результатов с одной строкой, содержащей следующие поля:

`slot_type` (`text`)

Тип слота репликации: `physical` или `NULL`.

`restart_lsn` (`text`)

`restart_lsn` слота репликации.

`restart_tli` (`int8`)

Идентификатор линии времени, связанной с `restart_lsn`, в текущей истории линии времени.

`START_REPLICATION` [`SLOT` *имя_слота*] [`PHYSICAL`] *XXX/XXX* [`TIMELINE` *лин_врем*]

Указывает серверу начать потоковую передачу WAL, начиная с позиции *XXX/XXX* в WAL. Если указывается параметр `TIMELINE`, передача начинается на линии времени *лин_врем*, иначе выбирается текущая линия времени сервера. Сервер может вернуть в ответ ошибку, например, если запрошенный сегмент WAL уже потерян. Если проблем не возникает, сервер возвращает сообщение CopyBothResponse, а затем начинает передавать поток WAL клиенту.

Если в параметрах передаётся *`имя_слота`*, сервер будет отражать состояние репликации в этом слоте и отслеживать, какие сегменты, а если включён режим `hot_standby_feedback`, то и в каких транзакциях, всё ещё нужны этому резервному серверу.

Если клиент запрашивает не последнюю, но существующую в истории сервера линию времени, сервер будет передавать весь WAL на этой линии времени, начиная с запрошенной стартовой точки до момента, когда сервер переключился на другую линию времени. Если клиент запрашивает передачу с начальной позицией точно в конце старой линии времени, сервер полностью пропускает режим COPY.

После передачи всех записей WAL на линии времени, не являющейся текущей, сервер завершает потоковую передачу, выходя из режима копирования. Когда клиент подтверждает завершение передачи, также выходя из режима копирования, сервер возвращает набор результатов в одной строке с двумя столбцами, сообщая таким образом о следующей линии времени в истории сервера. В первом столбце передаётся идентификатор следующей линии времени (типа `int8`), а во втором — позиция в WAL, в которой произошло переключение (типа `text`). Обычно в этой же позиции завершается передача потока WAL, но возможны исключения, когда сервер может передавать записи WAL из старой линии времени, которые он сам ещё не воспроизвёл до переключения. Наконец сервер передаёт два сообщения CommandComplete (одно говорит о завершении CopyData, а второе — о завершении самой команды `START_REPLICATION`), после чего он готов принять следующую команду.

Данные WAL передаются в серии сообщений CopyData; за подробностями обратитесь к Разделу 53.6 и Разделу 53.7. (Это позволяет перемежать их с другой информацией; в частности, сервер может передать сообщение ErrorResponse, если он столкнулся с проблемами, уже начав передачу потока.) Полезная нагрузка каждого сообщения CopyData от сервера к клиенту содержит данные в одном из следующих форматов:

XLogData (B) — данные журнала транзакций

Byte1('w')

Указывает, что в этом сообщении передаются данные WAL.

Int64

Начальная точка данных WAL в этом сообщении.

Int64

Текущее положение конца WAL на сервере.

Int64

Показания системных часов сервера в момент передачи, в микросекундах с полуночи 2000-01-01.

Byte*n*

Фрагмент потока данных WAL.

Одна запись WAL никогда не разделяется на два сообщения XLogData. Когда запись WAL пересекает границу страницы WAL, и таким образом от неё уже оказывается отделена продолжающая запись, её можно разделить на сообщения по границе страницы. Другими словами, первая основная запись WAL и продолжающие её записи могут быть переданы в различных сообщениях XLogData.

Primary keepalive message (B) — Сообщение об активности ведущего

Byte1('k')

Указывает, что это сообщение об активности отправителя.

Int64

Текущее положение конца WAL на сервере.

Int64

Показания системных часов сервера в момент передачи, в микросекундах с полуночи 2000-01-01.

Byte1

Значение 1 означает, что клиент должен ответить на это сообщение как можно скорее, во избежание отключения по тайм-ауту. Со значением 0 это не требуется.

Принимающий процесс может передавать ответы отправителю в любое время, используя один из следующих форматов данных (также в полезной нагрузке сообщения CopyData):

Standby status update (F) — Обновление состояния резервного сервера

Byte1('r')

Указывает, что это сообщение передаёт обновлённое состояние получателя.

Int64

Положение следующего за последним байтом WAL, полученным и записанным на диск на резервном сервере.

Int64

Положение следующего за последним байтом WAL, сохранённым на диске на резервном сервере.

Int64

Положение следующего за последним байтом WAL, применённым на резервном сервере.

Int64

Показания системных часов клиента в момент передачи, в микросекундах с полуночи 2000-01-01.

Byte1

Если содержит 1, клиент запрашивает от сервера немедленный ответ на это сообщение. Так клиент может запросить отклик сервера и проверить, продолжает ли функционировать соединение.

Hot standby feedback message (F) — Сообщение обратной связи горячего резерва

Byte1('h')

Указывает, что это сообщение обратной связи горячего резерва.

Int64

Показания системных часов клиента в момент передачи, в микросекундах с полуночи 2000-01-01.

Int32

Текущее глобальное значение `xmin` данного резервного сервера, не учитывающее `catalog_xmin` всех слотов репликации. Если и это значение, и следующее `catalog_xmin`, равны 0, это воспринимается как уведомление о том, что через данное подключение больше не будут передаваться сообщения обратной связи горячего резерва. Последующие ненулевые сообщения могут возобновить работу механизма обратной связи.

Int32

Эпоха глобального идентификатора транзакции `xmin` на резервном сервере.

Int32

Наименьшее значение `catalog_xmin` для всех слотов репликации на резервном сервере. Значение 0 показывает, что на резервном сервере нет `catalog_xmin`, либо обратная связь горячего резерва отключена.

Int32

Эпоха идентификатора транзакции `catalog_xmin` на резервном сервере.

`START_REPLICATION` `SLOT` *`имя_слота`* `LOGICAL` *`XXX/XXX`* [(*`имя_параметра`* [*`значение_параметра`*] [, ...])]

Указывает серверу начинать потоковую передачу WAL для логической репликации с позиции *`XXX/XXX`* в WAL или с `confirmed_flush_lsn` слота (см. Раздел 52.19), если это значение больше. Таким образом клиентам не приходится обновлять локальный статус LSN, если не нужно обрабатывать данные. Однако в случаях, когда репликация начинается не с запрошенного LSN, некоторые ошибки на стороне клиента могут остаться незамеченными. Поэтому прежде чем передавать `START_REPLICATION`, имеет смысл запросить значение `confirmed_flush_lsn` и убедиться в том, что оно соответствует ожиданиям.

Сервер может вернуть в ответ ошибку, например, если такого слота нет. Если проблем не возникает, сервер возвращает сообщение CopyBothResponse, а затем начинает передавать поток WAL клиенту.

Данные, передаваемые внутри сообщений CopyBothResponse, имеют тот же формат, что описан для команды `START_REPLICATION ... PHYSICAL`, включая два сообщения CommandComplete.

Обработку выводимых данных для передачи выполняет модуль вывода, связанный с выбранным слотом.

`SLOT` *`имя_слота`*

Имя слота, из которого передаются изменения. Это имя является обязательным, оно должно соответствовать существующему логическому слоту репликации, созданному командой `CREATE_REPLICATION_SLOT` в режиме `LOGICAL`.

`XXX/XXX`

Позиция в WAL, с которой должна начаться передача.

`имя_параметра`

Имя параметра, передаваемого модулю вывода логического декодирования для выбранного слота. Параметры, принимаемые стандартным модулем (`pgoutput`), описаны в Разделе 53.5.

`значение_параметра`

Необязательное значение, в форме строковой константы, связываемое с указанным параметром.

`DROP_REPLICATION_SLOT` *`имя_слота`* [`WAIT`]

Удаляет слот репликации, что приводит к освобождению всех зарезервированных для него ресурсов на стороне сервера.

`имя_слота`

Имя слота, подлежащего удалению.

`WAIT`

С этим указанием команда будет ждать, пока активный слот не станет неактивным (по умолчанию в этом случае выдаётся ошибка).

`UPLOAD_MANIFEST`

Загружает манифест копии при подготовке к инкрементальному резервному копированию.

`BASE_BACKUP` [(*`параметр`* [, ...])]

Указывает серверу начать потоковую передачу базовой копии. Система автоматически переходит в режим резервного копирования до начала передачи, и выходит из него после завершения копирования. Эта команда принимает следующие параметры:

`LABEL` *`'метка'`*

Устанавливает метку для резервной копии. Если метка не задана, по умолчанию устанавливается метка `base backup`. Для метки действуют те же правила применения кавычек, что и для стандартных строк SQL при включённым режиме standard_conforming_strings.

`TARGET` *`получатель`*

Указывает серверу, куда передавать резервную копию. Если выбрано значение `client`, используемое по умолчанию, данные резервной копии передаются клиенту. При значении `server` файлы копии записываются на сервере в каталог, заданный параметром `TARGET_DETAIL`. При значении `blackhole` данные никуда не передаются, а просто отбрасываются.

Чтобы установить значение `server`, нужно иметь права суперпользователя или роли `pg_write_server_files`.

`TARGET_DETAIL` *`дополнительная_информация`*

Дополнительная информация о получателе.

Сейчас этот параметр можно использовать, только когда получателем является `server`. Указывает каталог хранения данных на целевом сервере.

`PROGRESS [` *`boolean`* `]`

Если параметр включён (по умолчанию он отключён), запрашивается информация, необходимая для отслеживания прогресса операции. Сервер передаёт в ответ приблизительный размер в заголовке каждого табличного пространства, исходя из которого можно понять, насколько продвинулась передача потока. Для вычисления этого размера анализируются размеры всех файлов ещё до начала передачи, и это может негативно повлиять на производительность — в частности, может увеличиться задержка передачи первых данных. Так как файлы базы данных могут меняться во время резервного копирования, оценка размера не будет точной; размер базы может увеличиться или уменьшиться за время от вычисления этой оценки до передачи актуальных файлов.

`CHECKPOINT { 'fast' | 'spread' }`

Задаёт тип контрольной точки, которая будет выполняться в начале базового копирования. По умолчанию `spread` (протяжённая).

`WAL [` *`boolean`* `]`

Если параметр включён (по умолчанию он отключён), в резервную копию помещаются необходимые сегменты WAL. При этом в подкаталог `pg_wal` архива базового каталога будут помещены все файлы с начала до конца копирования.

`WAIT [` *`boolean`* `]`

Если параметр включён (по умолчанию), при копировании ожидается завершение архивации последнего требуемого сегмента WAL либо выдаётся предупреждение, если архивация WAL не включена. Если отключён, при копировании не будет ни ожидания, ни предупреждений, так что обеспечение наличия требуемого журнала становится задачей клиента.

`COMPRESSION` *`метод`*

Указывает серверу метод сжатия данных резервной копии. На данный момент поддерживаются следующие методы: `gzip`, `lz4` и `zstd`.

`COMPRESSION_DETAIL` *`дополнительная_информация`*

Дополнительная информация для выбранного метода сжатия. Используется только вместе с параметром `COMPRESSION`. Если значение параметра — целое число, оно обозначает уровень сжатия. В противном случае значение должно представлять собой список ключевых слов, разделённых запятыми, в форме *`ключевое_слово`* или *`ключевое_слово=значение`*. На данный момент поддерживаются ключевые слова `level`, `long` и `workers`.

Ключевое слово `level` задаёт уровень сжатия. Для `gzip` уровень сжатия задаётся целым числом от `1` до `9` (по умолчанию `Z_DEFAULT_COMPRESSION` или `-1`), для `lz4` — целым числом от 1 до 12 (по умолчанию `0` для максимально быстрого сжатия), а для `zstd` — целым числом от `ZSTD_minCLevel()` (обычно `-131072`) до `ZSTD_maxCLevel()` (обычно `22`), по умолчанию `ZSTD_CLEVEL_DEFAULT` или `3`.

Ключевое слово `long` включает режим поиска соответствий на большой дистанции для улучшения коэффициента сжатия за счёт повышения нагрузки на память. Этот режим поддерживается только для `zstd`.

Ключевое слово `workers` задаёт число параллельных потоков для сжатия. Сжатие в несколько параллельных потоков поддерживается только для `zstd`.

`MAX_RATE` *`скорость`*

Ограничивает (сдерживает) максимальный объём данных, передаваемый от сервера клиенту за единицу времени. Единица измерения этого параметра — килобайты в секунду. Если задаётся этот параметр, его значение должно быть равно нулю, либо должно находиться в диапазоне от 32 (килобайт/сек) до 1 Гбайта/сек (включая границы). Если передаётся ноль, либо параметр не задаётся, скорость передачи не ограничивается.

`TABLESPACE_MAP [` *`boolean`* `]`

Если параметр включён (по умолчанию он отключён), информация о символических ссылках, существующих в каталоге `pg_tblspc`, включается в файл `tablespace_map`. Файл карты табличных пространств содержит имена всех ссылок, содержащихся в каталоге `pg_tblspc/`, и полный путь для каждой ссылки.

`VERIFY_CHECKSUMS [` *`boolean`* `]`

Если параметр включён (по умолчанию), контрольные суммы проверяются в процессе резервного копирования, когда они присутствуют. При отключении параметра эта проверка опускается.

`MANIFEST` *`параметр_манифеста`*

Когда указывается этот параметр, и он имеет значение `yes` или `force-encode`, вместе с копией создаётся и передаётся манифест копии. Манифест содержит список всех файлов, содержащихся в копии, за исключением файлов WAL, которые могут быть добавлены дополнительно. В нём также сохраняется размер, дата последнего изменения и, возможно, контрольная сумма каждого файла. Значение `force-encode` указывает, что все имена файлов должны кодироваться в шестнадцатеричном виде; по умолчанию кодироваться будут только те имена, которые представлены не байтовыми последовательностями UTF-8. Это значение предназначено в первую очередь для тестирования, чтобы можно было проверить, что клиентские программы могут корректно прочитать закодированные имена. Для совместимости с предыдущими версиями подразумевается значение `MANIFEST 'no'`.

`MANIFEST_CHECKSUMS` *`алгоритм_контрольной_суммы`*

Задаёт алгоритм контрольных сумм, которые будут рассчитываться для всех файлов, описанных в манифесте копии. В настоящее время поддерживаются алгоритмы `NONE` (отсутствует), `CRC32C`, `SHA224`, `SHA256`, `SHA384` и `SHA512`. По умолчанию применяется `CRC32C`.

`INCREMENTAL`

Запрашивает инкрементальную резервную копию. Обязательно выполните команду `UPLOAD_MANIFEST` перед началом базового резервного копирования с этим параметром.

Когда запускается копирование, сервер сначала передаёт два обычных набора результатов, за которыми следуют один или более результатов CopyOutResponse.

В первом обычном наборе результатов передаётся начальная позиция резервной копии, в одной строке с двумя столбцами. В первом столбце содержится стартовая позиция в формате XLogRecPtr, а во втором идентификатор соответствующей линии времени.

Во втором обычном наборе результатов передаётся по одной строке для каждого табличного пространства. Эта строка содержит следующие поля:

`spcoid` (`oid`)

OID табличного пространства либо NULL, если это базовый каталог.

`spclocation` (`text`)

Полный путь к каталогу табличного пространства либо NULL, если это базовый каталог.

`size` (`int8`)

Приблизительный размер табличного пространства (в килобайтах, размером 1024 байта), если была запрошена информация о прогрессе операции; в противном случае NULL.

За вторым обычным набором результатов следует сообщение CopyOutResponse. Полезной нагрузкой каждого последующего сообщения CopyData будет сообщение в одном из следующих форматов:

new archive (B)

Byte1('n')

Указывает, что это сообщение начинает новый архив. Сервер передаст один архив для основного каталога данных и по одному для каждого дополнительного табличного пространства. Данные в архиве представлены в формате tar («формате обмена ustar», описанном в стандарте POSIX 1003.1-2008).

String

Имя файла этого архива.

String

Для основного каталога данных пустая строка. Для других табличных пространств — полный путь к каталогу, для которого был создан архив.

manifest (B)

Byte1('m')

Указывает, что это сообщение начинает манифест копии.

archive or manifest data (B)

Byte1('d')

Указывает, что это сообщение передаёт данные архива или манифеста.

Byte*n*

Байты данных.

progress report (B)

Byte1('p')

Указывает, что в этом сообщении передаётся отчёт о выполнении.

Int64

Объём обработанных данных в текущем табличном пространстве в байтах.

После передачи сообщения CopyOutResponse или всех таких сообщений передаётся обычный набор результатов, в котором содержится конечная позиция копии в WAL, в том же формате, что и стартовая позиция.

Архив tar каталога данных и всех табличных пространств будет содержать все файлы в этих каталогах, будь то файлы Postgres Pro или посторонние файлы, добавленные в эти каталоги. Исключение составляют только следующие файлы:

- `postmaster.pid`
- `postmaster.opts`
- `pg_internal.init` (находится в нескольких каталогах)
- Различные временные файлы и каталоги, создаваемые в процессе работы сервером Postgres Pro, в частности, файлы и каталоги с именами, начинающимися с `pgsql_tmp`, и временные отношения.
- Нежурналируемые отношения, за исключением слоя инициализации, который необходим при восстановлении для пересоздания нежурналируемого отношения (пустого).
- `pg_wal`, включая подкаталоги. Если в резервную копию включаются файлы WAL, в архив входит преобразованная версия `pg_wal`, в которой будут находиться только файлы, необходимые для восстановления копии, но не всё остальное содержимое этого каталога
- `pg_dynshmem`, `pg_notify`, `pg_replslot`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` и `pg_subtrans` копируются как пустые каталоги (даже если это символические ссылки)
- Файлы, кроме обычных файлов и каталогов, например, символические ссылки (кроме вышеупомянутых каталогов), а также файлы специальных устройств и файлы операционных систем, пропускаются (символические ссылки в `pg_tblspc` сохраняются).

Если файловая система сервера поддерживает это, в архив включается информация о владельце, группе и режиме файла.

Во всех вышеперечисленных командах *`значение_параметра`* можно не указывать, если это параметр типа `boolean`. Это равнозначно тому, чтобы указать значение `TRUE`.

53.5. Протокол логической потоковой репликации

В этом разделе описывается протокол логической репликации, регламентирующий поток сообщений, который запускается командой репликации `START_REPLICATION SLOT` *`имя_слота`* `LOGICAL`.

Протокол логической потоковой репликации построен на примитивах протокола физической потоковой репликации.

В логическом декодировании Postgres Pro поддерживаются модули вывода. Стандартный модуль `pgoutput` используется для встроенной логической репликации.

53.5.1. Параметры протокола логической потоковой репликации

Используя команду `START_REPLICATION`, модуль `pgoutput` принимает следующие параметры:

proto_version

Версия протокола. В настоящее время поддерживаются только версии `1`, `2`, `3` и `4`. Необходимо указывать только рабочую версию.

Версия `2`, позволяющая передавать в потоке большие выполняющиеся транзакции, поддерживается только с серверами версии 14 и выше.

Версия `3`, позволяющая передавать фиксацию транзакций в двух фазах, поддерживается только с серверами версии 15 и выше.

Версия `4`, позволяющая параллельно передавать в потоке большие выполняющиеся транзакции, поддерживается только с серверами версии 16 и выше.

publication_names

Список разделённых запятыми имён публикаций, на которые подписывается клиент (будет получать их изменения). Имена отдельных публикаций обрабатываются как стандартные имена объектов и могут так же заключаться в кавычки при необходимости. Требуется указывать хотя бы одно имя публикации.

binary

Логический параметр, указывающий использовать режим двоичной передачи. Этот режим работает быстрее текстового, но чуть менее устойчив.

messages

Логический параметр, который включает отправку сообщений, написанных функцией `pg_logical_emit_message`.

streaming

Параметр, позволяющий включить потоковую передачу текущих транзакций. Возможные значения `off` (по умолчанию), `on` и `parallel`. Значение `parallel` включает отправку дополнительной информации с некоторыми сообщениями, которые будут использоваться для параллельного выполнения. Для включения параметра (`on`) требуется протокол не ниже версии `2`. При указании значения `parallel` требуется протокол не ниже версии `4`.

two_phase

Логический параметр, включающий двухфазные транзакции. Для его включения требуется протокол не ниже версии `3`.

origin

Параметр для отправки изменений по их происхождению. Возможные значения: `none`, чтобы отправлять только те изменения, для которых не указан источник, или `any`, чтобы отправлять изменения независимо от их происхождения. Данный параметр можно использовать, чтобы избежать зацикливаний (бесконечной репликации одних и тех же данных) между узлами репликации.

53.5.2. Сообщения протокола логической репликации

Отдельные сообщения протокола рассматриваются в следующих подразделах. Собственно сообщения описаны в Раздел 53.9.

Все сообщения верхнего уровня начинаются с байта, определяющего тип сообщения. Хотя он представлен в коде символьным типом, это знаковый байт без явно заданной кодировки.

Так как в протоколе потоковой репликации передаётся длина сообщения, нет необходимости указывать длину в заголовках сообщений верхнего уровня.

53.5.3. Поток сообщений протокола логической репликации

За исключением команды `START_REPLICATION` и сообщений о прогрессе воспроизведения, весь информационный поток направлен от сервера к клиенту.

Протокол логической репликации передаёт отдельные транзакции одну за другой. Это значит, что все сообщения между парой сообщений Begin и Commit относятся к одной транзакции. Точно так же к одной транзакции относятся все сообщения между парой сообщений Begin Prepare и Prepare.

Этот протокол также передаёт содержимое больших транзакций по ходу их выполнения между парой сообщений Stream Start и Stream Stop. Последний поток для такой транзакции содержит сообщение Stream Commit или Stream Abort.

Каждая передаваемая транзакция содержит ноль или более сообщений DML (Insert, Update, Delete). В каскадной схеме она может также содержать сообщения Origin. Это сообщение показывает, что транзакция пришла с другого узла в схеме репликации. Так как этим узлом в контексте протокола логической репликации может быть что угодно, единственным идентификатором является его имя. Как воспринимать это имя (если это вообще нужно), определяют нижестоящие узлы. Сообщение Origin всегда передаётся перед всеми остальными сообщениями DML в транзакции.

Каждое DML-сообщение содержит идентификатор (OID) отношения, который указывает на целевое отношение на стороне публикации. Перед первым DML-сообщением для данного OID отношения будет отправлено сообщение Relation, описывающее схему данного отношения. Впоследствии будет отправлено новое сообщение Relation, если определение отношения изменится после отправки последнего сообщения Relation. (В протоколе предполагается, что клиент сможет кешировать метаданные для достаточно большого числа отношений.)

В сообщениях Relation типы столбцов определяются идентификаторами (OID). Для встроенного типа предполагается, что клиент может разрешить этот OID локально, поэтому никакие дополнительные данные не нужны. Однако для любых других типов перед сообщением Relation будет передаваться сообщение Type, связывающее имя типа с определённым OID. Таким образом, клиент, которому нужно однозначно определять типы столбцов в отношении, должен кешировать содержимое сообщений Type, и в случае обнаружения полученного OID в кеше использовать эту информацию, а в противном случае разрешать его локально.

53.6. Типы данных в сообщениях

В этом разделе описываются базовые типы данных, применяемые в сообщениях.

Int*n*(*i*)

Целое число из *n* бит с сетевым порядком байт (наиболее значащий байт первый). Если указано *i*, это поле будет содержать именно указанное значение, в противном случае значение переменное. Например: Int16, Int32(42).

Int*n*[*k*]

Массив из *k* *n*-битовых целых, каждое записывается с сетевым порядком байт. Длина массива *k* всегда определяется по предыдущему полю сообщения, например Int16[M].

String(*s*)

Строка, оканчивающаяся нулём (строка в стиле C). На длину строк ограничение не накладывается. Если указывается *s*, это поле будет содержать именно указанное значение, в противном случае значение переменное. Например: String, String("user").

Примечание

Нет никакого предопределённого ограничения длины строки, которую может возвратить сервер. Поэтому при реализации клиента лучше использовать расширяемый буфер, чтобы он мог принять строку любого размера, умещающуюся в памяти. Если такой возможности нет, прочитайте строку целиком и отбросьте последние символы, не помещающиеся в ваш буфер фиксированного размера.

Byte*n*(*c*)

В точности *n* байт. Если размер поля *n* задаётся не константой, он всегда определяется по предыдущему полю сообщения. Если указывается *c*, оно задаёт точное значение. Например: Byte2, Byte1('\n').

53.7. Форматы сообщений

В этом разделе подробно описывается формат каждого сообщения. Все сообщения помечены символами, обозначающими, какая сторона может их передавать: клиент (F), сервер (B) или обе стороны (F & B). Заметьте, что хотя каждое сообщение включает счётчик байт в начале, формат сообщения разработан так, чтобы конец сообщения можно было найти, не обращаясь к счётчику байт. Это помогает проверять корректность сообщений. (Исключением является сообщение CopyData, так как оно образует часть потока данных; содержимое любого отдельного сообщения CopyData нельзя интерпретировать само по себе.)

AuthenticationOk (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(0)

Показывает, что проверка подлинности прошла успешно.

AuthenticationKerberosV5 (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(2)

Указывает, что требуется проверка подлинности по протоколу Kerberos V5.

AuthenticationCleartextPassword (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(3)

Указывает, что требуется пароль, передаваемый открытым текстом.

AuthenticationMD5Password (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(12)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(5)

Указывает, что требуется пароль, преобразованный в хеш MD5.

Byte4

Значение соли, с которым должен хешироваться пароль.

AuthenticationGSS (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(7)

Указывает, что требуется проверка подлинности на базе GSSAPI.

AuthenticationGSSContinue (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(8)

Указывает, что это сообщение содержит данные GSSAPI или SSPI.

Byte*n*

Данные аутентификации для GSSAPI или SSPI.

AuthenticationSSPI (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(9)

Указывает, что требуется проверка подлинности на базе SSPI.

AuthenticationSASL (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(10)

Указывает, что требуется проверка подлинности на базе SASL.

Тело сообщения содержит список механизмов аутентификации SASL в порядке предпочтений сервера. За последним именем механизма аутентификации должен идти завершающий нулевой байт. Для каждого механизма передаётся:

String

Имя механизма аутентификации SASL.

AuthenticationSASLContinue (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(11)

Указывает, что это сообщение содержит данные вызова SASL.

Byte*n*

Данные SASL, специфичные для применяемого механизма SASL.

AuthenticationSASLFinal (B)

Byte1('R')

Указывает, что это сообщение представляет запрос аутентификации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(12)

Указывает, что аутентификация SASL завершена.

Byte*n*

«Дополнительные данные» результата SASL, специфичные для применяемого механизма SASL.

BackendKeyData (B)

Byte1('K')

Указывает, что это сообщение содержит ключевые данные для отмены запросов. Клиент должен сохранить эти данные, если ему нужна возможность впоследствии выдавать сообщения CancelRequest.

Int32(12)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

PID обслуживающего процесса.

Int32

Секретный ключ обслуживающего процесса.

Bind (F)

Byte1('B')

Указывает, что это сообщение представляет команду Bind.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя целевого портала (пустая строка выбирает безымянный портал).

String

Имя исходного подготовленного оператора (пустая строка выбирает безымянный подготовленный оператор).

Int16

Количество кодов форматов следующих параметров (обозначается ниже символом *C*). Может быть нулевым, что показывает, что параметры отсутствуют или все параметры передаются в формате по умолчанию (текстовом); либо равняться одному, в этом случае указанный один код формата применяется ко всем параметрам; либо может равняться действительному количеству параметров.

Int16[*C*]

Коды форматов параметров. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный).

Int16

Количество следующих значений параметров (может быть нулевым). Оно должно совпадать с количеством параметров, требующихся для запроса.

Затем для каждого параметра идёт следующая пара полей:

Int32

Длина значения параметра, в байтах (само поле длины не считается). Может быть нулевой. В качестве особого значения, -1 представляет значение NULL. В случае с NULL никакие байты значений далее не следуют.

Byte*n*

Значение параметра в формате, определённом соответствующим кодом формата. Переменная *n* задаёт длину значения.

За последним параметром идёт следующее поле:

Int16

Количество кодов формата для следующих столбцов результата (обозначается ниже символом *R*). Может быть нулевым, что показывает, что столбцы результата отсутствуют или для всех столбцов должен использоваться формат по умолчанию (текстовый), либо равняться одному, в этом случае указанный один код формата применяется ко всем столбцам (если они есть), либо может равняться действительному количеству столбцов результата запроса.

Int16[*R*]

Коды форматов столбцов результата. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный).

BindComplete (B)

Byte1('2')

Указывает, что это сообщение, сигнализирующее о завершении Bind.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

CancelRequest (F)

Int32(16)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(80877102)

Код запроса отмены. Это специально выбранное значение содержит `1234` в старших 16 битах и `5678` в младших 16 битах. (Во избежание неоднозначности этот код не должен совпадать с номером версии протокола.)

Int32

PID целевого обслуживающего процесса.

Int32

Секретный ключ целевого обслуживающего процесса.

Close (F)

Byte1('C')

Указывает, что это сообщение представляет команду Close.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte1

'`S`' для закрытия подготовленного оператора, '`P`' для закрытия портала.

String

Имя подготовленного оператора или портала, который должен быть закрыт (пустая строка выбирает безымянный подготовленный оператор или портал).

CloseComplete (B)

Byte1('3')

Указывает, что это сообщение, сигнализирующее о завершении Close.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

CommandComplete (B)

Byte1('C')

Указывает, что это сообщение об успешном завершении команды.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Тег команды. Обычно это одно слово, обозначающее завершённую команду SQL.

Для команды `INSERT` в качестве тега передаётся `INSERT` *`oid число_строк`*, где *`число_строк`* — количество вставленных строк. В поле *`oid`* передавался идентификатор объекта вставленной строки, когда *`число_строк`* было равно 1 и в целевой таблице содержались OID; таким образом, сейчас *`oid`* всегда равняется 0.

Для команды `DELETE` в качестве тега передаётся `DELETE` *`число_строк`*, где *`число_строк`* — количество удалённых строк.

Для команды `UPDATE` в качестве тега передаётся `UPDATE` *`число_строк`*, где *`число_строк`* — количество изменённых строк.

Для команды `MERGE` в качестве тега передаётся `MERGE` *`число_строк`*, где *`число_строк`* — количество вставленных, изменённых или удалённых строк.

Для команды `SELECT` или `CREATE TABLE AS` в качестве тега передаётся `SELECT` *`число_строк`*, где *`число_строк`* — число полученных строк.

Для команды `MOVE` в качестве тега передаётся `MOVE` *число_строк*, где *число_строк* — количество строк, на которое изменилась позиция курсора.

Для команды `FETCH` в качестве тега передаётся `FETCH` *число_строк*, где *число_строк* — количество строк, полученное через курсор.

Для команды `COPY` в качестве тега передаётся `COPY` *число_строк*, где *число_строк* — количество скопированных строк. (Заметьте: число строк выводится, начиная только с PostgreSQL 8.2.)

CopyData (F & B)

Byte1('d')

Указывает, что в этом сообщении передаются данные `COPY`.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte*n*

Данные, образующие часть информационного потока `COPY`. Сообщения от сервера всегда соответствуют отдельным строкам данных, но сообщения, передаваемые клиентами, могут разделять поток произвольным образом.

CopyDone (F & B)

Byte1('c')

Указывает, что это сообщение, сигнализирующее о завершении `COPY`.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

CopyFail (F)

Byte1('f')

Указывает, что это сообщение, сигнализирующее об ошибке операции `COPY`.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Сообщение об ошибке, описывающее причину сбоя операции.

CopyInResponse (B)

Byte1('G')

Указывает, что это сообщение является ответом на запуск входящего копирования. Получив его, клиент начинает передавать данные на вход операции копирования (если клиент не готов к этому, он передаёт сообщение CopyFail).

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int8

Значение 0 указывает, что для всей операции `COPY` применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами-разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат (подобный формату DataRow). За дополнительными сведениями обратитесь к COPY.

Int16

Количество столбцов в копируемых данных (ниже обозначается символом *N*).

Int16[*N*]

Коды формата для каждого столбца. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный). Если общий формат копирования — текстовый, все эти коды должны быть нулевыми.

CopyOutResponse (B)

Byte1('H')

Указывает, что это сообщение является ответом на запуск исходящего копирования. За этим сообщением следуют данные, исходящие со стороны сервера.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int8

Значение 0 указывает, что для всей операции `COPY` применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами-разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат (подобный формату DataRow). За дополнительными сведениями обратитесь к COPY.

Int16

Количество столбцов в копируемых данных (ниже обозначается символом *N*).

Int16[*N*]

Коды формата для каждого столбца. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный). Если общий формат копирования — текстовый, все эти коды должны быть нулевыми.

CopyBothResponse (B)

Byte1('W')

Указывает, что это сообщение является ответом на запуск двустороннего копирования. Это сообщение используется только для потоковой репликации.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int8

Значение 0 указывает, что для всей операции `COPY` применяется текстовый формат (строки разделяются символами новой строки, столбцы разделяются символами-разделителями и т. д.). Значение 1 указывает, что для всей операции копирования применяется двоичный формат (подобный формату DataRow). За дополнительными сведениями обратитесь к COPY.

Int16

Количество столбцов в копируемых данных (ниже обозначается символом *N*).

Int16[*N*]

Коды формата для каждого столбца. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный). Если общий формат копирования — текстовый, все эти коды должны быть нулевыми.

DataRow (B)

Byte1('D')

Указывает, что в этом сообщении передаётся строка данных.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int16

Количество последующих значений столбцов (может быть нулевым).

Затем для каждого столбца идёт следующая пара полей:

Int32

Длина значения столбца, в байтах (само поле длины не считается). Может быть нулевой. В качестве особого значения, -1 представляет значение NULL. В случае с NULL никакие байты значений далее не следуют.

Byte*n*

Значение столбца в формате, определённом соответствующим кодом формата. Переменная *n* задаёт длину значения.

Describe (F)

Byte1('D')

Указывает, что это сообщение представляет команду Describe.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte1

'`S`' для получения описания подготовленного оператора, '`P`' — портала.

String

Имя подготовленного оператора или портала, описание которого запрашивается (пустая строка выбирает безымянный подготовленный оператор или портал).

EmptyQueryResponse (B)

Byte1('I')

Указывает, что это сообщение является ответом на пустую строку запроса. (Это сообщение заменяет CommandComplete.)

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

ErrorResponse (B)

Byte1('E')

Указывает, что это сообщение ошибки.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Тело сообщения состоит из одного или нескольких определённых полей, за которыми в качестве завершающего следует нулевой байт. Поля могут идти в любом порядке. Для каждого поля передаётся:

Byte1

Код, задающий тип поля; ноль обозначает конец сообщения, после которого ничего нет. Типы полей, определённые в настоящее время, перечислены в Разделе 53.8. Так как в будущем могут появиться другие типы полей, клиенты должны просто игнорировать поля нераспознанного типа.

String

Значение поля.

Execute (F)

Byte1('E')

Указывает, что это сообщение представляет команду Execute.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя портала, подлежащего выполнению (пустая строка выбирает безымянный портал).

Int32

Максимальное число строк, которое должно быть возвращено, если портал содержит запрос, возвращающий строки (в противном случае игнорируется). Ноль означает «без ограничения».

Flush (F)

Byte1('H')

Указывает, что это сообщение представляет команду Flush.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

FunctionCall (F)

Byte1('F')

Указывает, что это сообщение представляет вызов функции.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

Задаёт идентификатор объекта вызываемой функции.

Int16

Количество кодов форматов следующих аргументов (обозначается ниже символом *C*). Может быть нулевым, что показывает, что аргументы отсутствуют или все аргументы передаются в формате по умолчанию (текстовом); либо равняться одному, в этом случае указанный один код формата применяется ко всем аргументами, либо может равняться действительному количеству аргументов.

Int16[*C*]

Коды форматов аргументов. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный).

Int16

Задаёт число аргументов, передаваемых функции.

Затем для каждого аргумента идёт следующая пара полей:

Int32

Длина значения аргумента, в байтах (само поле длины не считается). Может быть нулевой. В качестве особого значения, -1 представляет значение NULL. В случае с NULL никакие байты значений далее не следуют.

Byte*n*

Значение аргумента, в формате, определённом соответствующим кодом формата. Переменная *n* задаёт длину значения.

За последним аргументом идёт следующее поле:

Int16

Код формата результата функции. В настоящее время допускается код ноль (текстовый формат) и один (двоичный).

FunctionCallResponse (B)

Byte1('V')

Указывает, что в этом сообщении передаётся результат вызова функции.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

Длина значения результата функции, в байтах (само поле длины не считается). Может быть нулевой. В качестве особого значения, -1 представляет значение NULL. В случае с NULL никакие байты значения далее не следуют.

Byte*n*

Значение результата функции в формате, определённом соответствующим кодом формата. Переменная *n* задаёт длину значения.

GSSENCRequest (F)

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(80877104)

Код запроса GSSAPI Encryption. Это специально выбранное значение содержит `1234` в старших 16 битах и `5680` в младших 16 битах. (Во избежание неоднозначности этот код не должен совпадать с номером версии протокола.)

GSSResponse (F)

Byte1('p')

Обозначает это сообщение как ответ GSSAPI или SSPI. Заметьте, что оно также применяется для ответов SASL и при аутентификации по паролю. Точный тип сообщения можно определить из контекста.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte*n*

Данные сообщения, специфичные для GSSAPI/SSPI.

NegotiateProtocolVersion (B)

Byte1('v')

Указывает, что это сообщение согласования версии протокола.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

Новейшая младшая версия протокола, поддерживаемая сервером, для запрошенной клиентом старшей версии.

Int32

Число параметров протокола, не принятых сервером.

Затем для параметров протокола, не принятых сервером, передаётся:

String

Имя параметра.

NoData (B)

Byte1('n')

Указывает, что это сообщение сигнализирует об отсутствии данных.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

NoticeResponse (B)

Byte1('N')

Указывает, что это сообщение представляет замечание.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Тело сообщения состоит из одного или нескольких определённых полей, за которыми в качестве завершающего следует нулевой байт. Поля могут идти в любом порядке. Для каждого поля передаётся:

Byte1

Код, задающий тип поля; ноль обозначает конец сообщения, после которого ничего нет. Типы полей, определённые в настоящее время, перечислены в Разделе 53.8. Так как в будущем могут появиться другие типы полей, клиенты должны просто игнорировать поля нераспознанного типа.

String

Значение поля.

NotificationResponse (B)

Byte1('A')

Указывает, что это сообщение представляет уведомление.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32

PID обслуживающего процесса, отправляющего уведомление.

String

Имя канала, для которого было выдано уведомление.

String

Строка «сообщения», сопровождающего уведомление.

ParameterDescription (B)

Byte1('t')

Указывает, что это сообщение представляет описание параметра.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int16

Количество параметров для оператора (может быть нулевым).

Затем для каждого параметра передаётся:

Int32

Задаёт идентификатор объекта типа данных параметра.

ParameterStatus (B)

Byte1('S')

Указывает, что в этом сообщении передаётся состояние параметра времени выполнения.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя параметра времени выполнения, состояние которого передаётся.

String

Текущее значение параметра.

Parse (F)

Byte1('P')

Указывает, что это сообщение представляет команду Parse.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя целевого подготовленного оператора (пустая строка выбирает безымянный подготовленный оператор).

String

Строка запроса, которая должна быть разобрана.

Int16

Количество типов параметров (может быть нулевым). Заметьте, что это значение представляет не число параметров, которые могут фигурировать в строке запроса, а число параметров, для которых клиент хочет предопределить типы.

Затем для каждого параметра передаётся:

Int32

Задаёт идентификатор объекта типа данных параметра. Указание нулевого значения равносильно отсутствию указания типа.

ParseComplete (B)

Byte1('1')

Указывает, что это сообщение, сигнализирующее о завершении Parse.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

PasswordMessage (F)

Byte1('p')

Обозначает это сообщение как ответ SASL. Заметьте, что оно также применяется для ответов GSSAPI, SSPI и SASL. Точный тип сообщения можно определить из контекста.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Пароль (зашифрованный, если требуется).

PortalSuspended (B)

Byte1('s')

Указывает, что это сообщение сигнализирует о приостановке портала. Заметьте, что оно выдаётся только при достижении ограничения числа строк, заданного в сообщении Execute.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

Query (F)

Byte1('Q')

Указывает, что это сообщение представляет простой протокол запросов.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Собственно строка запроса.

ReadyForQuery (B)

Byte1('Z')

Определяет тип сообщения. Сообщение ReadyForQuery передаётся, когда сервер готов к новому циклу запросов.

Int32(5)

Длина содержимого сообщения в байтах, включая само поле длины.

Byte1

Индикатор текущего состояния транзакции на сервере. Возможные значения: '`I`', транзакция неактивна (вне блока транзакции), '`T`' в блоке транзакции, либо '`E`' в блоке прерванной транзакции (запросы не будут обрабатываться до завершения блока).

RowDescription (B)

Byte1('T')

Указывает, что это сообщение представляет описание строки.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int16

Задаёт количество полей в строке (может быть нулевым).

Затем для каждого поля передаётся:

String

Имя поля.

Int32

Если поле связано со столбцом определённой таблицы, идентификатор объекта этой таблицы; в противном случае — ноль.

Int16

Если поле связано со столбцом определённой таблицы, номер атрибута для этого столбца; в противном случае — ноль.

Int32

Идентификатор объекта типа данных поля.

Int16

Размер типа данных (см. `pg_type.typlen`). Заметьте, что отрицательные значения показывают, что тип имеет переменную длину.

Int32

Модификатор типа (см. `pg_attribute.atttypmod`). Смысл этого модификатора зависит от типа.

Int16

Код формата, используемого для поля. В настоящее время допускаются коды ноль (текстовый формат) и один (двоичный). В сообщении RowDescription, возвращаемом вариацией Describe для оператора, код формата ещё не известен и всегда будет нулевым.

SASLInitialResponse (F)

Byte1('p')

Обозначает это сообщение как начальный ответ SASL. Заметьте, что оно также применяется для ответов GSSAPI, SSPI и при аутентификации по паролю. Точный тип сообщения определяется из контекста.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

String

Имя механизма аутентификации SASL, выбранного клиентом.

Int32

Длина последующего сообщения «Начальный ответ клиента», специфичного для механизма SASL, или -1, если начального ответа нет.

Byte*n*

«Начальный ответ», специфичный для механизма SASL.

SASLResponse (F)

Byte1('p')

Обозначает это сообщение как ответ SASL. Заметьте, что оно также применяется для ответов GSSAPI, SSPI и при аутентификации по паролю. Точный тип сообщения можно определить из контекста.

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Byte*n*

Данные сообщения, специфичные для механизма SASL.

SSLRequest (F)

Int32(8)

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(80877103)

Код запроса SSL. Это специально выбранное значение содержит `1234` в старших 16 битах и `5679` в младших 16 битах. (Во избежание неоднозначности этот код не должен совпадать с номером версии протокола.)

StartupMessage (F)

Int32

Длина содержимого сообщения в байтах, включая само поле длины.

Int32(196608)

Номер версии протокола. В старших 16 битах задаётся старший номер версии (3 для протокола, описываемого здесь). В младших 16 битах задаётся младший номер версии (0 для протокола, описываемого здесь).

За номером версии протокола следует одна или несколько пар из имени параметра и строки значения. За последней парой имя/значение должен следовать нулевой байт. Передаваться параметры могут в любом порядке. Обязательным является только параметр `user`, остальные могут отсутствовать. Каждый параметр задаётся так:

String

Имя параметра. В настоящее время принимаются имена:

`user`

Имя пользователя баз данных, с которым выполняется подключение. Является обязательным, значения по умолчанию нет.

`database`

База данных, к которой выполняется подключение. По умолчанию подставляется имя пользователя.

`options`

Аргументы командной строки для обслуживающего процесса. (Этот способ считается устаревшим; теперь следует устанавливать отдельные параметры времени выполнения.) Пробелы в этой строке воспринимаются как разделяющие аргументы, если перед ними нет обратной косой черты (`\`); чтобы представить обратную косую черту буквально, продублируйте её (`\\`).

`replication`

Используется для подключения в режиме потоковой репликации, в котором вместо операторов SQL может выполняться небольшой набор команд репликации. Допустимые значения: `true`, `false` (по умолчанию) и `database`. За подробностями обратитесь к Разделу 53.4.

В дополнение к ним могут задаваться и другие параметры. Имена параметров, начинающиеся с `_pq_.`, резервируются для использования в расширениях протокола, а остальные воспринимаются как параметры времени выполнения, передаваемые во время запуска серверному процессу. Такие параметры будут применяться при запуске серверного процесса (после разбора аргументов командной строки, если они есть) и будут действовать как параметры сеанса по умолчанию.

String

Значение параметра.

Sync (F)

Byte1('S')

Указывает, что это сообщение представляет команду Sync.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

Terminate (F)

Byte1('X')

Указывает, что это сообщение завершает сеанс.

Int32(4)

Длина содержимого сообщения в байтах, включая само поле длины.

53.8. Поля сообщений с ошибками и замечаниями

В этом разделе описываются поля, которые могут содержаться в сообщениях ErrorResponse и NoticeResponse. Для каждого типа поля определён свой идентификационный маркер. Заметьте, что в сообщении может содержаться поле любого из этих типов, но не больше одного раза.

`S`

Важность: поле содержит `ERROR`, `FATAL` или `PANIC` (в сообщении об ошибке), либо `WARNING`, `NOTICE`, `DEBUG`, `INFO` или `LOG` (в сообщении с замечанием), либо переведённые значения (ОШИБКА, ВАЖНО, ПАНИКА, ПРЕДУПРЕЖДЕНИЕ, ЗАМЕЧАНИЕ, ОТЛАДКА, ИНФОРМАЦИЯ, СООБЩЕНИЕ, соответственно). Это поле присутствует всегда.

`V`

Важность: поле содержит `ERROR`, `FATAL` или `PANIC` (в сообщении об ошибке) либо `WARNING`, `NOTICE`, `DEBUG`, `INFO` или `LOG` (в сообщении с замечанием). Это поле подобно `S`, но его содержимое никогда не переводится. Присутствует только в сообщениях, выдаваемых Postgres Pro версии 9.6 и новее.

`C`

Код: код SQLSTATE выданной ошибки (см. Приложение A). Не переводится на другие языки, присутствует всегда.

`M`

Сообщение: основное сообщение об ошибке, предназначенное для человека. Должно быть точным, но кратким (обычно в одну строку). Присутствует всегда.

`D`

Необязательное дополнительное сообщение об ошибке, передающее более детальную информацию о проблеме. Может занимать несколько строк.

`H`

Подсказка: необязательное предложение решения проблемы. Оно должно отличаться от подробного описания тем, что предлагает совет (не обязательно подходящий во всех случаях), а не сухие факты. Может располагаться в нескольких строках.

`P`

Позиция: значение поля представляет целочисленное число в ASCII, указывающее на положение ошибки в исходной строке запроса. Первый символ находится в позиции 1, при этом позиции отсчитываются по символам, а не по байтам.

`p`

Внутренняя позиция: она определяется так же, как поле `P`, но отражает положение ошибки во внутренне сгенерированной команде, а не в строке, переданной клиентом. Вместе с этим полем всегда присутствует поле `q`.

`q`

Внутренний запрос: текст внутренне сгенерированной команды, в которой произошла ошибка. Это может быть, например, SQL-запрос, выполняемый функцией на PL/pgSQL.

`W`

Где: указывает на контекст, в котором произошла ошибка. В настоящее время включает трассировку стека вызовов текущей функции на процедурном языке и внутренне сгенерированных запросов. Записи трассировки разделяются по строкам, вначале последняя.

`s`

Имя схемы: если ошибка связана с некоторым объектом базы данных, это поле содержит имя схемы, к которой относится объект (если такая есть).

`t`

Имя таблицы: если ошибка связана с некоторой таблицей, это поле содержит имя таблицы. (Узнать имя схемы таблицы можно из соответствующего отдельного поля.)

`c`

Имя столбца: если ошибка связана с некоторым столбцом таблицы, это поле содержит имя столбца. (Идентифицировать таблицу можно, обратившись к полям, содержащим имя таблицы и схемы.)

`d`

Имя типа данных: если ошибка связана с некоторым типом данных, это поле содержит имя типа. (Узнать имя схемы типа можно из соответствующего поля.)

`n`

Имя ограничения: если ошибка связана с некоторым ограничением, это поле содержит имя ограничения. Чтобы узнать, к какой таблице или домену она относится, обратитесь к полям, описанным выше. (В данном контексте индексы считаются ограничениями, даже если они были созданы не с синтаксисом ограничений.)

`F`

Файл: имя файла с исходным кодом, в котором была обнаружена ошибка.

`L`

Строка: номер строки в исходном коде, в которой была обнаружена ошибка.

`R`

Программа: имя программы в исходном коде, в которой была обнаружена ошибка.

Примечание

Поля, содержащие имена схемы, таблицы, столбца, типа данных и ограничения, выдаются только для ограниченного числа типов ошибок; см. Приложение A. Клиенты не должны рассчитывать на то, что присутствие одного из полей обязательно влечёт присутствие другого поля. Системные источники ошибок устанавливают связь между ними, но пользовательские функции могут использовать эти поля по-другому. Подобным образом, клиенты не должны полагаться на то, что эти поля ссылаются на актуальные объекты в текущей базе данных.

Клиент отвечает за форматирование отображаемой информации в соответствии с его нуждами; в частности, он должен разбивать длинные строки, как требуется. Символы новой строки, встречающиеся в полях сообщения об ошибке, должны обрабатываться, как разрывы абзацев, а не строк.

53.9. Форматы сообщений логической репликации

В этом разделе подробно описывается формат каждого сообщения логической репликации. Эти сообщения или выдаются через SQL-интерфейс слота репликации или передаются процессом walsender. Когда их передаёт walsender, они помещаются внутрь WAL-сообщений протокола репликации, описанных в Разделе 53.4, и в общем следуют тому же потоку сообщений, что и сообщения физической репликации.

Begin

Byte1('B')

Указывает, что это начальное сообщение.

Int64 (XLogRecPtr)

Окончательный LSN транзакции.

Int64 (TimestampTz)

Время фиксации транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи Postgres Pro (2000-01-01).

Int32 (TransactionId)

Идентификатор транзакции.

Сообщение

Byte1('M')

Указывает, что это сообщение логического декодирования.

Int32 (TransactionId)

Идентификатор транзакции (только для транзакций, передаваемых потоками). Это поле присутствует со 2-й версии протокола.

Int8

Флаги; либо 0, если флагов нет, либо 1, если сообщение логического декодирования передано как транзакционное.

Int64 (XLogRecPtr)

LSN сообщения логического декодирования.

String

Префикс сообщения логического декодирования.

Int32

Длина содержимого сообщения.

Byte*n*

Содержание сообщения логического декодирования.

Commit

Byte1('C')

Указывает, что это сообщение о фиксации.

Int8(0)

Флаги; в настоящее время не используются.

Int64 (XLogRecPtr)

LSN записи фиксации.

Int64 (XLogRecPtr)

Конечный LSN транзакции.

Int64 (TimestampTz)

Время фиксации транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи PostgreSQL (2000-01-01).

Origin

Byte1('O')

Указывает, что это сообщение об источнике.

Int64 (XLogRecPtr)

LSN записи фиксации на сервере-источнике.

String

Имя источника.

Заметьте, что внутри одной транзакции может быть несколько сообщений Origin.

Relation

Byte1('R')

Указывает, что это сообщение об отношении.

Int32 (TransactionId)

Идентификатор транзакции (только для транзакций, передаваемых потоками). Это поле присутствует со 2-й версии протокола.

Int32 (Oid)

OID отношения.

String

Пространство имён (пустая строка для `pg_catalog`).

String

Имя отношения.

Int8

Свойство идентификации реплики для отношения (то же, что и `relreplident` в `pg_class`).

Int16

Число столбцов.

Затем для каждого столбца в публикации (за исключением генерируемых) идёт следующий блок сообщения:

Int8

Флаги столбца. В настоящее время это может быть 0 (флагов нет) или 1 (столбец помечается как часть ключа).

String

Имя столбца.

Int32 (Oid)

OID типа данных столбца.

Int32

Модификатор типа столбца (`atttypmod`).

Тип

Byte1('Y')

Указывает, что это сообщение о типе.

Int32 (TransactionId)

Идентификатор транзакции (только для транзакций, передаваемых потоками). Это поле присутствует со 2-й версии протокола.

Int32 (Oid)

OID типа данных.

String

Пространство имён (пустая строка для `pg_catalog`).

String

Имя типа данных.

Insert

Byte1('I')

Указывает, что это сообщение о добавлении данных.

Int32 (TransactionId)

Идентификатор транзакции (только для транзакций, передаваемых потоками). Это поле присутствует со 2-й версии протокола.

Int32 (Oid)

OID отношения, соответствующий идентификатору в сообщении об отношении.

Byte1('N')

Обозначает следующее сообщение TupleData как содержащее новый кортеж.

TupleData

Блок сообщения TupleData, представляющий содержимое нового кортежа.

Update

Byte1('U')

Указывает, что это сообщение об изменении данных.

Int32 (TransactionId)

Идентификатор транзакции (только для транзакций, передаваемых потоками). Это поле присутствует со 2-й версии протокола.

Int32 (Oid)

OID отношения, соответствующий идентификатору в сообщении об отношении.

Byte1('K')

Указывает, что следующий блок TupleData содержит ключ. Это поле является необязательным и присутствует, только если изменение затронуло столбцы, являющиеся частью индекса REPLICA IDENTITY.

Byte1('O')

Указывает, что следующий блок TupleData содержит старый кортеж. Это поле является необязательным и присутствует, только если у таблицы, в которой произошло изменение, свойство REPLICA IDENTITY равно FULL.

TupleData

Блок сообщения TupleData, представляющий содержимое старого кортежа или первичного ключа. Присутствует, только если перед ним идёт признак 'O' или 'K'.

Byte1('N')

Обозначает следующее сообщение TupleData как содержащее новый кортеж.

TupleData

Блок сообщения TupleData, представляющий содержимое нового кортежа.

Сообщение Update может содержать либо блок 'K', либо блок 'O', либо ни один из них, но не оба сразу.

Delete

Byte1('D')

Указывает, что это сообщение об удалении данных.

Int32 (TransactionId)

Идентификатор транзакции (только для транзакций, передаваемых потоками). Это поле присутствует со 2-й версии протокола.

Int32 (Oid)

OID отношения, соответствующий идентификатору в сообщении об отношении.

Byte1('K')

Указывает, что следующий блок TupleData содержит ключ. Это поле присутствует, если таблица, в которой произошло удаление, использует индекс в качестве REPLICA IDENTITY.

Byte1('O')

Указывает, что следующий блок TupleData содержит старый кортеж. Это поле присутствует, если у таблицы, в которой произошло удаление, свойство REPLICA IDENTITY равно FULL.

TupleData

Блок сообщения TupleData, представляющий содержимое старого кортежа или первичного ключа, в зависимости от предыдущего поля.

Сообщение Delete может содержать либо блок 'K', либо блок 'O', но не оба сразу.

Truncate

Byte1('T')

Указывает, что это сообщение об усечении отношений.

Int32 (TransactionId)

Идентификатор транзакции (только для транзакций, передаваемых потоками). Это поле присутствует со 2-й версии протокола.

Int32

Число отношений.

Int8

Битовые флаги для `TRUNCATE`: 1 соответствует указанию `CASCADE`, 2 — `RESTART IDENTITY`.

Int32 (Oid)

OID отношения, соответствующий идентификатору в сообщении об отношении. Это поле повторятся для каждого отношения.

Описанные ниже сообщения Stream Start, Stream Stop, Stream Commit и Stream Abort доступны со 2-й версии протокола.

Stream Start

Byte1('S')

Указывает, что это сообщение о начале потока.

Int32 (TransactionId)

Идентификатор транзакции.

Int8

1 означает, что это первый сегмент потока с этим идентификатором; 0, если это любой другой сегмент.

Stream Stop

Byte1('E')

Указывает, что это сообщение об окончании потока.

Stream Commit

Byte1('c')

Указывает, что это сообщение о фиксации транзакции в потоке.

Int32 (TransactionId)

Идентификатор транзакции.

Int8(0)

Флаги; в настоящее время не используются.

Int64 (XLogRecPtr)

LSN записи фиксации.

Int64 (XLogRecPtr)

Конечный LSN транзакции.

Int64 (TimestampTz)

Время фиксации транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи Postgres Pro (2000-01-01).

Stream Abort

Byte1('A')

Указывает, что это сообщение о прерывании транзакции в потоке.

Int32 (TransactionId)

Идентификатор транзакции.

Int32 (TransactionId)

Идентификатор подтранзакции (для транзакций верхнего уровня совпадает с идентификатором транзакции).

Int64 (XLogRecPtr)

LSN операции прерывания, существует только если задано значение `parallel`. Это поле присутствует с 4-й версии протокола.

Int64 (TimestampTz)

Время прерывания транзакции, существует только если задано значение `parallel`. Значение задаётся в микросекундах, прошедших с начала эпохи PostgreSQL (2000-01-01). Это поле присутствует с 4-й версии протокола.

Описанные ниже сообщения (Begin Prepare, Prepare, Commit Prepared, Rollback Prepared, Stream Prepare) доступны с 3-й версии протокола.

Begin Prepare

Byte1('b')

Указывает, что это сообщение о начале подготовленной транзакции.

Int64 (XLogRecPtr)

LSN подготовки транзакции.

Int64 (XLogRecPtr)

Конечный LSN подготовленной транзакции.

Int64 (TimestampTz)

Время подготовки транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи PostgreSQL (2000-01-01).

Int32 (TransactionId)

Идентификатор транзакции.

String

Пользовательский GID подготовленной транзакции.

Prepare

Byte1('P')

Указывает, что это сообщение о подготовленной транзакции.

Int8(0)

Флаги; в настоящее время не используются.

Int64 (XLogRecPtr)

LSN подготовки транзакции.

Int64 (XLogRecPtr)

Конечный LSN подготовленной транзакции.

Int64 (TimestampTz)

Время подготовки транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи PostgreSQL (2000-01-01).

Int32 (TransactionId)

Идентификатор транзакции.

String

Пользовательский GID подготовленной транзакции.

Commit Prepared

Byte1('K')

Указывает, что это сообщение о фиксации подготовленной транзакции.

Int8(0)

Флаги; в настоящее время не используются.

Int64 (XLogRecPtr)

LSN фиксации подготовленной транзакции.

Int64 (XLogRecPtr)

Конечный LSN фиксации подготовленной транзакции.

Int64 (TimestampTz)

Время фиксации транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи PostgreSQL (2000-01-01).

Int32 (TransactionId)

Идентификатор транзакции.

String

Пользовательский GID подготовленной транзакции.

Rollback Prepared

Byte1('r')

Указывает, что это сообщение об откате подготовленной транзакции.

Int8(0)

Флаги; в настоящее время не используются.

Int64 (XLogRecPtr)

Конечный LSN подготовленной транзакции.

Int64 (XLogRecPtr)

Конечный LSN отката подготовленной транзакции.

Int64 (TimestampTz)

Время подготовки транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи PostgreSQL (2000-01-01).

Int64 (TimestampTz)

Время отката транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи PostgreSQL (2000-01-01).

Int32 (TransactionId)

Идентификатор транзакции.

String

Пользовательский GID подготовленной транзакции.

Stream Prepare

Byte1('p')

Указывает, что это сообщение о подготовленной транзакции, передаваемой в потоке.

Int8(0)

Флаги; в настоящее время не используются.

Int64 (XLogRecPtr)

LSN подготовки транзакции.

Int64 (XLogRecPtr)

Конечный LSN подготовленной транзакции.

Int64 (TimestampTz)

Время подготовки транзакции. Значение задаётся в микросекундах, прошедших с начала эпохи PostgreSQL (2000-01-01).

Int32 (TransactionId)

Идентификатор транзакции.

String

Пользовательский GID подготовленной транзакции.

Описанные выше сообщения имеют следующие общие блоки.

TupleData

Int16

Число столбцов.

Затем для каждого столбца (за исключением генерируемых) идёт одно из следующих вложенных сообщений:

Byte1('n')

Обозначает данные как значение NULL.

Или

Byte1('u')

Обозначает неизменённое значение TOAST (само значение не передаётся).

Или

Byte1('t')

Обозначает данные как значение в текстовом формате.

Или

Byte1('b')

Обозначает данные как значение в двоичном формате.

Int32

Длина значения столбца.

Byte*n*

Значение столбца в двоичном или текстовом формате, определённом предшествующим байтом формата. Переменная *n* задаёт длину значения.

53.10. Сводка изменений по сравнению с протоколом версии 2.0

В этом разделе представлен краткий список изменений к сведению разработчиков, желающих модернизировать существующие клиентские библиотеки до протокола 3.0.

В начальном стартовом пакете вместо фиксированного формата применяется гибкий формат списка строк. Заметьте, что теперь сеансовые значения по умолчанию для параметров времени выполнения можно задать непосредственно в стартовом пакете. (Вообще, это можно было делать и раньше, используя поле `options`, но из-за ограниченного размера `options` и невозможности задавать значения с пробелами, это вариант был не очень безопасным.)

Во всех сообщениях непосредственно за байтом типа сообщения следует счётчик длины (за исключением стартовых пакетов, в которых нет байта типа). Также заметьте, что байт типа теперь есть в сообщении PasswordMessage.

Сообщения ErrorResponse и NoticeResponse ('`E`' и '`N`') могут содержать несколько полей, из которых клиентский код может собрать сообщение об ошибке желаемого уровня детализации. Заметьте, что текст отдельных полей обычно не завершается новой строкой, тогда как в старом протоколе одиночная строка всегда завершалась так.

Сообщение ReadyForQuery ('`Z`') включает индикатор статуса транзакции.

Различие между типами данных BinaryRow и DataRow ушло; один тип сообщений DataRow позволяет возвращать данные во всех форматах. Заметьте, что формат DataRow был изменён для упрощения его разбора. Также изменилось представление двоичных значений: оно больше не привязано к внутреннему представлению сервера.

В протоколе появился новый раздел «расширенный протокол запросов», в котором добавлены типы сообщений для команд Parse, Bind, Execute, Describe, Close, Flush и Sync, а также типы серверных сообщений ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData и CloseComplete. Существующие клиенты могут не подстраиваться под этот раздел протокола, но если они задействует его, это позволит улучшить производительность или функциональность.

Данные `COPY` теперь внедряются в сообщения CopyData и CopyDone. Есть чётко определённый способ восстановить работу в случае ошибок в процессе `COPY`. Специальная последняя строка «`\.`» больше не нужна, она не передаётся при выполнении `COPY OUT`. (Она по-прежнему воспринимается как завершающая последовательность в потоке `COPY IN`, но это считается устаревшим способом завершения, и в конце концов он будет исключён.) Поддерживается `COPY` в двоичном режиме. Сообщения CopyInResponse и CopyOutResponse включают поля, определяющие число столбцов и формат каждого столбца.

Изменилась структура сообщений FunctionCall и FunctionCallResponse. Сообщение FunctionCall теперь позволяет передавать функциям аргументы NULL. Ещё в нём могут передаваться параметры и получаться результаты в текстовом или двоичном формате. Не осталось повода считать сообщение FunctionCall потенциально небезопасным, так как оно не даёт прямого доступа к внутренней презентации данных на сервере.

Сервер отправляет сообщения ParameterStatus ('`S`') при попытке подключения для всех параметров, которые он считает интересными для клиентской библиотеки. Как следствие, при любом изменении активного значения одного из этих параметров также выдаётся сообщение ParameterStatus.

Сообщение RowDescription ('`T`') содержит поля с OID таблицы и номером столбца для каждого столбца описываемой строки. В нём также передаётся код формата для каждого столбца.

Сервер более не выдаёт сообщение CursorResponse ('`P`').

В сообщении NotificationResponse ('`A`') добавилось ещё одно строковое поле, в котором может передаваться строка «сообщения» от отправителя события `NOTIFY`.

Раньше сообщение EmptyQueryResponse ('`I`') включало пустой строковый параметр; теперь он ликвидирован.

Глава 54. Написание обработчика процедурного языка

Все функции, написанные на языке, вызываемом не через текущий интерфейс «версии 1» для компилируемых языков (а именно, это функции на процедурных языках и функции, написанные на SQL) выполняются через *обработчик вызова* для заданного языка. Задача такого обработчика вызова — выполнить функцию должным образом, например, интерпретируя для этого её исходный текст. В этой главе в общих чертах рассказывается, как можно написать обработчик нового процедурного языка.

Обработчик вызова процедурного языка — это «обычная» функция, которая разрабатывается на компилируемом языке, таком как C, вызывается через интерфейс версии 1, и регистрируется в Postgres Pro как не принимающая аргументы и возвращающая тип `language_handler`. Этот специальный псевдотип помечает функцию как обработчик вызова и препятствует её вызову непосредственно из команд SQL. Более подробно соглашение о вызовах и динамическая загрузка кода на C описывается в Разделе 36.10.

Обработчик вызова вызывается так же, как и любая другая функция: он получает указатель на переменную `struct FunctionCallInfoBaseData`, содержащую значения аргументов и информацию о вызываемой функции, и должен вернуть результат типа `Datum` (и, возможно, установить признак `isnull` в структуре `FunctionCallInfoBaseData`, если нужно вернуть результат SQL NULL). Отличие обработчика вызова от обычной вызываемой функции состоит в том, что поле `flinfo->fn_oid` структуры `FunctionCallInfoBaseData` для него будет содержать OID вызываемой функции, а не самого обработчика. По этому OID обработчик вызова должен понять, какую функцию вызывать. Кроме того, список передаваемых аргументов для него формируется в соответствии с объявлением целевой функции, а не обработчика вызова.

Обработчик вызова сам должен выбрать запись функции из системного каталога `pg_proc` и проанализировать типы аргументов и результата вызываемой функции. Содержимое предложения `AS` команды `CREATE FUNCTION` для этой функции будет находиться в столбце `prosrc` строки в `pg_proc`. Обычно это исходный текст на процедурном языке, но в принципе это может быть и что-то другое, например, путь к файлу или иные данные, говорящие обработчику вызова, что именно делать.

Часто функция многократно вызывается в одном SQL-операторе. Чтобы в таких случаях избежать повторных обращений за информацией о вызываемой функции, обработчик вызова может воспользоваться полем `flinfo->fn_extra`. Изначально оно содержит `NULL`, но обработчик вызова может поместить в него указатель на требуемую информацию. При последующих вызовах, если поле `flinfo->fn_extra` будет отлично от `NULL`, им можно воспользоваться и пропустить шаг получения этой информации. Обработчик вызова должен позаботиться о том, чтобы указатель в `flinfo->fn_extra` указывал на блок памяти, который не будет освобождён раньше, чем завершится запрос (именно столько может существовать структура `FmgrInfo`). В качестве одного из вариантов, этого можно добиться, разместив дополнительные данные в контексте памяти, заданном в `flinfo->fn_mcxt`; срок жизни таких данных обычно совпадает со сроком жизни самой структуры `FmgrInfo`. С другой стороны, обработчик может выбрать и более долгоживущий контекст памяти с тем, чтобы кешировать определения функций и между запросами.

Когда функция на процедурном языке вызывается как триггер, ей не передаются аргументы обычным способом; вместо этого поле `context` в `FunctionCallInfoBaseData` указывает на структуру `TriggerData`, тогда как при обычном вызове функции оно содержит `NULL`. Обработчик языка, в свою очередь, должен каким-либо образом предоставить эту информацию функциям на этом процедурном языке.

Хотя обработчика вызова достаточно для создания простейшего процедурного языка, есть ещё две функции, которые можно реализовать дополнительно, чтобы пользоваться языком было удобнее: функция *проверки* и *обработчик внедрённого кода*. Функцию проверки можно реализовать, чтобы производить проверку синтаксиса языка во время CREATE FUNCTION. Если же реализован обработчик внедрённого кода, этот язык будет поддерживать выполнение анонимных блоков кода командой DO.

Если для процедурного языка предоставляется функция проверки, она должна быть объявлена как функция, принимающая один параметр типа `oid`. Результат функции проверки игнорируется, так что она обычно объявляется как возвращающая тип `void`. Эта функция будет вызываться в конце выполнения команды `CREATE FUNCTION`, создающей или изменяющей функцию, написанную на процедурном языке. Переданный ей OID указывает на строку в `pg_proc` для этой функции. Функция проверки должна выбрать эту строку обычным образом и произвести все необходимые проверки. Прежде всего нужно вызвать `CheckFunctionValidatorAccess()`, чтобы отличить явные вызовы этой функции от происходящих при выполнении команды `CREATE FUNCTION`. Затем обычно проверяется, например, что типы аргументов и результата функции поддерживаются языком и что тело функции синтаксически правильно для данного языка. Если функция проверки заключает, что всё в порядке, она должна просто завершиться. Если же она обнаруживает ошибку, она должна сообщить о ней через обычный механизм `ereport()`. Выданная таким образом ошибка приведёт к откату транзакции, так что определение некорректной функции зафиксировано не будет.

Функции проверки обычно должны учитывать параметр check_function_bodies: если он отключён, то дорогостоящие или зависящие от контекста проверки содержимого функции выполнять не следует. Если язык подразумевает выполнение кода в процессе компиляции, проверяющая функция должна избегать проверок, которые влекут за собой такое выполнение. В частности, указанный параметр отключает утилита pg_dump, чтобы она могла загружать функции на процедурных языках, не заботясь о побочных эффектах или зависимостях содержимого функций от других объектов базы. (Вследствие этого требования, обработчик языка не должен полагать, что функция прошла полную проверку. Смысл существования функции проверки не в том, чтобы убрать эти проверки из обработчика вызова, а в том, чтобы немедленно уведомить пользователя об очевидных ошибках при выполнении `CREATE FUNCTION`.) Хотя выбор, что именно должно проверяться, по большому счёту остаётся за функцией проверки, заметьте, что основной код `CREATE FUNCTION` выполняет присваивания `SET`, связанные с функцией, только когда `check_function_bodies` включён. Таким образом, проверки, результаты которых могут зависеть от параметров GUC, определённо должны опускаться, когда `check_function_bodies` отключён, во избежание ложных ошибок при восстановлении базы из копии.

Если для процедурного языка предоставляется обработчик встроенного кода, он должен объявляться в виде функции, принимающей один параметр типа `internal`. Результат такого обработчика игнорируется, поэтому обычно он объявляется как возвращающий тип `void`. Обработчик встроенного кода будет вызываться при выполнении оператора `DO` с данным процедурным языком. В качестве параметра ему на самом деле передаётся указатель на структуру `InlineCodeBlock`, содержащую информацию о параметрах `DO`, в частности, текст выполняемого анонимного блока внедрённого кода.

Все подобные объявления функций, а также саму команду `CREATE LANGUAGE`, рекомендуется упаковывать в *расширение* так, чтобы для установки языка было достаточно простой команды `CREATE EXTENSION`. За информацией о разработке расширений обратитесь к Разделу 36.17.

Реализация процедурных языков, включённых в стандартный дистрибутив, может послужить хорошим примером при написании собственных обработчиков языков. Некоторые полезные детали можно узнать на странице справки CREATE LANGUAGE.

Глава 55. Написание обёртки сторонних данных

Все операции со сторонней таблицей производятся через созданную для неё обёртку сторонних данных, представляющую собой набор подпрограмм, которые вызывает ядро сервера. Обёртка сторонних данных отвечает за получение данных из удалённого источника данных и передачу их исполнителю запросов Postgres Pro. Чтобы поддерживалось изменение данных в сторонних таблицах, эту операцию также должна выполнять обёртка. В данной главе освещается написание обёртки сторонних данных.

Реализация обёрток сторонних данных, включённых в стандартный дистрибутив, может послужить хорошим примером при написании собственных обёрток. Её вы можете найти в подкаталоге `contrib` дерева исходного кода. Некоторые полезные детали также можно узнать на странице справки CREATE FOREIGN DATA WRAPPER.

> **Примечание**
>
> В стандарте SQL описан интерфейс для написания обёрток сторонних данных, но Postgres Pro не реализует его, так как это потребовало бы больших усилий, а данный стандартизированный API всё равно не получил широкого распространения.

55.1. Функции обёрток сторонних данных

Автор FDW (Foreign Data Wrapper, Обёртки сторонних данных) должен реализовать функцию-обработчик и может дополнительно добавить функцию проверки. Обе функции должны быть написаны на компилируемом языке, таком как C, и использовать интерфейс версии 1. Подробнее соглашение о вызовах и динамическая загрузка кода на C описывается в Разделе 36.10.

Функция-обработчик просто возвращает структуру с указателями на реализующие подпрограммы, которые будут вызываться планировщиком, исполнителем и различными служебными командами. Основная часть разработки FDW заключается в написании этих реализующих подпрограмм. Функция-обработчик должна быть зарегистрирована в Postgres Pro как функция без аргументов, возвращающая специальный псевдотип `fdw_handler`. Реализующие подпрограммы представляют собой обычные функции на C, которые не видны и не могут вызываться на уровне SQL. Они описаны в Разделе 55.2.

Функция проверки отвечает за проверку параметров, передаваемых с командами `CREATE` и `ALTER` для этой обёртки сторонних данных, а также параметров сторонних серверов, сопоставлений пользователей и сторонних таблиц, доступных через эту обёртку. Эта функция должна быть зарегистрирована как принимающая два аргумента: текстовый массив, содержащий параметры для проверки, и OID, представляющий тип объекта, с которым связаны эти параметры. Второй аргумент соответствует OID системного каталога, в котором будет сохраняться объект:

- `AttributeRelationId`
- `ForeignDataWrapperRelationId`
- `ForeignServerRelationId`
- `ForeignTableRelationId`
- `UserMappingRelationId`

. Если функция проверки отсутствует, параметры не проверяются ни при создании, ни при изменении объекта.

55.2. Подпрограммы обёртки сторонних данных

Функция-обработчик FDW возвращает структуру `FdwRoutine` (выделенную с помощью palloc), содержащую указатели на подпрограммы, которые реализуют описанные ниже функции. Из всех функций обязательными являются только те, что касаются сканирования, а остальные могут отсутствовать.

Тип структуры `FdwRoutine` объявлен в *src/include/foreign/fdwapi.h*, там же можно узнать дополнительные подробности.

55.2.1. Подпрограммы FDW для сканирования сторонних таблиц

```
void
GetForeignRelSize(PlannerInfo *root,
                  RelOptInfo *baserel,
                  Oid foreigntableid);
```

Выдаёт оценку размера отношения для сторонней таблицы. Она вызывается в начале планирования запроса, в котором сканируется сторонняя таблица. В параметре `root` передаётся общая информация планировщика о запросе, в `baserel` — информация о данной таблице, а в `foreigntableid` — OID записи в `pg_class` для данной таблицы. (Значение `foreigntableid` можно получить и из структуры данных планировщика, но простоты ради оно передаётся явно.)

Эта функция должна записать в `baserel->rows` ожидаемое число строк, которое будет получено при сканировании таблицы, с учётом фильтра, заданного ограничением выборки. Изначально в `baserel->rows` содержится просто постоянная оценка по умолчанию, которую следует заменить, если это вообще возможно. Функция также может поменять значение `baserel->width`, если она может дать лучшую оценку среднего размера строки результата. (Начальное значение зависит от типов столбцов и от среднего размера значений в них, рассчитанного при последнем `ANALYZE`.) Также эта функция может изменить значение `baserel->tuples`, если она может дать лучшую оценку общего количества строк в сторонней таблице. (Начальное значение берётся из поля `pg_class.reltuples`, которое содержит общее количество строк, полученное при последнем `ANALYZE`; или `-1`, если `ANALYZE` для этой сторонней таблицы ещё не выполнялся.)

За дополнительными сведениями обратитесь к Разделу 55.4.

```
void
GetForeignPaths(PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid);
```

Формирует возможные пути доступа для сканирования сторонней таблицы. Эта функция вызывается при планировании запроса. Ей передаются те же параметры, что и функции `GetForeignRelSize`, которая к этому времени уже будет вызвана.

Эта функция должна выдать минимум один путь доступа (узел `ForeignPath`) для сканирования сторонней таблицы и должна вызвать `add_path`, чтобы добавить каждый такой путь в `baserel->pathlist`. Для формирования узлов `ForeignPath` рекомендуется вызывать `create_foreignscan_path`. Данная функция может выдавать несколько путей доступа, то есть путей, для которых по заданным `pathkeys` можно получить уже отсортированный результат. Каждый путь доступа должен содержать оценки стоимости и может содержать любую частную информацию FDW, необходимую для выбора целевого метода сканирования.

За дополнительными сведениями обратитесь к Разделу 55.4.

```
ForeignScan *
GetForeignPlan(PlannerInfo *root,
               RelOptInfo *baserel,
               Oid foreigntableid,
               ForeignPath *best_path,
               List *tlist,
               List *scan_clauses,
               Plan *outer_plan);
```

Создаёт узел плана `ForeignScan` из выбранного пути доступа к сторонней таблице. Эта функция вызывается в конце планирования запроса. Ей передаются те же параметры, что и

`GetForeignRelSize`, плюс выбранный путь `ForeignPath` (до этого сформированный функциями `GetForeignPaths`, `GetForeignJoinPaths` или `GetForeignUpperPaths`), целевой список, который должен быть выдан этим узлом плана, условия ограничения, которые должны применяться для данного узла, и внешний вложенный подплан `ForeignScan`, применяемый для перепроверок, выполняемых функцией `RecheckForeignScan`. (Если путь задаётся для соединения, а не для базового отношения, в `foreigntableid` передаётся `InvalidOid`.)

Эта функция должна создать и выдать узел плана `ForeignScan`; для формирования этого узла рекомендуется использовать `make_foreignscan`.

За дополнительными сведениями обратитесь к Разделу 55.4.

```
void
BeginForeignScan(ForeignScanState *node,
                 int eflags);
```

Начинает сканирование сторонней таблицы. Эта функция вызывается при запуске исполнителя. Она должна выполнить все подготовительные действия, необходимые для осуществления сканирования, но не должна собственно производить сканирование (оно должно начаться с первым вызовом `IterateForeignScan`). Узел `ForeignScanState` уже был создан, но его поле `fdw_state` по-прежнему NULL. Информацию о сканируемой таблице можно получить через узел `ForeignScanState` (в частности, из нижележащего узла `ForeignScan`, содержащего частную информацию FDW, заданную функцией `GetForeignPlan`). Параметр `eflags` содержит битовые флаги, описывающие режим работы исполнителя для этого узла плана.

Заметьте, что когда `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` не равно нулю, эта функция не должна выполнять какие-либо внешне проявляющиеся действия; она должна сделать только то, что необходимо для получения состояния узла, подходящего для `ExplainForeignScan` и `EndForeignScan`.

```
TupleTableSlot *
IterateForeignScan(ForeignScanState *node);
```

Выбирает одну строку из стороннего источника и возвращает её в слоте таблицы кортежей (для этой цели следует использовать `ScanTupleSlot`, переданный с узлом). Когда строки заканчиваются, возвращает NULL. Инфраструктура слотов таблицы кортежей позволяет возвращать как физические, так и виртуальные кортежи; в большинстве случаев второй вариант предпочтительнее с точки зрения производительности. Заметьте, что эта функция вызывается в кратковременном контексте памяти, который будет сбрасываться между вызовами. Если вам нужна более долгоживущая память, создайте соответствующий контекст в `BeginForeignScan` либо используйте `es_query_cxt` из структуры `EState`, переданной с узлом.

Возвращаемые строки должны соответствовать целевому списку `fdw_scan_tlist`, если он передаётся, а в противном случае — типу строки сканируемой сторонней таблицы. Если вы решите для оптимизации не возвращать ненужные столбцы, в их позиции нужно вставить NULL, либо сформировать список `fdw_scan_tlist` без этих столбцов.

Заметьте, что для исполнителя Postgres Pro не важно, удовлетворяют ли возвращаемые строки каким-либо ограничениям, определённым для сторонней таблицы — но это важно для планировщика, так что запросы могут оптимизироваться некорректно, если в сторонней таблице будут видны строки, не удовлетворяющие объявленному ограничению. Если ограничение нарушается, тогда как пользователь объявил, что оно должно выполняться, может быть уместно сообщить об ошибке (точно так же, как и при несовпадении типов данных).

```
void
ReScanForeignScan(ForeignScanState *node);
```

Перезапускает сканирование с начала. Заметьте, что значения параметров, от которых зависит сканирование, могли измениться, так что новое сканирование не обязательно вернёт те же строки.

```
void
EndForeignScan(ForeignScanState *node);
```

Завершает сканирование и освобождает ресурсы. Обычно при этом не нужно освобождать память, выделенную через palloc, но например, открытые файлы и подключения к удалённым серверам следует закрыть.

55.2.2. Подпрограммы FDW для сканирования сторонних соединений

Если FDW поддерживает соединения на удалённой стороне (вместо того, чтобы считывать данные обеих таблиц и выполнять соединения локально), она должна предоставить эту реализующую подпрограмму:

```
void
GetForeignJoinPaths(PlannerInfo *root,
                    RelOptInfo *joinrel,
                    RelOptInfo *outerrel,
                    RelOptInfo *innerrel,
                    JoinType jointype,
                    JoinPathExtraData *extra);
```

Формирует возможные пути доступа для соединения двух (и более) сторонних таблиц, принадлежащих одному стороннему серверу. Эта необязательная функция вызывается во время планирования запроса. Как и `GetForeignPaths`, эта функция должна построить пути `ForeignPath` для переданного `joinrel` (для построения путей используйте `create_foreign_join_path`) и вызвать `add_path`, чтобы добавить эти пути в набор путей, подходящих для соединения. Но, в отличие от `GetForeignPaths`, эта функция не обязательно должна возвращать минимум один путь, так как всегда возможен альтернативный путь с локальным соединением таблиц.

Заметьте, что эта функция будет вызываться неоднократно для одного и того же соединения с разными комбинациями внутреннего и внешнего отношений; минимизировать двойную работу должна сама FDW.

Также обратите внимание, что набор условий, применяемых к соединению, который передаётся как `extra->restrictlist`, варьируется в зависимости от комбинации внутренних и внешних отношений. Путь `ForeignPath`, сгенерированный для `joinrel`, должен содержать набор используемых условий соединения, который будет использоваться планировщиком для преобразования пути `ForeignPath` в план, если он был выбран как лучший путь для `joinrel`.

Если для соединения выбирается путь `ForeignPath`, он будет представлять весь процесс соединения; пути, сформированные для задействованных таблиц и подчинённых соединений, в нём применяться не будут. Далее этот путь соединения обрабатывается во многом так же, как и путь сканирования одной сторонней таблицы. Одно различие состоит в том, что `scanrelid` результирующего плана узла `ForeignScan` должно быть равно нулю, так как он не представляет какое-либо одно отношение; вместо этого набор соединяемых отношений представляется в поле `fs_relids` узла `ForeignScan`. (Это поле заполняется автоматически кодом ядра планировщика, так что FDW делать это не нужно.) Ещё одно отличие в том, что список столбцов для удалённого соединения нельзя получить из системных каталогов и поэтому FDW должна выдать в `fdw_scan_tlist` требуемый список узлов `TargetEntry`, представляющий набор столбцов, которые будут выдаваться во время выполнения в возвращаемых кортежах.

Примечание

Начиная с Postgres Pro 16, в `fs_relids` учитываются наборы строк, полученные в результате внешних соединений, если они были задействованы в данном соединении. Новое поле `fs_base_relids` включает только базовые отношения, то есть копирует старую семантику `fs_relids`.

За дополнительными сведениями обратитесь к Разделу 55.4.

55.2.3. Подпрограммы FDW для планирования обработки после сканирования/соединения

Если FDW поддерживает удалённое выполнение операций после сканирования/соединения, например, удалённое агрегирование, она должна предоставить эту реализующую подпрограмму:

```
void
GetForeignUpperPaths(PlannerInfo *root,
                     UpperRelationKind stage,
                     RelOptInfo *input_rel,
                     RelOptInfo *output_rel,
                     void *extra);
```

Формирует возможные пути доступа для обработки *верхнего отношения*. Этот термин планировщика подразумевает любую обработку запросов после сканирования/соединения, в частности, агрегирование, вычисление оконных функций, сортировку и изменение таблиц. Эта необязательная функция вызывается во время планирования запроса. В настоящее время она вызывается, только если все базовые отношения, задействованные в запросе, относятся к одной FDW. Эта функция должна построить пути `ForeignPath` для любых действий после сканирования/соединения, которые FDW умеет выполнять удалённо (для построения путей используйте `create_foreign_upper_path`), и вызвать `add_path`, чтобы добавить эти пути к указанному верхнему отношению. Как и `GetForeignJoinPaths`, эта функция не обязательно должна возвращать какие-либо пути, так как всегда возможны пути с локальной обработкой.

Параметр `stage` определяет, какой шаг после сканирования/соединения рассматривается в данный момент. Параметр `output_rel` указывает на верхнее отношение, которое должно получить пути, представляющие вычисление этого шага, а `input_rel` — на отношение, представляющее входные данные для этого шага. В параметре `extra` передаётся дополнительная информация; в настоящее время он устанавливается только для `UPPERREL_PARTIAL_GROUP_AGG` и `UPPERREL_GROUP_AGG` (в этом случае указывает на структуру `GroupPathExtraData`) и для `UPPERREL_FINAL` (тогда он указывает на структуру `FinalPathExtraData`). Заметьте, что пути `ForeignPath`, добавляемые в `output_rel`, обычно не будут напрямую зависеть от путей `input_rel`, так как ожидается, что они будут обрабатываться снаружи. Однако изучить пути, построенные для предыдущего шага обработки, может быть полезно для исключения лишних операций при планировании.

За дополнительными сведениями обратитесь к Разделу 55.4.

55.2.4. Подпрограммы FDW для изменения данных в сторонних таблицах

Если FDW поддерживает запись в сторонние таблицы, она должна предоставить некоторые или все подпрограммы, реализующие следующие функции, в зависимости от потребностей и возможностей FDW:

```
void
AddForeignUpdateTargets(PlannerInfo *root,
                        Index rtindex,
                        RangeTblEntry *target_rte,
                        Relation target_relation);
```

Операции `UPDATE` и `DELETE` выполняются со строками, ранее выбранными функциями сканирования таблицы. FDW может потребоваться дополнительная информация, например, ID строки или значения столбцов первичного ключа, чтобы точно знать, какую именно строку нужно изменить или удалить. Для этого данная функция может добавить дополнительные скрытые или «отбросовые» целевые столбцы в список столбцов, которые должны быть получены из сторонней таблицы во время `UPDATE` или `DELETE`.

Для этого создайте структуру `Var`, представляющую нужное дополнительное значение, и передайте её функции `add_row_identity_var` вместе с именем отбросового столбца. (Это можно сделать несколько раз, если требуется добавить несколько столбцов.) Имя отбросового столбца в каждой структуре `Var` должно быть уникальным, за исключением случая с одноимёнными структурами Var, отличающимися только значением `varno` — в таких структурах имена могут и должны быть одинаковыми. Основная система использует следующие имена отбросовых столбцов: `tableoid` для столбца `tableoid` таблицы, `ctid` или `ctidN` для `ctid`, `wholerow` для переменной `Var`, содержащей целую строку и отмеченной `vartype` = `RECORD`, и `wholerowN` для переменной `Var`, содержащей целую строку, с `vartype`, равным объявленному типу строки таблицы. По возможности используйте эти имена повторно (планировщик объединит повторяющиеся запросы для идентичных отбросовых столбцов). Если вам нужен другой тип отбросового столбца, помимо уже используемых, будет разумно выбрать имя с префиксом вашего имени расширения, чтобы избежать конфликтов с другими FDW.

Если указатель `AddForeignUpdateTargets` равен `NULL`, дополнительные целевые выражения не добавляются. (Это делает невозможным реализацию операций `DELETE`, хотя операция `UPDATE` может быть всё же возможна, если FDW идентифицирует строки, полагаясь на то, что первичный ключ не меняется.)

```
List *
PlanForeignModify(PlannerInfo *root,
                  ModifyTable *plan,
                  Index resultRelation,
                  int subplan_index);
```

Выполняет любые дополнительные действия планирования, необходимые для добавления, изменения или удаления в сторонней таблице. Эта функция формирует частную информацию FDW, которая будет добавлена в узел плана `ModifyTable`, осуществляющий изменение. Эта информация должна возвращаться в списке (`List`); она будет доставлена в функцию `BeginForeignModify` на стадии выполнения.

В `root` передаётся общая информация планировщика о запросе, а в `plan` — узел плана `ModifyTable`, заполненный, не считая поля `fdwPrivLists`. Параметр `resultRelation` указывает на целевую стороннюю таблицу по номеру в списке отношений, а `subplan_index` определяет целевое отношение в данном узле `ModifyTable`, начиная с нуля; воспользуйтесь этим индексом, обращаясь к вложенным структурам узла `plan`, соответствующим целевым отношениям.

За дополнительными сведениями обратитесь к Разделу 55.4.

Если указатель `PlanForeignModify` равен `NULL`, дополнительные действия во время планирования не предпринимаются, и в качестве `fdw_private` в `BeginForeignModify` поступит NULL.

```
void
BeginForeignModify(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo,
                   List *fdw_private,
                   int subplan_index,
                   int eflags);
```

Начинает выполнение операции изменения данных в сторонней таблице. Эта подпрограмма выполняется при запуске исполнителя. Она должна выполнять любые подготовительные действия, необходимые для того, чтобы собственно произвести изменения в таблице. Впоследствии для кортежей, которые будут вставляться, изменяться или удаляться, будет вызываться `ExecForeignInsert/ExecForeignBatchInsert`, `ExecForeignUpdate` или `ExecForeignDelete`.

В параметре `mtstate` передаётся общее состояние выполняемого плана узла `ModifyTable`; через эту структуру доступны глобальные сведения о плане и состояние выполнения. В `rinfo` передаётся структура `ResultRelInfo`, описывающая целевую стороннюю таблицу. (Если FDW нужно сохранить частное состояние, необходимое для этой операции, она может воспользоваться полем `ri_FdwState` структуры `ResultRelInfo`.) В `fdw_private` передаются частные данные, если они были сформированы процедурой `PlanForeignModify`. Параметр `subplan_index` определяет целевое отношение в данном узле `ModifyTable`, а в `eflags` передаются битовые флаги, описывающие режим работы исполнителя для этого узла плана.

Заметьте, что когда `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` не равно нулю, эта функция не должна выполнять какие-либо внешне проявляющиеся действия; она должна сделать только то, что необходимо для получения состояния узла, подходящего для `ExplainForeignModify` и `EndForeignModify`.

Если указатель на `BeginForeignModify` равен `NULL`, никакое действие при запуске исполнителя не выполняется.

```
TupleTableSlot *
ExecForeignInsert(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

Вставляет один кортеж в стороннюю таблицу. В `estate` передаётся глобальное состояние выполнения запроса, а в `rinfo` — структура `ResultRelInfo`, описывающая целевую стороннюю таблицу. Параметр `slot` содержит кортеж, который должен быть вставлен; он будет соответствовать определению типа строки сторонней таблицы. Параметр `planSlot` содержит кортеж, сформированный вложенным планом узла `ModifyTable`; он отличается от `slot` тем, что может содержать дополнительные «отбросовые» столбцы. (Значение `planSlot` обычно не очень интересно для операций `INSERT`, но оно представлено для полноты.)

Возвращаемым значением будет либо слот, содержащий данные, которые были фактически вставлены (они могут отличаться от переданных данных, например, в результате действий триггеров), либо NULL, если никакая строка фактически не была вставлена (опять же, обычно в результате действий триггеров). Чтобы вернуть результат, также можно использовать передаваемый на вход `slot`.

Данные в возвращаемом слоте используются, только если оператор `INSERT` содержит предложение `RETURNING`, задействуется представление с указанием `WITH CHECK OPTION` либо если для сторонней таблицы определён триггер `AFTER ROW`. Триггерам нужны все столбцы, но FDW может для оптимизации не возвращать некоторые или все, в зависимости от содержания предложения `RETURNING` или ограничения `WITH CHECK OPTION`. Так или иначе, какой-либо слот необходимо вернуть, чтобы отметить, что операция успешна, иначе число возвращённых запросом строк будет неверным.

Если указатель на `ExecForeignInsert` равен `NULL`, вставить данные в стороннюю таблицу не удастся, в ответ будет выдаваться сообщение об ошибке.

Заметьте, что эта функция также вызывается при добавлении кортежей, перенаправленных в секцию в сторонней таблице, или при выполнении `COPY FROM` со сторонней таблицей. В данных случаях она вызывается не так, как при выполнении обычного `INSERT`. Ниже описаны функции обратного вызова, позволяющие реализовать поддержку этих операций в обёртке сторонних данных.

```
TupleTableSlot **
ExecForeignBatchInsert(EState *estate,
                       ResultRelInfo *rinfo,
                       TupleTableSlot **slots,
                       TupleTableSlot **planSlots,
                       int *numSlots);
```

Вставляет сразу несколько кортежей в стороннюю таблицу. Этой функции передаются по сути те же параметры, что `ExecForeignInsert`, но в `slots` и `planSlots` передаются массивы кортежей, а `*numSlots` задаёт количество кортежей в этих массивах.

Возвращаемым значением будет массив слотов, содержащих данные, которые были фактически вставлены (они могут отличаться от переданных данных, например, в результате действий триггеров). Чтобы вернуть результат, также можно использовать передаваемый на вход массив `slots`. Количество успешно вставленных кортежей выдаётся в `*numSlots`.

Данные в возвращаемом слоте используются, только если при выполнении оператора `INSERT` задействуется представление с указанием `WITH CHECK OPTION` или если для сторонней таблицы определён триггер `AFTER ROW`. Триггерам нужны все столбцы, но FDW может для оптимизации не возвращать некоторые или все, в зависимости от содержания ограничений `WITH CHECK OPTION`.

Если указатель на `ExecForeignBatchInsert` или `GetForeignModifyBatchSize` равен `NULL`, для добавления данных в стороннюю таблицу вместо данной функции будет использоваться функция `ExecForeignInsert`. Данная функция также не используется, если в `INSERT` присутствует предложение `RETURNING`.

Заметьте, что эта функция также вызывается при добавлении кортежей, перенаправленных в секцию в сторонней таблице, или при выполнении `COPY FROM` со сторонней таблицей. В данных случаях она вызывается не так, как при выполнении обычного `INSERT`. Ниже описаны функции обратного вызова, позволяющие реализовать поддержку этих операций в обёртке сторонних данных.

```
int
GetForeignModifyBatchSize(ResultRelInfo *rinfo);
```

Выдаёт максимальное количество кортежей, которое может быть обработано за один вызов функции `ExecForeignBatchInsert` для указанной сторонней таблицы. Исполнитель передаёт не более данного количества кортежей в `ExecForeignBatchInsert`. `rinfo` — это структура `ResultRelInfo`, описывающая целевую стороннюю таблицу. Ожидается, что FDW даст пользователю возможность задать этот параметр на уровне стороннего сервера и/или таблицы, либо будет использоваться некоторое фиксированное значение.

Если указатель на `ExecForeignInsert` или `GetForeignModifyBatchSize` равен `NULL`, для добавления данных в стороннюю таблицу будет использоваться функция `ExecForeignInsert`.

```
TupleTableSlot *
ExecForeignUpdate(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

Изменяет один кортеж в сторонней таблице. В `estate` передаётся глобальное состояние выполнения запроса, а в `rinfo` — структура `ResultRelInfo`, описывающая целевую стороннюю таблицу. Параметр `slot` содержит новые данные для кортежа; он будет соответствовать определению типа строки сторонней таблицы. Параметр `planSlot` содержит кортеж, сформированный вложенным планом узла `ModifyTable`. Он отличается от `slot` тем, что содержит только новые значения для столбцов, изменённых в результате запроса, поэтому не следует обращаться к элементам `planSlot` по номерам атрибутов сторонней таблицы. Кроме того, `planSlot` обычно содержит дополнительные «отбросовые» столбцы. В частности, в этом слоте можно получить любые отбросовые столбцы, запрошенные в `AddForeignUpdateTargets`.

Возвращаемым значением будет либо слот, содержащий строку в состоянии после изменения (её содержимое может отличаться от переданного, например, в результате действий триггеров), либо NULL, если никакая строка фактически не была изменена (опять же, обычно в результате действий триггеров). Чтобы вернуть результат, также можно использовать передаваемый на вход `slot`.

Данные в возвращаемом слоте используются, только если оператор `UPDATE` содержит предложение `RETURNING`, задействуется представление с указанием `WITH CHECK OPTION` либо если для сторонней таблицы определён триггер `AFTER ROW`. Триггерам нужны все столбцы, но FDW может для оптимизации не возвращать некоторые или все, в зависимости от содержания предложения `RETURNING` или ограничения `WITH CHECK OPTION`. Так или иначе, какой-либо слот необходимо вернуть, чтобы отметить, что операция успешна, иначе число возвращённых запросом строк будет неверным.

Если указатель на `ExecForeignUpdate` равен `NULL`, изменить данные в сторонней таблице не удастся, а в ответ будет выдаваться сообщение об ошибке.

```
TupleTableSlot *
```

```
ExecForeignDelete(EState *estate,
                  ResultRelInfo *rinfo,
                  TupleTableSlot *slot,
                  TupleTableSlot *planSlot);
```

Удаляет один кортеж из сторонней таблицы. В `estate` передаётся глобальное состояние выполнения запроса, а в `rinfo` — структура `ResultRelInfo`, описывающая целевую стороннюю таблицу. Параметр `slot` при вызове не содержит ничего полезного, но в эту структуру можно поместить возвращаемый кортеж. Параметр `planSlot` содержит кортеж, сформированный вложенным планом узла `ModifyTable`; в частности, в нём могут содержаться отбросовые столбцы, запрошенные в `AddForeignUpdateTargets`. Отбросовые столбцы необходимы, чтобы определить, какой именно кортеж удалять.

Возвращаемым значением будет либо слот, содержащий строку, которая была удалена, либо NULL, если не удалена никакая строка (обычно в результате действия триггеров). Для размещения возвращаемого кортежа можно использовать передаваемый на вход `slot`.

Данные в возвращаемом слоте используются, только если запрос `DELETE` содержит предложение `RETURNING` или для сторонней таблицы определён триггер `AFTER ROW`. Триггерам нужны все столбцы, но для предложения `RETURNING` FDW может ради оптимизации не возвращать некоторые или все столбцы, в зависимости от его содержания. Так или иначе, какой-либо слот необходимо вернуть, чтобы отметить, что операция успешна, иначе возвращённое число строк будет неверным.

Если указатель на `ExecForeignDelete` равен `NULL`, удалить данные из сторонней таблицы не удастся, а в ответ будет выдаваться сообщение об ошибке.

```
void
EndForeignModify(EState *estate,
                 ResultRelInfo *rinfo);
```

Завершает изменение данных в таблице и освобождает ресурсы. Обычно при этом не нужно освобождать память, выделенную через palloc, но например, открытые файлы и подключения к удалённым серверам следует закрыть.

Если указатель на `EndForeignModify` равен `NULL`, никакое действие при завершении исполнителя не выполняется.

Кортежи, вставляемые в секционированную таблицу командами `INSERT` и `COPY FROM`, направляются в соответствующие секции. Если обёртка сторонних данных поддерживает перенаправление в секции в сторонних таблицах, она также должна предоставить описанные ниже обработчики. Эти функции также вызываются, когда результат `COPY FROM` помещается в стороннюю таблицу.

```
void
BeginForeignInsert(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo);
```

Начинает выполнение операции добавления данных в сторонней таблице. Эта подпрограмма вызывается непосредственно перед тем, как первый кортеж будет вставлен в стороннюю таблицу — и когда это секция, выбранная для размещения кортежа, и когда это целевое отношение команды `COPY FROM`. Она должна выполнять любые подготовительные действия, необходимые перед собственно добавлением данных. Впоследствии для кортежей, добавляемых в стороннюю таблицу, будет вызываться обработчик `ExecForeignInsert` или `ExecForeignBatchInsert`.

В параметре `mtstate` передаётся общее состояние выполняемого плана узла `ModifyTable`; через эту структуру доступна глобальная информация о плане и состоянии выполнения. В `rinfo` передаётся структура `ResultRelInfo`, описывающая целевую стороннюю таблицу. (Если FDW нужно сохранить частное состояние, необходимое для этой операции, она может воспользоваться полем `ri_FdwState` структуры `ResultRelInfo`.)

Когда этот обработчик вызывается командой `COPY FROM`, связанные с планом глобальные данные в `mtstate` не передаются. При этом параметр `planSlot` обработчика `ExecForeignInsert`, впоследствии вызываемого для каждого вставляемого кортежа, равен `NULL` — и когда сторонняя таблица является секцией, выбранной для помещения кортежа, и когда это целевое отношение данной команды.

Если указатель на `BeginForeignInsert` равен `NULL`, никакое действие при инициализации не выполняется.

Заметьте, что если обёртка сторонних данных не поддерживает перенаправление кортежей в секции и/или операцию `COPY FROM` со сторонними таблицами, эта функция или вызываемая за ней `ExecForeignInsert/ExecForeignBatchInsert` должны выдать соответствующую ошибку.

```
void
EndForeignInsert(EState *estate,
                 ResultRelInfo *rinfo);
```

Завершает операцию добавления и освобождает ресурсы. Обычно при этом не нужно освобождать память, выделенную через palloc, но например, открытые файлы и подключения к удалённым серверам следует закрыть.

Если указатель на `EndForeignInsert` равен `NULL`, никакое действие при завершении не выполняется.

```
int
IsForeignRelUpdatable(Relation rel);
```

Сообщает, какие операции изменения данных поддерживает указанная сторонняя таблица. Возвращаемое значение должно быть битовой маской кодов событий, обозначающих операции, поддерживаемые таблицей, и заданных в перечислении `CmdType`; то есть, `(1 << CMD_UPDATE) = 4` для `UPDATE`, `(1 << CMD_INSERT) = 8` для `INSERT` и `(1 << CMD_DELETE) = 16` для `DELETE`.

Если указатель на `IsForeignRelUpdatable` равен `NULL`, предполагается, что сторонние таблицы позволяют добавлять, изменять и удалять строки, если FDW предоставляет процедуры для функций `ExecForeignInsert`, `ExecForeignUpdate` или `ExecForeignDelete`, соответственно. Данная функция необходима, только если FDW поддерживает операции изменения для одних таблиц и не поддерживает для других. (Хотя для этого можно выдать ошибку в подпрограмме, выполняющей операцию, а не задействовать эту функцию. Однако данная функция позволяет корректно отражать поддержку изменений в представлениях `information_schema`.)

Некоторые операции добавления, изменений и удаления данных в сторонних таблицах можно соптимизировать, применив альтернативный набор интерфейсов. Обычные интерфейсы для операций добавления, изменения и удаления выбирают строки с удалённого сервера, а затем модифицируют их по одной. В некоторых случаях такой подход «строка за строкой» необходим, но он может быть не самым эффективным. Если есть возможность определить на стороннем сервере, какие строки должны модифицироваться, собственно не считывая их, и если никакие локальные структуры (локальные триггеры уровня строк, хранимые генерируемые столбцы или ограничения `WITH CHECK OPTION` из родительских представлений) на эту операцию не влияют, её можно организовать так, чтобы она выполнялась целиком на удалённом сервере. Это позволяют осуществить описанные ниже интерфейсы.

```
bool
PlanDirectModify(PlannerInfo *root,
                 ModifyTable *plan,
                 Index resultRelation,
                 int subplan_index);
```

Определяет, возможно ли безопасно выполнить прямую модификацию на удалённом сервере. Если да, возвращает `true`, произведя требуемые для этого операции планирования. В противном случае возвращает `false`. Эта необязательная функция вызывается во время планирования запроса. Если результат этой функции положительный, на стадии выполнения будут вызываться `BeginDirectModify`, `IterateDirectModify` и `EndDirectModify`. Иначе модификация таблиц будет осуществляться посредством функций изменения, описанных выше. Данная функция принимает те же параметры, что и `PlanForeignModify`.

Для осуществления прямой модификации на удалённом сервере эта функция должна подставить в целевой подплан узел `ForeignScan`, выполняющий прямую модификацию на удалённом сервере. Значения полей `operation` и `resultRelation` структуры `ForeignScan` должны быть заданы соответствующим образом. В поле `operation` должно быть установлено соответствующее значение перечисления `CmdType`: то есть, `CMD_UPDATE` для `UPDATE`, `CMD_INSERT` для `INSERT` и `CMD_DELETE` для `DELETE`; в поле `resultRelation` нужно скопировать аргумент `resultRelation`.

За дополнительными сведениями обратитесь к Разделу 55.4.

Если указатель на `PlanDirectModify` равен `NULL`, сервер не будет пытаться произвести прямую модификацию.

```
void
BeginDirectModify(ForeignScanState *node,
                  int eflags);
```

Подготавливает прямую модификацию на удалённом сервере. Эта функция вызывается при запуске исполнителя. Она должна выполнить все подготовительные действия, необходимые для осуществления прямой модификации (модификация должна начаться с первым вызовом `IterateDirectModify`). Узел `ForeignScanState` уже был создан, но его поле `fdw_state` по-прежнему NULL. Информацию о модифицируемой таблице можно получить через узел `ForeignScanState` (в частности, из нижележащего узла `ForeignScan`, содержащего частную информацию FDW, заданную функцией `PlanDirectModify`). Параметр `eflags` содержит битовые флаги, описывающие режим работы исполнителя для этого узла плана.

Заметьте, что когда `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` не равно нулю, эта функция не должна выполнять какие-либо внешне проявляющиеся действия; она должна сделать только то, что необходимо для получения состояния узла, подходящего для `ExplainDirectModify` и `EndDirectModify`.

Если указатель на `BeginDirectModify` равен `NULL`, сервер не будет пытаться произвести прямую модификацию.

```
TupleTableSlot *
IterateDirectModify(ForeignScanState *node);
```

Когда в запросе `INSERT`, `UPDATE` или `DELETE` отсутствует предложение `RETURNING`, просто возвращает NULL после прямой модификации на удалённом сервере. Когда в запросе есть это предложение, выбирает одну строку результата с данными, требующимися для вычисления `RETURNING`, и возвращает её в слоте таблицы кортежей (для этой цели следует использовать `ScanTupleSlot`, переданный с узлом). Данные, которые были фактически добавлены, изменены или удалены, нужно сохранить в `node->resultRelInfo->ri_projectReturning->pi_exprContext->ecxt_scantuple`. Возвращает NULL, если строк больше нет. Заметьте, что эта функция вызывается в контексте кратковременной памяти, который будет сбрасываться между вызовами. Если вам нужна более долгоживущая память, создайте соответствующий контекст в `BeginDirectModify` либо используйте `es_query_cxt` из переданной с узлом структуры `EState`.

Возвращаемые строки должны соответствовать целевому списку `fdw_scan_tlist`, если он передаётся, а в противном случае — типу строки изменяемой сторонней таблицы. Если вы решите для оптимизации не возвращать ненужные столбцы, не требующиеся для получения `RETURNING`, в их позиции нужно вставить NULL, либо сформировать список `fdw_scan_tlist` без этих столбцов.

Независимо от того, есть ли в запросе это предложение или нет, число строк, возвращаемых запросом, должно увеличиваться самой FDW. Когда этого предложения в запросе нет, FDW должна также увеличивать число строк для узла `ForeignScanState` в случае `EXPLAIN ANALYZE`.

Если указатель на `IterateDirectModify` равен `NULL`, сервер не будет пытаться произвести прямую модификацию.

```
void
EndDirectModify(ForeignScanState *node);
```

Очищает ресурсы после непосредственной модификации на удалённом сервере. Обычно при этом не нужно освобождать память, выделенную через palloc, но например, открытые файлы и подключения к удалённому серверу следует закрыть.

Если указатель на `EndDirectModify` равен `NULL`, сервер не будет пытаться произвести прямую модификацию.

55.2.5. Подпрограммы FDW для TRUNCATE

```
void
ExecForeignTruncate(List *rels,
                    DropBehavior behavior,
                    bool restart_seqs);
```

Опустошает сторонние таблицы. Эта функция вызывается, когда для сторонней таблицы выполняется TRUNCATE. В `rels` передаётся список структур `Relation`, описывающих сторонние таблицы, подлежащие опустошению.

Значением `behavior` может быть либо `DROP_RESTRICT`, либо `DROP_CASCADE`, в зависимости от того, какое указание (`RESTRICT` или `CASCADE`, соответственно) присутствовало в исходной команде `TRUNCATE`.

Если `restart_seqs` имеет значение `true`, в исходной команде `TRUNCATE` было запрошено поведение `RESTART IDENTITY`, в противном случае было запрошено поведение `CONTINUE IDENTITY`.

Заметьте, что параметры `ONLY`, указанные в исходной команде `TRUNCATE`, не передаются в функцию `ExecForeignTruncate`. Подобным образом работают функции-обработчики `SELECT`, `UPDATE` и `DELETE` для сторонней таблицы.

`ExecForeignTruncate` вызывается отдельно для каждого стороннего сервера, на котором должны опустошаться сторонние таблицы. Это означает, что все сторонние таблицы, включённые в `rels`, должны относиться к одному серверу.

Если указатель на `ExecForeignTruncate` равен `NULL`, опустошить данные в сторонней таблице не удастся, а в ответ будет выдаваться сообщение об ошибке.

55.2.6. Подпрограммы FDW для блокировки строк

Если FDW желает поддержать функцию *поздней блокировки строк* (описанную в Разделе 55.5), она должна предоставить следующие реализующие подпрограммы:

```
RowMarkType
GetForeignRowMarkType(RangeTblEntry *rte,
                      LockClauseStrength strength);
```

Сообщает, какой вариант пометки строк будет использоваться для сторонней таблицы. Здесь `rte` представляет узел `RangeTblEntry` для таблицы, а `strength` описывает силу блокировки, запрошенную соответствующим предложением `FOR UPDATE/SHARE`, если оно имеется. Результатом должно быть значение перечисления `RowMarkType`.

Эта функция вызывается в процессе планирования запроса для каждой сторонней таблицы, которая участвует в запросе `UPDATE`, `DELETE` или `SELECT FOR UPDATE/SHARE`, и не является целевой в запросе `UPDATE` или `DELETE`.

Если указатель `GetForeignRowMarkType` равен `NULL`, всегда выбирается вариант `ROW_MARK_COPY`. (Вследствие этого, функция `RefetchForeignRow` никогда не будет вызываться, так что и её задавать не нужно.)

За подробностями обратитесь к Разделу 55.5.

```
void
RefetchForeignRow(EState *estate,
                  ExecRowMark *erm,
                  Datum rowid,
                  TupleTableSlot *slot,
                  bool *updated);
```

Повторно считывает один кортеж из сторонней таблицы после блокировки, если она требуется. В `estate` передаётся глобальное состояние выполнения запроса. В `erm` передаётся структура `ExecRowMark`, описывающая целевую стороннюю таблицу и тип запрашиваемой блокировки (если она требуется). В параметре `slot` при вызове не содержится ничего полезного, но в него можно поместить возвращаемый кортеж. Параметр `updated` является выходным.

Эта функция должна сохранить кортеж в переданном слоте или очистить его, если получить блокировку строки не удаётся. Тип запрашиваемой блокировки строки определяется значением `erm->markType`, которое было до этого возвращено функцией `GetForeignRowMarkType`. (Вариант `ROW_MARK_REFERENCE` означает, что нужно просто повторно выбрать кортеж, не запрашивая никакую блокировку, а `ROW_MARK_COPY` никогда не поступает в эту подпрограмму.)

Кроме того, переменной `*updated` следует присвоить `true`, если была считана изменённая версия кортежа, а не версия, полученная ранее. (Если FDW не знает этого наверняка, рекомендуется всегда возвращать `true`.)

Заметьте, что по умолчанию в случае неудачи при попытке получить блокировку строки должна выдаваться ошибка; пустой слот может возвращаться, только если в `erm->waitPolicy` выбран вариант `SKIP LOCKED`.

В `rowid` передаётся значение `ctid`, полученное ранее для строки, которую нужно считать повторно. Хотя значение `rowid` передаётся в виде `Datum`, в настоящее время это может быть только `tid`. Такой интерфейс функции выбран с расчётом на то, чтобы в будущем в качестве идентификаторов строк могли приниматься и другие типы данных.

Если указатель на `RefetchForeignRow` равен `NULL`, повторно выбрать данные не удастся, в ответ будет выдаваться сообщение об ошибке.

За подробностями обратитесь к Разделу 55.5.

```
bool
RecheckForeignScan(ForeignScanState *node,
                   TupleTableSlot *slot);
```

Перепроверяет, соответствует ли по-прежнему ранее возвращённый кортеж применимым условиям сканирования и соединения, и возможно выдаёт изменённую версию кортежа. Для обёрток сторонних данных, которые не выносят соединение наружу, обычно удобнее присвоить этому указателю `NULL` и задать `fdw_recheck_quals`. Однако когда внешние соединения выносятся наружу, недостаточно повторно применить к результирующему кортежу проверки, относящиеся ко всем базовым таблицам, даже если присутствуют все атрибуты, так как невыполнение некоторого условия может приводить и к обнулению некоторых атрибутов, а не только исключению этого кортежа. `RecheckForeignScan` может перепроверить условия и возвратить true, если они по-прежнему выполняются, или false в противном случае, но также она может записать в переданный слот кортеж на замену предыдущему.

Чтобы вынести соединение наружу, обёртка сторонних данных обычно конструирует альтернативный план локального соединения, применяемый только для перепроверок; он становится внешним подпланом узла `ForeignScan`. Когда требуется перепроверка, может быть выполнен этот подплан и результирующий кортеж сохранён в слоте. Этот план может не быть эффективным, так как ни одна базовая таблица не выдаст больше одной строки; например, он может реализовывать все соединения в виде вложенных циклов. Для поиска подходящего локального пути соединения в существующих путях можно воспользоваться функцией `GetExistingLocalJoinPath`. Функция `GetExistingLocalJoinPath` ищет непараметризованный путь в списке путей заданного отношения соединения. (Если такой путь не находится, она возвращает NULL, и в этом случае обёртка сторонних данных может построить локальный путь сама или решить не создавать пути доступа для этого соединения.)

55.2.7. Подпрограммы FDW для `EXPLAIN`

```
void
ExplainForeignScan(ForeignScanState *node,
                   ExplainState *es);
```

Дополняет вывод `EXPLAIN` для сканирования сторонней таблицы. Эта функция может вызывать `ExplainPropertyText` и связанные функции и добавлять поля в вывод `EXPLAIN`. Поля флагов в `es` позволяют определить, что именно выводить, а для выдачи статистики времени выполнения в случае с `EXPLAIN ANALYZE` можно проанализировать состояние узла `ForeignScanState`.

Если указатель `ExplainForeignScan` равен `NULL`, никакая дополнительная информация при `EXPLAIN` не выводится.

```
void
ExplainForeignModify(ModifyTableState *mtstate,
                     ResultRelInfo *rinfo,
                     List *fdw_private,
                     int subplan_index,
                     struct ExplainState *es);
```

Дополняет вывод `EXPLAIN` для изменений в сторонней таблице. Эта функция может вызывать `ExplainPropertyText` и связанные функции и добавлять поля в вывод `EXPLAIN`. Поля флагов в `es` позволяют определить, что именно выводить, а для выдачи статистики времени выполнения в случае с `EXPLAIN ANALYZE` можно проанализировать состояние узла `ModifyTableState`. Первые четыре аргумента у этой функции те же, что и у `BeginForeignModify`.

Если указатель `ExplainForeignModify` равен `NULL`, никакая дополнительная информация при `EXPLAIN` не выводится.

```
void
ExplainDirectModify(ForeignScanState *node,
                    ExplainState *es);
```

Дополняет вывод `EXPLAIN` для прямой модификации данных на удалённом сервере. Эта функция может вызывать `ExplainPropertyText` и связанные функции и добавлять поля в вывод `EXPLAIN`. Поля флагов в `es` позволяют определить, что именно выводить, а для выдачи статистики времени выполнения в случае `EXPLAIN ANALYZE` можно проанализировать состояние узла `ForeignScanState`.

Если указатель `ExplainDirectModify` равен `NULL`, никакая дополнительная информация при `EXPLAIN` не выводится.

55.2.8. Подпрограммы FDW для `ANALYZE`

```
bool
AnalyzeForeignTable(Relation relation,
                    AcquireSampleRowsFunc *func,
                    BlockNumber *totalpages);
```

Эта функция вызывается, когда для сторонней таблицы выполняется ANALYZE. Если FDW может собрать статистику для этой сторонней таблицы, эта функция должна вернуть `true` и передать в *`func`* указатель на функцию, которая будет выдавать строки выборки из таблицы, а в *`totalpages`* ожидаемый размер таблицы в страницах. В противном случае эта функция должна вернуть `false`.

Если FDW не поддерживает сбор статистики ни для каких таблиц, в `AnalyzeForeignTable` можно установить значение `NULL`.

Функция выдачи выборки, если она предоставляется, должна иметь следующую сигнатуру:

```
int
AcquireSampleRowsFunc(Relation relation,
                      int elevel,
                      HeapTuple *rows,
                      int targrows,
                      double *totalrows,
                      double *totaldeadrows);
```

Она должна выбирать из таблицы максимум *`targrows`* строк и помещать их в переданный вызывающим кодом массив *`rows`*. Возвращать она должна фактическое число выбранных строк. Кроме того, эта функция должна сохранить общее количество актуальных и «мёртвых»строк в таблице в выходных параметрах *`totalrows`* и *`totaldeadrows`*, соответственно. (Если для данной FDW нет понятия «мёртвых» строк, в *`totaldeadrows`* нужно записать 0.)

55.2.9. Подпрограммы FDW для `IMPORT FOREIGN SCHEMA`

```
List *
ImportForeignSchema(ImportForeignSchemaStmt *stmt, Oid serverOid);
```

Получает список команд, создающих сторонние таблицы. Эта функция вызывается при выполнении команды IMPORT FOREIGN SCHEMA; ей передаётся дерево разбора этого оператора и OID целевого стороннего сервера. Она должна вернуть набор строк C, в каждой из которых должна содержаться команда CREATE FOREIGN TABLE. Эти строки будут разобраны и выполнены ядром сервера.

В структуре `ImportForeignSchemaStmt` поле `remote_schema` задаёт имя удалённой схемы, из которой импортируются таблицы. Поле `list_type` устанавливает, как фильтровать имена таблиц: вариант `FDW_IMPORT_SCHEMA_ALL` означает, что нужно импортировать все таблицы в удалённой схеме (в этом случае поле `table_list` пустое), `FDW_IMPORT_SCHEMA_LIMIT_TO` означает, что нужно импортировать только таблицы, перечисленные в `table_list`, и `FDW_IMPORT_SCHEMA_EXCEPT` означает, что нужно исключить таблицы, перечисленные в списке `table_list`. В поле `options` передаётся список параметров для процесса импорта. Значение этих параметров определяется самой FDW. Например, у FDW может быть параметр, определяющий, нужно ли сохранять у импортируемых столбцов атрибут `NOT NULL`. Эти параметры могут не иметь ничего общего с параметрами, которые принимает FDW в качестве параметров объектов базы.

FDW может игнорировать поле `local_schema` в `ImportForeignSchemaStmt`, так как ядро сервера само вставит это имя в разобранные команды `CREATE FOREIGN TABLE`.

Также, FDW может не выполнять сама фильтрацию по полям `list_type` и `table_list`, так как ядро сервера автоматически пропустит все возвращённые команды для таблиц, исключённых по заданным критериям. Однако часто лучше сразу избежать лишней работы, не формируя команды для исключаемых таблиц. Для проверки, удовлетворяет ли фильтру заданное имя сторонней таблицы, может быть полезна функция `IsImportableForeignTable()`.

Если FDW не поддерживает импорт определений таблиц, указателю `ImportForeignSchema` можно присвоить `NULL`.

55.2.10. Подпрограммы FDW для параллельного выполнения

Узел `ForeignScan` может, хотя это не требуется, поддерживать параллельное выполнение. Параллельный `ForeignScan` будет выполняться в нескольких процессах и должен возвращать одну строку только единожды. Для этого взаимодействующие процессы могут координировать свои действия через фиксированного размера блоки в динамической разделяемой памяти. Эта разделяемая память не будет гарантированно отображаться по одному адресу в разных процессах, так что она не может содержать указатели. Все следующие функции являются необязательными, но большинство из них необходимы при реализации поддержки параллельного выполнения.

```
bool
```

```
IsForeignScanParallelSafe(PlannerInfo *root, RelOptInfo *rel,
                          RangeTblEntry *rte);
```

Проверяет, будет ли сканирование выполняться параллельным исполнителем. Эта функция будет вызываться, только когда планировщик считает, что параллельный план принципиально возможен, и должна возвращать true, если такое сканирование может безопасно выполняться параллельным исполнителем. Обычно это не так, если удалённый источник данных является транзакционным. Но возможно исключение, когда в подключении рабочего процесса к этому источнику каким-то образом используется тот же транзакционный контекст, что и в ведущем процессе.

Если эта функция не определена, считается, что сканирование должно происходить в ведущем процессе. Заметьте, что возвращённое значение true не означает, что само сканирование может выполняться в параллельном режиме, а только то, что сканирование будет производится в параллельном исполнителе. Таким образом, может быть полезно определить этот обработчик, даже если параллельное выполнение не поддерживается.

```
Size
EstimateDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt);
```

Оценивает объём динамической разделяемой памяти, которая потребуется для параллельной операции. Это значение может превышать объём, который будет занят фактически, но не должно быть меньше. Возвращаемое значение задаётся в байтах. Эта функция является необязательной и может быть опущена, если не требуется; но в этом случае должны быть также опущены следующие три функции, так как для FDW не будет выделена разделяемая память.

```
void
InitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                         void *coordinate);
```

Инициализирует динамическую разделяемую память, которая потребуется для параллельной операции. `coordinate` указывает на область разделяемой памяти размера, равного возвращаемому значению `EstimateDSMForeignScan`. Эта функция является необязательной и может быть опущена, если не требуется.

```
void
ReInitializeDSMForeignScan(ForeignScanState *node, ParallelContext *pcxt,
                           void *coordinate);
```

Заново инициализирует динамическую разделяемую память, требуемую для параллельной операции, перед тем как будет повторно просканирован узел чтения сторонних данных. Эта функция является необязательной и может быть опущена, если не требуется. В этой функции рекомендуется сбрасывать только общее состояние, а в функции `ReScanForeignScan` сбрасывать только локальное. В настоящее время эта функция будет вызываться перед `ReScanForeignScan`, но лучше на этот порядок не рассчитывать.

```
void
InitializeWorkerForeignScan(ForeignScanState *node, shm_toc *toc,
                            void *coordinate);
```

Инициализирует локальное состояние параллельного исполнителя на основе общего состояния, заданного ведущим исполнителем во время `InitializeDSMForeignScan`. Эта функция является необязательной и может быть опущена, если не требуется.

```
void
ShutdownForeignScan(ForeignScanState *node);
```

Освобождает ресурсы, когда становится понятно, что этот узел больше не будет выполняться. Этот обработчик вызывается не во всех случаях; иногда может вызываться только `EndForeignScan`. Так как сегмент DSM, используемый параллельным запросом, освобождается сразу после вызова этого обработчика, обёртки сторонних данных, которым нужно выполнять некоторые действия до ликвидации сегмента DSM, должны реализовывать этот метод.

55.2.11. Подпрограммы FDW для асинхронного выполнения

Узел `ForeignScan` может дополнительно поддерживать асинхронное выполнение. Перечисленные ниже функции являются необязательными, но все они нужны для обеспечения поддержки асинхронного выполнения.

```
bool
IsForeignPathAsyncCapable(ForeignPath *path);
```

Проверяет, возможно ли для данного пути `ForeignPath` асинхронное сканирование нижележащего стороннего отношения. Эта функция будет вызываться только в конце планирования запроса, если данный путь является прямым потомком пути `AppendPath` и когда планировщик считает, что асинхронное выполнение улучшает производительность. Если для данного пути возможно асинхронное сканирование, эта функция должна возвращать true.

Если эта функция не определена, предполагается, что для данного пути стороннее отношение сканируется с помощью функции `IterateForeignScan`. (Это означает, что функции обратного вызова, описанные ниже, никогда не будут вызываться, поэтому их также не нужно задавать.)

```
void
ForeignAsyncRequest(AsyncRequest *areq);
```

Асинхронно выдаёт один кортеж из узла `ForeignScan`. `areq` — это структура `AsyncRequest`, описывающая узел `ForeignScan` и родительский узел `Append`, запросивший у него кортеж. Эта функция должна поместить кортеж в слот, переданный в `areq->result`, и записать `true` в `areq->request_complete`, если кортеж готов. Если же он ещё не готов и необходимо дождаться внешнего по отношению к главному серверу события, например получения данных из сети, эта функция должна записать `false` в `areq->request_complete` и `true` в `areq->callback_pending`, чтобы для узла `ForeignScan` были вызваны описанные ниже функции-обработчики. Если кортежей больше нет, нужно передать в качестве слота NULL или пустой слот, а в `areq->request_complete` записать `true`. Для формирования выходных параметров в `areq` рекомендуется использовать функцию `ExecAsyncRequestDone` или `ExecAsyncRequestPending`.

```
void
ForeignAsyncConfigureWait(AsyncRequest *areq);
```

Настраивает событие файлового дескриптора, которого хочет дождаться узел `ForeignScan` Эта функция будет вызываться, только когда `ForeignScan` установил флаг `areq->callback_pending`, и должна добавить событие в набор `as_eventset` родительского узла `Append`, описываемого параметром `areq`. Когда произойдёт событие файлового дескриптора, будет вызвана функция `ForeignAsyncNotify`.

```
void
ForeignAsyncNotify(AsyncRequest *areq);
```

Обрабатывает соответствующее произошедшее событие, затем асинхронно выдаёт один кортеж из узла `ForeignScan`. Эта функция должна устанавливать выходные параметры в `areq` так же, как это делает функция `ForeignAsyncRequest`.

55.2.12. Подпрограммы FDW для изменения параметризации путей

```
List *
ReparameterizeForeignPathByChild(PlannerInfo *root, List *fdw_private,
                                 RelOptInfo *child_rel);
```

Эта функция вызывается при преобразовании пути, параметризованного самым верхним родителем данного дочернего отношения `child_rel`, в путь, параметризованный дочерним отношением. Она используется для изменения параметров любых путей или трансляции любых узлов выражений, сохранённых в поле `fdw_private` переданной структуры `ForeignPath`. Этот обработчик может по мере необходимости использовать `reparameterize_path_by_child`, `adjust_appendrel_attrs` или `adjust_appendrel_attrs_multilevel`.

55.3. Вспомогательные функции для обёрток сторонних данных

Ядро сервера экспортирует набор полезных вспомогательных функций, которые позволяют разработчикам обёрток сторонних данных легко обращаться к атрибутам объектов, связанных с FDW, например, к параметрам FDW. Чтобы использовать эти функции, необходимо включить в исходный файл заголовочный файл *`foreign/foreign.h`*. В этом заголовочном файле также определяются типы структур, возвращаемых этими функциями.

```
ForeignDataWrapper *
GetForeignDataWrapperExtended(Oid fdwid, bits16 flags);
```

Эта функция возвращает объект `ForeignDataWrapper` для обёртки сторонних данных с указанным OID. Объект `ForeignDataWrapper` содержит свойства FDW (они описаны в `foreign/foreign.h`). В аргументе `flags` передаётся полученная битовым сложением маска, отмечающая дополнительный набор параметров. Её значением может быть `FDW_MISSING_OK`, указывающее, что вместо ошибки в случае неопределённого объекта должно возвращается `NULL`.

```
ForeignDataWrapper *
GetForeignDataWrapper(Oid fdwid);
```

Эта функция возвращает объект `ForeignDataWrapper` для обёртки сторонних данных с указанным OID. Объект `ForeignDataWrapper` содержит свойства FDW (они описаны в `foreign/foreign.h`).

```
ForeignServer *
GetForeignServerExtended(Oid serverid, bits16 flags);
```

Эта функция возвращает объект `ForeignServer` для стороннего сервера с указанным OID. Объект `ForeignServer` содержит свойства сервера (они описаны в `foreign/foreign.h`). В аргументе `flags` передаётся полученная битовым сложением маска, отмечающая дополнительный набор параметров. Её значением может быть `FSV_MISSING_OK`, указывающее, что вместо ошибки в случае неопределённого объекта должно возвращается `NULL`.

```
ForeignServer *
GetForeignServer(Oid serverid);
```

Эта функция возвращает объект `ForeignServer` для стороннего сервера с указанным OID. Объект `ForeignServer` содержит свойства сервера (они описаны в `foreign/foreign.h`).

```
UserMapping *
GetUserMapping(Oid userid, Oid serverid);
```

Эта функция возвращает объект `UserMapping` для сопоставления пользователя, которое определено для указанной роли на указанном сервере. (Если сопоставление для указанной роли отсутствует, она возвращает сопоставление для `PUBLIC` или выдаёт ошибку, если его нет.) Объект `UserMapping` содержит свойства сопоставления пользователя (они описаны в `foreign/foreign.h`).

```
ForeignTable *
GetForeignTable(Oid relid);
```

Эта функция возвращает объект `ForeignTable` для сторонней таблицы с указанным OID. Объект `ForeignTable` содержит свойства сторонней таблицы (они описаны в `foreign/foreign.h`).

```
List *
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

Эта функция возвращает параметры FDW уровня столбцов для столбца из таблицы с указанным OID сторонней таблицы и указанным номером, в виде списка `DefElem`. Если для столбца не определены параметры, возвращается NULL.

В дополнение к функциям, выбирающим объекты по OID, для некоторых объектов добавлены функции поиска по именам:

```
ForeignDataWrapper *
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

Эта функция возвращает объект `ForeignDataWrapper` для обёртки сторонних данных с указанным именем. В случае отсутствия такой обёртки возвращается NULL, если missing_ok равно true, а иначе выдаётся ошибка.

```
ForeignServer *
GetForeignServerByName(const char *name, bool missing_ok);
```

Эта функция возвращает объект `ForeignServer` для стороннего сервера с указанным именем. В случае отсутствия такого сервера возвращается NULL, если missing_ok равно true, а иначе выдаётся ошибка.

55.4. Планирование запросов с обёртками сторонних данных

Процедуры в FDW, реализующие функции `GetForeignRelSize`, `GetForeignPaths`, `GetForeignPlan`, `PlanForeignModify`, `GetForeignJoinPaths`, `GetForeignUpperPaths` и `PlanDirectModify`, должны вписываться в работу планировщика Postgres Pro. Здесь даётся несколько замечаний о том, как это должно происходить.

Для уменьшения объёма выбираемых из сторонней таблицы данных (и как следствие, сокращения стоимости) может использоваться информация, поступающая в `root` и `baserel`. Особый интерес представляет поле `baserel->baserestrictinfo`, так как оно содержит ограничивающие условия (предложение `WHERE`), по которым можно отфильтровать выбираемые строки. (Сама FDW не обязательно должна применять эти ограничения, так как их может проверить и ядро исполнителя.) Список `baserel->reltarget->exprs` позволяет определить, какие именно столбцы требуется выбрать; но учтите, что в нём перечисляются только те столбцы, которые выдаются узлом плана `ForeignScan`, но не столбцы, которые задействованы в ограничивающих условиях и при этом не выводятся запросом.

Когда функциям планирования FDW требуется сохранять свою информацию, они могут использовать различные частные поля. Вообще, все структуры, которые FDW помещает в закрытые поля, должны выделяться функцией palloc, чтобы они автоматически освобождались при завершении планирования.

Для хранения информации, относящейся к определённой сторонней таблице, функции планирования FDW могут использовать поле `baserel->fdw_private`, которое может содержать указатель на `void`. Ядро планировщика никак не касается его, кроме того, что записывает в него NULL при создании узла `RelOptInfo`. Оно полезно для передачи информации из `GetForeignRelSize` в `GetForeignPaths` и/или из `GetForeignPaths` в `GetForeignPlan` и позволяет избежать повторных вычислений.

`GetForeignPaths` может обозначить свойства различных путей доступа, сохранив частную информацию в поле `fdw_private` узлов `ForeignPath`. Это поле `fdw_private` объявлено как указатель на список (`List`), но в принципе может содержать всё, что угодно, так как ядро планировщика его не касается. Однако лучше поместить в него данные, которые сможет представить функция `nodeToString`, для применения средств отладки, имеющихся на сервере.

`GetForeignPlan` может изучить поле `fdw_private` выбранного узла `ForeignPath` и сформировать списки `fdw_exprs` и `fdw_private`, которые будут помещены в узел `ForeignScan`, где они будут находиться во время выполнения запроса. Оба эти списка должны быть представлены в форме, которую способна копировать функция `copyObject`. Список `fdw_private` не имеет других ограничений и никаким образом не интерпретируется ядром сервера. Список `fdw_exprs`, если этот указатель не NULL, предположительно содержит деревья выражений, которые должны быть вычислены при выполнении запроса. Затем планировщик обрабатывает эти деревья, чтобы они были полностью готовы к выполнению.

`GetForeignPlan` обычно может скопировать полученный целевой список в узел плана как есть. Передаваемый список `scan_clauses` содержит те же предложения, что и `baserel->baserestrictinfo`, но, возможно, в другом порядке для более эффективного выполнения. В простых случаях FDW может просто убрать узлы `RestrictInfo` из списка `scan_clauses` (используя функцию `extract_actual_clauses`) и поместить все предложения в список ограничений узла плана, что будет означать, что эти предложения будут проверяться исполнителем во время выполнения. Более сложные FDW могут самостоятельно проверять некоторые предложения, и в этом случае такие предложения можно удалить из списка ограничений узла, чтобы исполнитель не тратил время на их перепроверку.

Например, FDW может распознавать некоторые предложения ограничений вида *`сторонняя_переменная = подвыражение`*, которые, по её представлению, могут выполняться на удалённом сервере с локально вычисленным значением *`подвыражения`*. Собственно выявление такого предложения должно происходить в функции `GetForeignPaths`, так как это влияет на оценку стоимости пути. Эта функция может включить в поле `fdw_private` конкретного пути указатель на узел `RestrictInfo` этого предложения. Затем `GetForeignPlan` удалит это предложение из `scan_clauses`, но добавит *`подвыражение`* в `fdw_exprs`, чтобы оно было приведено к исполняемой форме. Она также может поместить управляющую информацию в поле `fdw_private` плана узла, которая скажет исполняющим функциям, что делать во время выполнения. Запрос, передаваемый удалённому серверу, будет содержать что-то вроде `WHERE` *`сторонняя_переменная`* `= $1`, а значение параметра будет получено во время выполнения в результате вычисления дерева выражения `fdw_exprs`.

Все предложения, удаляемые из списка условий узла плана, должны быть добавлены в `fdw_recheck_quals` или перепроверены функцией `RecheckForeignScan` для обеспечения корректного поведения на уровне изоляции `READ COMMITTED`. Когда имеет место параллельное изменение в некоторой другой таблице, задействованной в запросе, исполнителю может потребоваться убедиться в том, что все исходные условия по-прежнему выполняются для кортежа, возможно, с другим набором значений параметров. Использовать `fdw_recheck_quals` обычно проще, чем реализовывать проверки внутри `RecheckForeignScan`, но этот метод недостаточен, когда внешние соединения выносятся наружу, так как вследствие перепроверки в соединённых кортежах могут обнуляться некоторые поля, но сами кортежи не будут исключаться.

Ещё одно поле `ForeignScan`, которое могут заполнять FDW, это `fdw_scan_tlist`, описывающее кортежи, возвращаемые обёрткой для этого узла плана. Для простых сторонних таблиц в него можно записать `NIL`, из чего будет следовать, что возвращённые кортежи имеют тип, объявленный для сторонней таблицы. Отличное от `NIL` значение должно указывать на список целевых элементов (список структур `TargetEntry`), содержащий переменные и/или выражения, представляющие возвращаемые столбцы. Это можно использовать, например, чтобы показать, что FDW опустила некоторые столбцы, которые по её наблюдению не нужны для запроса. Также, если FDW может вычислить выражения, используемые в запросе, более эффективно, чем это можно сделать локально, она должна добавить эти выражения в список `fdw_scan_tlist`. Заметьте, что планы соединения (полученные из путей, созданных функцией `GetForeignJoinPaths`) должны всегда заполнять `fdw_scan_tlist`, описывая набор столбцов, которые они будут возвращать.

FDW должна всегда строить минимум один путь, зависящий только от предложений ограничения таблицы. В запросах с соединением она может также построить пути, зависящие от ограничения соединения, например *`сторонняя_переменная = локальная_переменная`*. Такие предложения будут отсутствовать в `baserel->baserestrictinfo`; их нужно искать в списках соединений отношений. Путь, построенный с таким предложением, называется «параметризованным». Другие отношения, задействованные в выбранном предложении соединения, должны связываться с этим путём соответствующим значением `param_info`; для получения этого значения используется `get_baserel_parampathinfo`. В `GetForeignPlan` часть *`локальная_переменная`* предложения соединения будет добавлена в `fdw_exprs`, и затем, во время выполнения, это будет работать так же, как и обычное предложение ограничения.

Если FDW поддерживает удалённые соединения, `GetForeignJoinPaths` должна выдавать пути `ForeignPath` для потенциально удалённых соединений почти так же, как это делает `GetForeignPaths` для базовых таблиц. Информация о выбранном соединении может быть передана функции `GetForeignPlan` так же, как было описано выше. Однако поле `baserestrictinfo` неприменимо к отношениям соединения; вместо этого соответствующие предложения соединения для конкретного соединения передаются в `GetForeignJoinPaths` в отдельном параметре (`extra->restrictlist`).

FDW может дополнительно поддерживать прямое выполнение некоторых действий плана, находящихся выше уровня сканирований и соединений, например, группировки или агрегирования. Для реализации этой возможности FDW должна сформировать пути и вставить их в соответствующее *верхнее отношение*. Например, путь, представляющий удалённое агрегирование, должен вставляться в отношение `UPPERREL_GROUP_AGG` с помощью `add_path`. Этот путь будет сравниваться по стоимости с локальным агрегированием, выполненным по результатам пути простого сканирования стороннего отношения (заметьте, что такой путь также должен быть сформирован, иначе во время планирования произойдёт ошибка). Если путь с удалённым агрегированием выигрывает, что, как правило, и происходит, он будет преобразован в план обычным образом, вызовом `GetForeignPlan`. Такие пути рекомендуется формировать в обработчике `GetForeignUpperPaths`, который вызывается для каждого верхнего отношения (то есть на каждом шаге обработки после сканирования/соединения), если все базовые отношения запроса выдаются одной обёрткой.

`PlanForeignModify` и другие обработчики, описанные в Подразделе 55.2.4, рассчитаны на то, что стороннее отношение будет сканироваться обычным способом, а затем отдельные изменения строк будут обрабатываться локальным узлом плана `ModifyTable`. Этот подход необходим в общем случае, когда для такого изменения требуется прочитать не только сторонние, но и локальные таблицы. Однако если операция может быть целиком выполнена сторонним сервером, FDW может построить путь, представляющий эту возможность, и вставить его в верхнее отношение `UPPERREL_FINAL`, где он будет конкурировать с подходом `ModifyTable`. Этот подход также должен применяться для реализации удалённого `SELECT FOR UPDATE`, вместо обработчиков блокировки строк, описанных Подразделе 55.2.6. Учтите, что путь, вставляемый в `UPPERREL_FINAL`, отвечает за реализацию *всех* аспектов поведения запроса.

При планировании запросов `UPDATE` или `DELETE` функции `PlanForeignModify` и `PlanDirectModify` могут обратиться к структуре `RelOptInfo` сторонней таблицы и воспользоваться информацией `baserel->fdw_private`, записанной ранее функциями планирования сканирования. Однако при запросе `INSERT` целевая таблица не сканируется, так что для неё `RelOptInfo` не заполняется. На список (`List`), возвращаемый функцией `PlanForeignModify`, накладываются те же ограничения, что и на список `fdw_private` в узле плана `ForeignScan`, то есть он должен содержать только такие структуры, которые способна копировать функция `copyObject`.

Команда `INSERT` с предложением `ON CONFLICT` не поддерживает указание объекта конфликта, так как уникальные ограничения или ограничения-исключения в удалённых таблицах неизвестны локально. Из этого, в свою очередь, вытекает, что предложение `ON CONFLICT DO UPDATE` не поддерживается, так как в нём это указание является обязательным.

55.5. Блокировка строк в обёртках сторонних данных

Если нижележащий механизм хранения FDW поддерживает концепцию блокировки отдельных строк, предотвращающую одновременное изменение этих строк, обычно имеет смысл реализовать в FDW установление блокировок на уровне строк в приближении, настолько близком к обычным таблицам Postgres Pro, насколько это возможно и практично. При этом нужно учитывать ряд замечаний.

Первое важное решение, которое нужно принять — будет ли реализована *ранняя блокировка* или *поздняя блокировка*. С ранней блокировкой строка блокируется, когда впервые считывается из нижележащего хранилища, тогда как с поздней блокировкой строка блокируется, только когда известно, что её нужно заблокировать. (Различие возникает из-за того, что некоторые строки могут быть отброшены локально проверяемыми условиями ограничений или соединений.) Ранняя блокировка гораздо проще и не требует дополнительных обращений к удалённому хранилищу, но может вызывать блокировку строк, которые можно было бы не блокировать, что может повлечь учащение конфликтов и даже неожиданные взаимоблокировки. Кроме того, поздняя блокировка возможна, только если блокируемая строка может быть однозначно идентифицирована позже. Поэтому в идентификаторе строки следует идентифицировать определённую версию строки, как это делает TID в Postgres Pro.

По умолчанию Postgres Pro игнорирует возможности блокировки, обращаясь к FDW, но FDW может установить ранние блокировки и без явной поддержки со стороны ядра. Функции, описанные в Подразделе 55.2.6, которые были добавлены в API в Postgres Pro 9.5, позволяют FDW применять поздние блокировки, если она этого пожелает.

Также следует учесть, что в режиме изоляции `READ COMMITTED` серверу Postgres Pro может потребоваться перепроверить условия ограничений и соединения с изменённой версией некоторого целевого кортежа. Для перепроверки условий соединения требуется повторно получить копии исходных строк, которые ранее были соединены в целевой кортеж. В случае со стандартными таблицами Postgres Pro для этого в список столбцов, проходящих через соединение, включаются TID из исходных таблиц, а затем исходные строки извлекаются заново при необходимости. При таком подходе набор данных соединения остаётся компактным, но требуется недорогая операция повторного чтения строк, а также возможность однозначно идентифицировать повторно считываемую версию строки по TID. Поэтому по умолчанию при работе со сторонними таблицами в список столбцов, проходящих через соединение, включается копия всей строки, извлекаемой из сторонней таблицы. Это не накладывает специальных требований на FDW, но может привести к снижению производительности при соединении слиянием или по хешу. FDW, которая может удовлетворить требованиям повторного чтения, может реализовать первый вариант.

Для команд `UPDATE` или `DELETE` со сторонней таблицей рекомендуется, чтобы операция `ForeignScan` в целевой таблице выполняла раннюю блокировку строк, которые она выбирает, возможно, используя аналог `SELECT FOR UPDATE`. FDW может определить, является ли таблица целевой таблицей команд `UPDATE`/`DELETE`, во время планирования, сравнив её relid с `root->parse->resultRelation`, или во время планирования, вызвав `ExecRelationIsTargetRelation()`. Также возможно выполнять позднюю блокировку в обработчике `ExecForeignUpdate` или `ExecForeignDelete`, но специальной поддержки для этого нет.

Для сторонних таблиц, блокировка которых запрашивается командой `SELECT FOR UPDATE/SHARE`, операция `ForeignScan` так же может произвести раннюю блокировку, выбрав кортежи, используя аналог `SELECT FOR UPDATE/SHARE`. Чтобы вместо этого произвести позднюю блокировку, предоставьте подпрограммы-обработчики, описанные в Подразделе 55.2.6. В `GetForeignRowMarkType` выберите вариант отметки строк `ROW_MARK_EXCLUSIVE`, `ROW_MARK_NOKEYEXCLUSIVE`, `ROW_MARK_SHARE` или `ROW_MARK_KEYSHARE`, в зависимости от запрошенной силы блокировки. (Код ядра будет работать одинаково при любом из этих четырёх вариантов.) Затем вы сможете определить, должна ли сторонняя таблица блокироваться командой этого типа, вызвав функцию `get_plan_rowmark` во время планирования либо `ExecFindRowMark` во время выполнения; нужно проверить не только, что возвращённая структура rowmark отлична от NULL, но и что её поле `strength` не равно `LCS_NONE`.

Наконец, для сторонних таблиц, задействованных в командах `UPDATE`, `DELETE` или `SELECT FOR UPDATE/SHARE`, но не требующих блокировки строк, можно переопределить поведение по умолчанию, заключающееся в копировании строк целиком, выбрав в `GetForeignRowMarkType` вариант `ROW_MARK_REFERENCE`, получив значение силы блокировки `LCS_NONE`. В результате `RefetchForeignRow` будет вызываться с таким значением `markType`; она должна будет заново считывать строку, не запрашивая новую блокировку. (Если вы реализуете функцию `GetForeignRowMarkType`, но не хотите повторно считывать незаблокированные строки, выберите для `LCS_NONE` вариант `ROW_MARK_COPY`.)

Глава 56. Написание метода извлечения выборки таблицы

Реализация предложения `TABLESAMPLE` в Postgres Pro поддерживает подключение собственных методов извлечения выборки таблицы в дополнение к методам `BERNOULLI` и `SYSTEM`, которые требуются стандартом SQL. Метод выборки определяет, какие строки таблицы будут выбираться, когда используется предложение `TABLESAMPLE`.

На уровне SQL метод извлечения выборки таблицы представляется одной функцией SQL, обычно реализуемой на C, имеющей сигнатуру

```
method_name(internal) RETURNS tsm_handler
```

Имя функции будет совпадать с именем метода, указываемым в предложении `TABLESAMPLE`. Аргумент `internal` является фиктивным (в нём всегда передаётся ноль) и введён только для того, чтобы эту функцию нельзя было вызывать напрямую из команд SQL. Возвращать эта функция должна структуру типа `TsmRoutine` (выделенную вызовом palloc), содержащую указатели на опорные функции для метода извлечения выборки. Эти опорные функции представляют собой простые функции на C, которые не видны и не могут вызываться на уровне SQL. Эти опорные функции описаны в Разделе 56.1.

В дополнение к указателям на функции в структуре `TsmRoutine` должны задаваться следующие дополнительные поля:

`List *parameterTypes`

Это список OID, содержащий OID типов данных параметров, которые будут приниматься предложением `TABLESAMPLE` при использовании этого метода извлечения выборки. Например, для встроенных методов этот список содержит один элемент со значением `FLOAT4OID`, представляющий процент выборки. Другие методы могут иметь дополнительные или иные параметры.

`bool repeatable_across_queries`

Если это поле равно `true`, данный метод извлечения выборки может выдавать одинаковые выборки при последовательных запросах с одними и теми же параметрами и значением затравки `REPEATABLE` при условии неизменности содержимого таблицы. Если равно `false`, предложение `REPEATABLE` не будет приниматься с этим методом извлечения выборки.

`bool repeatable_across_scans`

Если это поле равно `true`, метод извлечения выборки может выдавать одинаковые выборки при последовательном сканировании в рамках одного запроса (предполагается неизменность параметров, значения затравки и снимка данных). Если равно `false`, планировщик не будет выбирать планы, требующие неоднократного сканирования выборки, так как это может привести к несогласованному результату запроса.

Методы извлечения выборки, включённые в стандартный дистрибутив, могут послужить хорошим примером, если вы хотите написать свой метод. Дополнительные методы вы можете найти в подкаталоге `contrib`.

56.1. Опорные функции метода извлечения выборки

Функция-обработчик TSM возвращает структуру `TsmRoutine` (выделенную вызовом palloc) с указателями на опорные функции, описанные ниже. Большинство этих функций обязательные, но некоторые — нет, и их указатели могут быть равны NULL.

```
void
SampleScanGetSampleSize (PlannerInfo *root,
                         RelOptInfo *baserel,
```

```
                                  List *paramexprs,
                                  BlockNumber *pages,
                                  double *tuples);
```

Эта функция вызывается во время планирования. Она должна рассчитать число страниц отношения, которые будут прочитаны при сканировании выборки, и число кортежей, выбираемых при сканировании. (Например, эти числа можно получить, оценив процент выбираемых данных, а затем умножив `baserel->pages` и `baserel->tuples` на это значение и округлив результат до целых.) Список `paramexprs` содержит выражения, переданные в параметрах предложению `TABLESAMPLE`. Если для целей оценивания нужны их значения, рекомендуется воспользоваться `estimate_expression_value()`, чтобы попытаться свести эти выражения к константам; но данная функция должна выдавать оценку размера, даже если это не удастся, и не должна выдавать ошибку, даже если считает переданные значения неверными (помните, что это только приблизительные оценки чисел, которые будут получены во время выполнения). Параметры `pages` и `tuples` являются выходными.

```
void
InitSampleScan (SampleScanState *node,
                int eflags);
```

Выполняет инициализацию перед выполнением узла плана SampleScan. Эта функция вызывается при запуске исполнителя. Она должна выполнить все подготовительные действия, необходимые для начала обработки. Узел `SampleScanState` уже был создан, но его поле `tsm_state` содержит NULL. Функция `InitSampleScan` может выделить через palloc область для любых внутренних данных, нужных методу извлечения выборки, и сохранить указатель на неё в `node->tsm_state`. Информацию о сканируемой таблице можно получить через другие поля узла `SampleScanState` (но заметьте, что дескриптор сканирования `node->ss.ss_currentScanDesc` ещё не настроен). Параметр `eflags` содержит битовые флаги, описывающие режим работы исполнителя для этого узла плана.

Когда `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` не равно нулю, собственно сканирование не будет выполняться, поэтому эта функция должна сделать только то, что необходимо для получения состояния узла, подходящего для `EXPLAIN` и `EndSampleScan`.

Эту функцию можно опустить (присвоить указателю NULL), тогда вся инициализация, необходимая для метода извлечения выборки, должна иметь место в `BeginSampleScan`.

```
void
BeginSampleScan (SampleScanState *node,
                 Datum *params,
                 int nparams,
                 uint32 seed);
```

Начинает выполнение сканирования выборки. Эта функция вызывается непосредственно перед первой попыткой выбрать кортеж и может вызываться повторно, если потребуется перезапустить сканирование. Информацию о сканируемой таблице можно получить через поля узла `SampleScanState` (но заметьте, что дескриптор сканирования `node->ss.ss_currentScanDesc` ещё не настроен). Массив `params`, длины `nparams`, содержит значения параметров, переданных в предложении `TABLESAMPLE`. Их количество и типы задаются в списке `parameterTypes` метода выборки, и они гарантированно не равны NULL. Параметр `seed` содержит значение затравки, которое этот метод должен учитывать при генерации любых случайных чисел; это либо хеш, полученный из значения `REPEATABLE`, если оно было передано, либо результат `random()` в противном случае.

Эта функция может скорректировать поля `node->use_bulkread` и `node->use_pagemode`. Если поле `node->use_bulkread` равно `true` (это значение по умолчанию), при сканировании будет использоваться стратегия доступа к буферу, ориентированная на переработку буферов после использования. Может быть разумным присвоить ему `false`, если при сканировании будет просматриваться только небольшой процент страниц. Если поле `node->use_pagemode` равно `true` (это значение по умолчанию), при сканировании проверка видимости будет выполняться в один проход для всех кортежей на каждой просматриваемой странице. Может иметь смысл присвоить ему `false`, если при сканировании выбирается только небольшой процент кортежей на странице. В результате будет выполняться меньше проверок видимости кортежей, хотя каждая проверка будет дороже, так как потребует расширенную блокировку.

Если метод выборки помечен как `repeatable_across_scans`, он должен быть способен выбирать при повторном сканировании тот же набор кортежей, что был выбран в первый раз, то есть новый вызов `BeginSampleScan` должен приводить к выборке тех же кортежей, что и предыдущий (если параметры `TABLESAMPLE` и значение затравки не меняются).

```
BlockNumber
NextSampleBlock (SampleScanState *node, BlockNumber nblocks);
```

Возвращает номер блока следующей сканируемой страницы либо `InvalidBlockNumber`, если страниц для сканирования не осталось.

Эту функцию можно опустить (присвоить её указателю NULL), в этом случае код ядра произведёт последовательное сканирование всего отношения. Такое сканирование может быть синхронизированным, так что метод выборки не должен полагать, что страницы отношения каждый раз просматриваются в одном и том же порядке.

```
OffsetNumber
NextSampleTuple (SampleScanState *node,
                 BlockNumber blockno,
                 OffsetNumber maxoffset);
```

Возвращает номер смещения следующего кортежа, выбираемого с указанной страницы, либо `InvalidOffsetNumber`, если кортежей для выборки не осталось. В `maxoffset` задаётся максимальный номер смещения, допустимый на этой странице.

Примечание

`NextSampleTuple` не говорит явно, для каких из номеров смещений в диапазоне `1 .. maxoffset` действительно содержатся актуальные кортежи. Это обычно не проблема, так как код ядра игнорирует запросы на выборку несуществующих или невидимых кортежей; это не должно приводить к отклонениям в выборке. Однако при необходимости функция может прочитать в поле `node->donetuples` количество кортежей, которые оказались актуальными и видимыми, из числа тех, что она выдала.

Примечание

Функция `NextSampleTuple` *не* должна полагать, что в `blockno` будет получен тот же номер страницы, что был выдан при последнем вызове `NextSampleBlock`. Этот номер определённо был выдан при каком-то предыдущем вызове `NextSampleBlock`, но код ядра может вызывать `NextSampleBlock` перед тем, как собственно сканировать страницы, для поддержки упреждающего чтения. Однако можно рассчитывать на то, что как только начнётся выборка кортежей с одной данной страницы, все последующие вызовы `NextSampleTuple` будут обращаться к этой странице, пока не будет возвращено значение `InvalidOffsetNumber`.

```
void
EndSampleScan (SampleScanState *node);
```

Завершает сканирование и освобождает ресурсы. Обычно при этом не нужно освобождать память, выделенную через palloc, но все видимые извне ресурсы должны быть очищены. Эту функцию чаще всего можно опустить (присвоить её указателю NULL), если таких ресурсов нет.

Глава 57. Написание провайдера нестандартного сканирования

Postgres Pro поддерживает набор экспериментальных средств, предназначенных для того, чтобы модули расширения могли добавлять в систему новые типы сканирования. В отличие от обёртки сторонних данных, которая должна знать, как сканировать только собственные таблицы, провайдер сканирования может реализовать нестандартный вариант сканирования любого отношения в системе. Обычно к написанию провайдера нестандартного сканирования подталкивает желание реализовать какую-то оптимизацию, не поддерживаемую основной системой, например, кеширование или аппаратное ускорение некоторого рода. В этой главе рассказывается, как написать свой провайдер нестандартного сканирования.

Процесс реализации нестандартного сканирования нового типа состоит из трёх этапов. Во-первых, во время планирования необходимо построить пути доступа, представляющие сканирование с предлагаемой стратегией. Во-вторых, если один из этих путей доступа выбирается планировщиком как оптимальная стратегия сканирования определённого отношения, этот путь доступа должен быть преобразован в план. Наконец, должно быть возможно выполнить этот план, получив при этом те же результаты, что были бы получены с любым другим путём доступа, выбранным для того же отношения.

57.1. Создание нестандартных путей сканирования

Провайдер нестандартного сканирования обычно добавляет пути для базового отношения, установив следующий обработчик, который вызывается после того, как ядро построит все пути, какие сможет построить для отношения (за исключением путей Gather и Gather Merge, которые создаются после этого вызова с тем, чтобы они могли использовать частичные пути, добавленные данным обработчиком):

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root,
                                             RelOptInfo *rel,
                                             Index rti,
                                             RangeTblEntry *rte);
extern PGDLLIMPORT set_rel_pathlist_hook_type set_rel_pathlist_hook;
```

Хотя эта функция-обработчик может изучать, изменять или удалять пути, сформированные основной системой, провайдер нестандартного сканирования обычно ограничивается созданием объектов `CustomPath` и добавлением их в `rel` с помощью `add_path` или `add_partial_path`, если это частичные пути. Провайдер нестандартного сканирования отвечает за инициализацию объекта `CustomPath`, который описан так:

```
typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List     *custom_paths;
    List     *custom_restrictinfo;
    List     *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

Поле `path` должно инициализироваться как для любого другого пути и включать оценку числа строк, стоимость запуска и общую, а также порядок сортировки, устанавливаемый этим путём. Поле `flags` задаёт битовую маску, которая указывает, поддерживает ли провайдер сканирования определённые необязательные возможности. Поле `flags` должно включать флаг `CUSTOMPATH_SUPPORT_BACKWARD_SCAN`, если нестандартный путь поддерживает сканирование назад, `CUSTOMPATH_SUPPORT_MARK_RESTORE`, если он поддерживает пометку позиции и её восстановление, и `CUSTOMPATH_SUPPORT_PROJECTION`, если он поддерживает проекции. (Если `CUSTOMPATH_SUPPORT_PROJECTION` не установлен, узлу сканирования будет предложено только выдать переменные для сканируемого отношения, а если этот флаг установлен, узел сканирования должен иметь возможность вычислять скалярные выражения с этими переменными.) В необязательном поле `custom_paths` задаётся список узлов `Path`, используемых данным узлом; они будут преобразованы планировщиком в узлы `Plan`. Как описано ниже, для отношений соединения также можно создавать нестандартные пути. В таком случае поле `custom_restrictinfo` следует использовать для хранения набора условий соединения, которые будут применяться к соединению, заменяемому нестандартным путём. В противном случае оно должно иметь значение NIL. В поле `custom_private` могут быть сохранены внутренние данные нестандартного пути. Сохранять их нужно в форме, которую может принять `nodeToString`, чтобы отладочные процедуры, пытающиеся вывести нестандартный путь, работали ожидаемым образом. Поле `methods` должно указывать на объект (обычно статически размещённый), реализующий требуемые методы нестандартного пути, которые подробно описаны ниже.

Провайдер нестандартного сканирования может также реализовать пути соединений. Как и для базовых отношений, такой путь должен выдавать тот же результат, какой был бы получен обычным соединением, которое он заменяет. Для этого провайдер соединения должен установить следующий обработчик, а затем внутри функции-обработчика создать пути `CustomPath` для отношения соединения.

```
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root,
                                              RelOptInfo *joinrel,
                                              RelOptInfo *outerrel,
                                              RelOptInfo *innerrel,
                                              JoinType jointype,
                                              JoinPathExtraData *extra);
extern PGDLLIMPORT set_join_pathlist_hook_type set_join_pathlist_hook;
```

Этот обработчик будет вызываться многократно для одного отношения соединения с разными сочетаниями внутренних и внешних отношений; задача обработчика — минимизировать при этом дублирующиеся операции.

Также обратите внимание, что набор предложений соединения, который передаётся как `extra->restrictlist`, варьируется в зависимости от комбинации внутренних и внешних отношений. Путь `CustomPath`, созданный для `joinrel`, должен содержать набор используемых им условий соединения, который будет использоваться планировщиком для преобразования пути `CustomPath` в план, если он был выбран как лучший путь для `joinrel`.

57.1.1. Обработчики пути нестандартного сканирования

```
Plan *(*PlanCustomPath) (PlannerInfo *root,
                         RelOptInfo *rel,
                         CustomPath *best_path,
                         List *tlist,
                         List *clauses,
                         List *custom_plans);
```

Преобразует нестандартный путь в законченный план. Возвращаемым значением обычно будет объект `CustomScan`, который этот обработчик должен разместить в памяти и инициализировать. За подробностями обратитесь к Разделу 57.2.

```
List *(*ReparameterizeCustomPathByChild) (PlannerInfo *root,
                                          List *custom_private,
                                          RelOptInfo *child_rel);
```

Этот обработчик вызывается при преобразовании пути, параметризованного самым верхним родителем данного дочернего отношения `child_rel`, в путь, параметризованный дочерним отношением. Он используется для изменения параметров любых путей или трансляции любых узлов выражений, сохранённых в поле `custom_private` переданной структуры `CustomPath`. Этот обработчик может по мере необходимости использовать `reparameterize_path_by_child`, `adjust_appendrel_attrs` или `adjust_appendrel_attrs_multilevel`.

57.2. Создание нестандартных планов сканирования

Нестандартное сканирование представляется в окончательном дереве плана в виде следующей структуры:

```
typedef struct CustomScan
{
    Scan      scan;
    uint32    flags;
    List     *custom_plans;
    List     *custom_exprs;
    List     *custom_private;
    List     *custom_scan_tlist;
    Bitmapset *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;
```

Объект в поле `scan` должен быть инициализирован, как и для любого другого сканирования, и включать оценки стоимости, целевые списки, условия и т. д. Поле `flags` содержит битовую маску с тем же значением, что и в `CustomPath`. В поле `custom_plans` могут быть сохранены дочерние узлы `Plan`. В `custom_exprs` могут быть сохранены деревья выражений, которые будут исправляться кодом в `setrefs.c` и `subselect.c`, а в `custom_private` следует сохранить другие внутренние данные, которые будут использоваться только самим провайдером нестандартного сканирования. Поле `custom_scan_tlist` может содержать NIL при сканировании базового отношения, что будет показывать, что нестандартное сканирование возвращает кортежи, соответствующие типу строк базового отношения. В противном случае оно должно указывать на целевой список, описывающий фактические кортежи. Список `custom_scan_tlist` должен устанавливаться при соединениях и может задаваться при сканировании, если провайдер сканирования может вычислять какие-либо выражения без переменных. Поле `custom_relids` заполняется ядром и задаёт набор отношений (индексов в списке отношений), которые обрабатывает данный узел сканирования; когда имеет место сканирование, а не соединение, в этом списке будет всего один элемент. Поле `methods` должно указывать на объект (обычно статически размещённый), реализующий требуемые методы нестандартного сканирования, которые подробнее описываются ниже.

Когда `CustomScan` сканирует одно отношение, в `scan.scanrelid` должен задаваться индекс сканируемой таблицы в списке отношений. Когда он заменяет соединение, поле `scan.scanrelid` должно быть нулевым.

Деревья планов должны поддерживать возможность копирования функцией `copyObject`, так что все данные, сохранённые в «дополнительных» полях, должны быть узлами, которые может обработать эта функция. Более того, провайдеры нестандартного сканирования не могут заменять структуру `CustomScan` расширенной структурой, её содержащей, что возможно с `CustomPath` или `CustomScanState`.

57.2.1. Обработчики плана нестандартного сканирования

```
Node *(*CreateCustomScanState) (CustomScan *cscan);
```

Выделяет структуру `CustomScanState` для заданного объекта `CustomScan`. Фактически выделенная область будет обычно больше, чем требуется для самой структуры `CustomScanState`, так как многие провайдеры могут включать её в расширенную структуру в качестве первого поля. В возвращаемом значении должны быть подходящим образом заполнены тег узла и поле `methods`, но другие поля на данном этапе должны быть обнулены; после того как `ExecInitCustomScan` произведёт базовую инициализацию, будет вызван обработчик `BeginCustomScan`, в котором провайдер нестандартного сканирования может выполнить все остальные требуемые действия.

57.3. Выполнение нестандартного сканирования

Когда выполняется узел `CustomScan`, его состояние представляется структурой `CustomScanState`, объявленной следующим образом:

```
typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;
```

Поле `ss` инициализируется как и для состояния любого другого сканирования, за исключением того, что когда это сканирование для соединения, а не для базового отношения, в `ss.ss_currentRelation` остаётся NULL. Поле `flags` содержит битовую маску с тем же значением, что и в `CustomPath` и `CustomScan`. Поле `methods` должно указывать на объект (обычно статически размещённый), реализующий требуемые методы состояния нестандартного сканирования, подробнее описанные ниже. Обычно структура `CustomScanState`, которой не нужно поддерживать `copyObject`, фактически включается в расширенную структуру в качестве её первого члена.

57.3.1. Обработчики выполнения нестандартного сканирования

```
void (*BeginCustomScan) (CustomScanState *node,
                         EState *estate,
                         int eflags);
```

Завершает инициализацию переданного объекта `CustomScanState`. Стандартные поля инициализируются в `ExecInitCustomScan`, но все внутренние поля должны инициализироваться здесь.

```
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
```

Считывает следующий кортеж. В случае наличия кортежей эта функция должна записать в `ps_ResultTupleSlot` следующий кортеж в текущем направлении сканирования и вернуть слот с кортежем. Если же кортежей больше нет, она должна вернуть `NULL` или пустой слот.

```
void (*EndCustomScan) (CustomScanState *node);
```

Очищает все внутренние данные, связанные с `CustomScanState`. Этот метод является обязательным, но он может ничего не делать, если такие данные отсутствуют или они будут очищены автоматически.

```
void (*ReScanCustomScan) (CustomScanState *node);
```

Возвращает позицию текущего сканирования в начало и подготавливает повторное сканирование отношения.

```
void (*MarkPosCustomScan) (CustomScanState *node);
```

Сохраняет текущую позицию сканирования, чтобы к ней впоследствии можно было вернуться, вызвав обработчик `RestrPosCustomScan`. Данный обработчик является необязательным и должен присутствовать, только если установлен флаг `CUSTOMPATH_SUPPORT_MARK_RESTORE`.

```
void (*RestrPosCustomScan) (CustomScanState *node);
```

Восстанавливает предыдущую позицию сканирования, сохранённую обработчиком `MarkPosCustomScan`. Данный обработчик является необязательным и должен присутствовать, только если установлен флаг `CUSTOMPATH_SUPPORT_MARK_RESTORE`.

```
Size (*EstimateDSMCustomScan) (CustomScanState *node,
                               ParallelContext *pcxt);
```

Оценивает объём динамической разделяемой памяти, которая потребуется для параллельной операции. Это значение может превышать объём, который будет занят фактически, но не должно быть меньше. Возвращаемое значение задаётся в байтах. Этот обработчик не является обязательным и должен устанавливаться, только если провайдер нестандартного сканирования поддерживает параллельное выполнение.

```
void (*InitializeDSMCustomScan) (CustomScanState *node,
```

```
                                       ParallelContext *pcxt,
                                       void *coordinate);
```

Инициализирует динамическую разделяемую память, которая потребуется для параллельной операции; `coordinate` указывает на область разделяемой памяти размера, равного возвращаемому значению `EstimateDSMCustomScan`. Этот обработчик является необязательным и должен устанавливаться, только если провайдер нестандартного сканирования поддерживает параллельное выполнение.

```
void (*ReInitializeDSMCustomScan) (CustomScanState *node,
                                   ParallelContext *pcxt,
                                   void *coordinate);
```

Заново инициализирует динамическую разделяемую память, требуемую для параллельной операции, перед тем как будет повторно просканирован узел нестандартного сканирования. Этот обработчик является необязательным и должен устанавливаться, только если провайдер нестандартного сканирования поддерживает параллельное выполнение. В этом обработчике рекомендуется сбрасывать только общее состояние, а в обработчике `ReScanCustomScan` сбрасывать только локальное. В настоящее время этот обработчик будет вызываться перед `ReScanCustomScan`, но лучше на этот порядок не рассчитывать.

```
void (*InitializeWorkerCustomScan) (CustomScanState *node,
                                    shm_toc *toc,
                                    void *coordinate);
```

Инициализирует локальное состояние параллельного исполнителя на основе общего состояния, заданного ведущим исполнителем во время `InitializeDSMCustomScan`. Этот обработчик является необязательным и должен устанавливаться, только если провайдер нестандартного сканирования поддерживает параллельное выполнение.

```
void (*ShutdownCustomScan) (CustomScanState *node);
```

Освобождает ресурсы, когда становится понятно, что этот узел больше не будет выполняться. Этот обработчик вызывается не во всех случаях; иногда может вызываться только `EndCustomScan`. Так как сегмент DSM, используемый параллельным запросом, освобождается сразу после вызова этого обработчика, провайдеры нестандартного сканирования, которым нужно выполнять некоторые действия до ликвидации сегмента DSM, должны реализовывать этот метод.

```
void (*ExplainCustomScan) (CustomScanState *node,
                           List *ancestors,
                           ExplainState *es);
```

Выводит дополнительную информацию для `EXPLAIN` об узле нестандартного сканирования. Этот обработчик является необязательным. Общие данные, сохранённые в `ScanState`, такие как целевой список и сканируемое отношение, будут выводиться и без этого обработчика, но с помощью этого обработчика можно выдать дополнительные, внутренние сведения.

Глава 58. Генетический оптимизатор запросов

Автор

Разработал Мартин Утеш (`<utesch@aut.tu-freiberg.de>`) для Института автоматического управления в Техническом университете Фрайбергская горная академия, Германия.

58.1. Обработка запроса как сложная задача оптимизации

Среди всех реляционных операторов самым сложным для обработки и оптимизации является *соединение*. В первую очередь потому, что по мере увеличения числа соединений в запросе число возможных планов запроса увеличивается экспоненциально. Дополнительная сложность оптимизации связана с наличием различных *методов соединения* (например, в Postgres Pro это вложенный цикл, соединение по хешу и соединение слиянием) для каждого отдельного соединения и разнообразием *индексов* (например, в Postgres Pro это B-дерево, хеш, GiST и GIN), определяющих путь доступа к отношениям.

Традиционный оптимизатор запросов Postgres Pro выполняет *почти исчерпывающий поиск* во всём множестве возможных стратегий. Этот алгоритм, появившийся в СУБД IBM System R, находит порядок соединений, близкий к оптимальному, но может требовать огромного количества времени и памяти, когда число соединений оказывается большим. В результате обычный оптимизатор Postgres Pro оказывается неподходящим для запросов, в которых соединяется большое количество таблиц.

Институт автоматического управления в Техническом университете Фрайбергская горная академия, Германия, столкнулся с этими проблемами, разрабатывая систему принятия решений на основе базы знаний для обслуживания электростанций, в которой в качестве СУБД планировалось применять PostgreSQL. Для машины, делающей выводы на основе базы знаний, СУБД должна была выполнять запросы с таким количеством соединений, что использование обычного оптимизатора запросов оказалось неприемлемым.

Далее мы опишем реализацию *генетического алгоритма*, который решает проблему выбора порядка соединений эффективным способом для запросов с большим числом соединений.

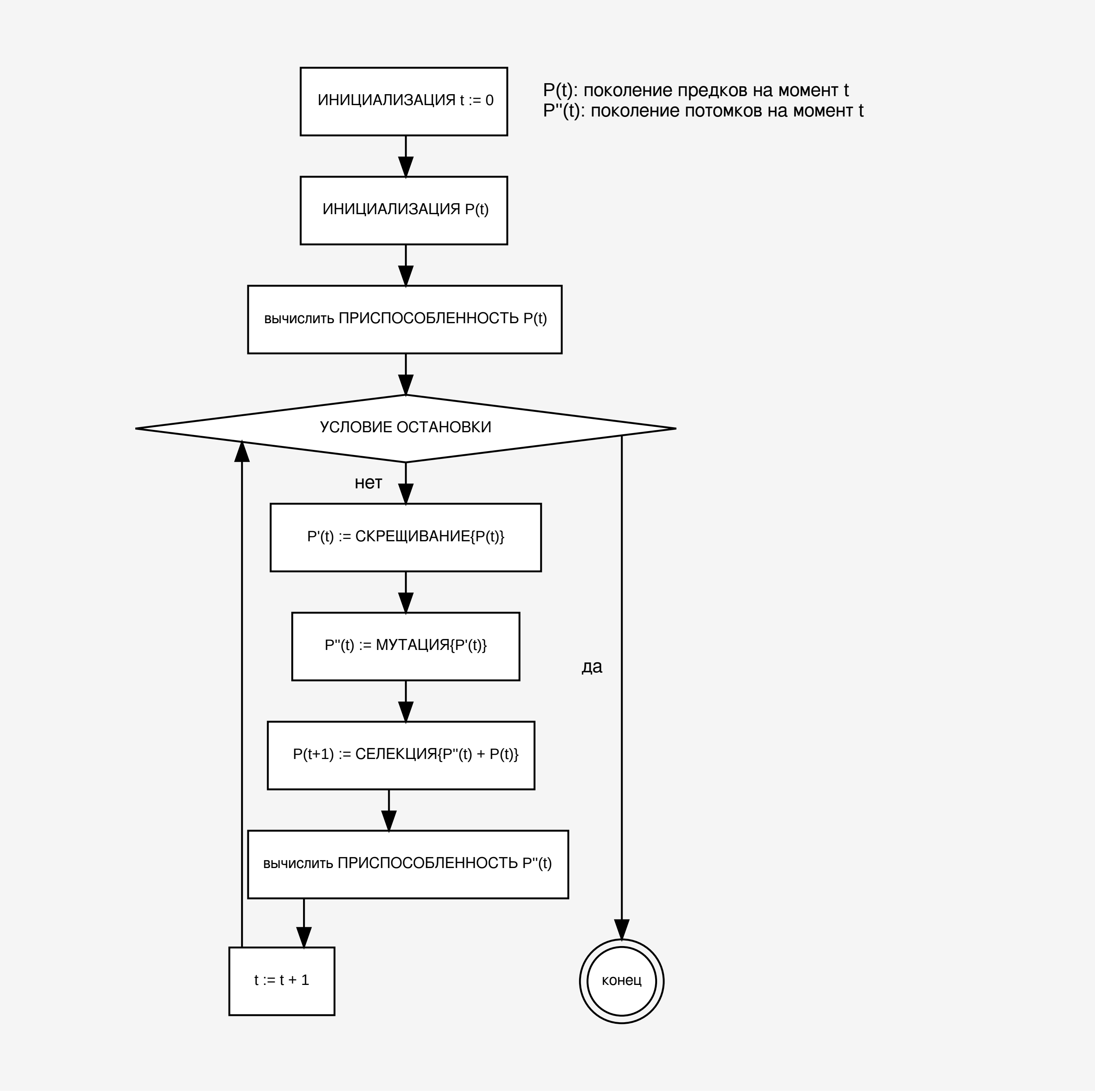
58.2. Генетические алгоритмы

Генетический алгоритм (ГА) реализует метод эвристической оптимизации, построенный на случайном поиске. В данном контексте множество возможных решений проблемы оптимизации называется *популяцией особей*. Степень адаптации особи к среде определяет функция *приспособленности*.

Координаты особи в пространстве поиска представляются *хромосомами*, которые по сути являются символьными строками. Фрагмент хромосомы, кодирующий значение одного оптимизируемого параметра, называется *геном*. Обычно ген кодируется в виде *двоичного* или *целочисленного* значения.

В результате симуляции эволюционных операций (*скрещивания*, *мутации* и *селекции*) данный алгоритм формирует новые поколения особей, у которых приспособленность в среднем будет выше, чем у их предшественников. Схема этих операций показана на Рисунке 58.1.

Рисунок 58.1. Структура генетического алгоритма



Как сказано в ответах на вопросы в группе comp.ai.genetic, нельзя не отметить, что ГА реализует не чисто случайный поиск решения проблемы. В ГА происходят вероятностные процессы, но результат явно оказывается не случайным (лучше случайного).

58.3. Генетическая оптимизация запросов (GEQO) в Postgres Pro

Модуль GEQO (Genetic Query Optimization, Генетическая оптимизация запросов) подходит к проблеме оптимизации запроса как к хорошо известной задаче коммивояжёра (TSP, Traveling Salesman Problem). Возможные планы запроса кодируются числами в строковом виде. Каждая строка представляет порядок соединения одного отношения из запроса со следующим. Например, дерево соединения

```
   /\
  /\ 2
 /\ 3
4  1
```

кодируется строкой целых чисел '4-1-3-2', которая означает: сначала соединить отношения '4' и '1', потом добавить '3', а затем '2', где 1, 2, 3, 4 — идентификаторы отношений внутри оптимизатора Postgres Pro.

Реализация GEQO в Postgres Pro имеет следующие особые характеристики:

- Использование ГА с *зафиксированным состоянием* (когда заменяются наименее приспособленные особи популяции, а не всё поколение) способствует быстрой сходимости к улучшенным планам запроса. Это важно для обработки запроса за приемлемое время;
- Использование *скрещивания с обменом рёбер*, которое очень удачно минимизирует число потерянных рёбер при решении задачи коммивояжёра с применением ГА;
- Мутация как генетический оператор считается устаревшей, так что для получения допустимых путей TSP не требуются механизмы исправления.

Части модуля GEQO взяты из алгоритма Genitor, разработанного Д. Уитли.

В результате, модуль GEQO позволяет оптимизатору запросов Postgres Pro эффективно выполнять запросы со множеством соединений, обходясь без полного перебора вариантов.

58.3.1. Построение возможных планов с GEQO

В процедуре планирования в GEQO используется код стандартного планировщика, который строит планы сканирования отдельных отношений. Затем вырабатываются планы соединений с применением генетического подхода. Как было сказано выше, каждый план соединения представляется последовательностью чисел, определяющей порядок соединений базовых отношений. На начальной стадии код GEQO просто случайным образом генерирует несколько возможных последовательностей. Затем для каждой рассматриваемой последовательности вызывается функция стандартного планировщика, оценивающая стоимость запроса в случае выбора этого порядка соединений. (Для каждого шага последовательности рассматриваются все три возможные стратегии соединения и все изначально выбранные планы сканирования отношений. Результирующей оценкой стоимости будет минимальная из всех возможных.) Последовательности соединений с наименьшей оценкой стоимости считаются «более приспособленными», чем последовательности с большей оценкой. Проанализировав возможные последовательности, генетический алгоритм отбрасывает наименее приспособленные из них. Затем генерируются новые кандидаты путём объединения генов более приспособленных последовательностей — для этого выбираются случайные фрагменты известных последовательностей с низкой стоимостью, из которых складываются новые последовательности для рассмотрения. Этот процесс повторяется, пока не будет рассмотрено некоторое предопределённое количество последовательностей соединений; после этого для построения окончательного плана выбирается лучшая последовательность, найденная за всё время поиска.

Этот процесс по природе своей недетерминирован, вследствие случайного выбора при формировании начальной популяции и последующей «мутации» лучших кандидатов. Но во избежание неожиданных изменений выбранного плана, на каждом проходе алгоритм GEQO перезапускает свой генератор случайных чисел с текущим значением параметра geqo_seed. Поэтому пока значение `geqo_seed` и другие параметры GEQO остаются неизменными, для определённого запроса (и других входных данных планировщика, в частности, статистики) будет строиться один и тот же план. Если вы хотите поэкспериментировать с разными путями соединений, попробуйте изменить `geqo_seed`.

58.3.2. Будущее развитие модуля PostgreSQL GEQO

Требуется провести дополнительную работу для выбора оптимальных параметров генетического алгоритма. Мы должны найти компромиссные значения параметров, удовлетворяющие двум несовместимым требованиям:

- Оптимальность плана запроса

- Время вычисления

В текущей реализации приспособленность каждой рассматриваемой последовательности соединений рассчитывается стандартным планировщиком, который каждый раз вычисляет избирательность соединения и стоимость заново. С учётом того, что различные кандидаты могут содержать общие подпоследовательности соединений, при этом будет повторяться большой объём работы. Таким образом, расчёт можно значительно ускорить, сохраняя оценки стоимости для внутренних соединений, но сложность состоит в том, чтобы уместить это состояние в разумные объёмы памяти.

На более общем уровне не вполне понятно, насколько уместно для оптимизации запросов использовать ГА, предназначенный для решения задачи коммивояжёра. В этой задаче стоимость, связанная с любой подстрокой (частью тура) не зависит от остального маршрута, но это определённо не так для оптимизации запросов. Таким образом, возникает вопрос, насколько эффективно скрещивание путём обмена рёбрами.

58.4. Дополнительные источники информации

Дополнительную информацию о генетических алгоритмах можно получить в следующих источниках:

- *The Hitch-Hiker's Guide to Evolutionary Computation*, (Руководство для путешествующих автостопом по эволюционным вычислениям, Ответы на часто задаваемые вопросы в группе *news://comp.ai.genetic*)
- *Evolutionary Computation and its application to art and design* (Эволюционные вычисления и их применение в искусстве и дизайне), Крейг Рейнольдс
- elma04
- fong

Глава 59. Определение интерфейса для табличных методов доступа

В этой главе описывается интерфейс между ядром системы Postgres Pro и *табличными методами доступа*, которые управляют хранением таблиц. Ядро системы не знает об этих методах доступа ничего, кроме того, что описано здесь; благодаря этому можно реализовывать абсолютно новые типы методов в рамках расширений.

Каждый табличный метод доступа описывается строкой в системном каталоге `pg_am`. В записи `pg_am` указывается имя и *функция-обработчик* для этого табличного метода. Эти записи могут создаваться и удаляться командами SQL CREATE ACCESS METHOD и DROP ACCESS METHOD.

Функция-обработчик табличного метода доступа должна объявляться как принимающая один аргумент типа `internal` и возвращающая псевдотип `table_am_handler`. Аргумент в данном случае фиктивный и нужен только для того, чтобы эту функцию нельзя было вызывать непосредственно из команд SQL. Возвращать эта функция должна указатель на структуру типа `TableAmRoutine`, содержащую всё, что нужно знать коду ядра, чтобы использовать этот метод доступа. Возвращаемое значение должно существовать всё время жизни сервера, что обычно достигается объявлением глобальной переменной `static const`. Структура `TableAmRoutine`, также называемая структурой API метода доступа, определяет поведение метода доступа через обработчики. Эти обработчики задаются как обычные указатели на функции уровня C, поэтому они не видны и не доступны на уровне SQL. Все обработчики и их свойства задаются в структуре `TableAmRoutine`, в определении которой можно найти комментарии с требованиями к ним. Для большинства обработчиков созданы функции-обёртки, документированные с точки зрения пользователя (а не разработчика) табличного метода доступа. Более подробно это описывается в файле *`src/include/access/tableam.h`*.

Чтобы реализовать метод доступа (МД), разработчик обычно должен создать для него специальный слот таблицы кортежей (см. *`src/include/executor/tuptable.h`*), позволяющий коду снаружи метода доступа иметь ссылки на кортежи данного МД и обращаться к столбцам кортежа.

В настоящее время способ хранения данных, определяемый МД, может быть практически любым. Например, МД может по своему усмотрению использовать кеш общих буферов. Если этот кеш используется, скорее всего имеет смысл применять и стандартную компоновку страницы, описанную в Разделе 63.6.

Одним довольно серьёзным ограничением API табличных методов доступа является то, что в настоящее время МД может поддерживать модификации данных и/или индексы, только если для каждого кортежа имеется идентификатор (TID), состоящий из номера блока и номера элемента (см. также Раздел 63.6). Компоненты TIDs в принципе могут иметь значение, отличное от принятого для метода `heap`, но если желательно поддерживать сканирование по битовой карте (вообще это не обязательно), номер блока должен обеспечивать локальность данных.

Для восстановления при сбое МД может использовать WAL сервера или собственную реализацию журнала. В случае использования WAL в МД можно задействовать Унифицированные записи WAL или реализовать Пользовательский журнал ресурсов WAL.

Чтобы реализовать поддержку транзакций способом, позволяющим обращаться к различным табличным методам в одной транзакции, скорее всего потребуется интегрировать её в механизм *`src/backend/access/transam/xlog.c`*.

Глава 60. Определение интерфейса для индексных методов доступа

В этой главе описывается интерфейс между ядром системы Postgres Pro и *индексными методами доступа*, которые управляют отдельными типами индексов. Ядро системы не знает об индексах ничего, кроме того, что описано здесь; благодаря этому можно реализовывать абсолютно новые типы индексов в рамках расширений.

Все индексы Postgres Pro являются, говоря на техническом уровне, *вторичными индексами*; то есть, они физически отделены от файла таблицы, к которой относятся. Каждый индекс хранится в собственном отдельном физическом *отношении* и описывается в отдельной записи в каталоге `pg_class`. Содержимое индекса находится полностью под контролем соответствующего метода доступа. На практике все индексные методы доступа делят индексы на страницы стандартного размера, чтобы для обращения к содержимому индекса можно было задействовать обычный менеджер хранилища и менеджер буферов. (Более того, большинство существующих методов доступа используют одну структуру страницы, описанную в Разделе 63.6, и одинаковый формат заголовков кортежей индекса; но эти решения методам доступа не навязываются.)

Индекс по сути представляет собой сопоставление некоторых значений ключей данных с *идентификаторами кортежей*, TID (Tuple Identifier), или версиями строк в основной таблице индекса. TID состоит из номера блока и номера записи в этом блоке (см. Раздел 63.6). Этой информации достаточно, чтобы выбрать определённую версию строки из таблицы. Индексы сами по себе не знают, что в модели MVCC у одной логической строки может быть несколько существующих версий; для индекса каждый кортеж — независимый объект, которому нужна своя запись в индексе. Таким образом, при изменении строки для неё всегда заново создаются новые записи индекса, даже если значения ключа не изменились. (Кортежи HOT представляют собой исключение из этого утверждения; но индексы всё равно не имеют с этим дела.) Записи индексов для мёртвых кортежей высвобождаются (при очистке), когда высвобождаются сами мёртвые кортежи.

60.1. Базовая структура API для индексов

Каждый индексный метод доступа описывается строкой в системном каталоге `pg_am`. В записи `pg_am` указывается имя и *функция-обработчик* для этого метода. Эти записи могут создаваться и удаляться командами SQL CREATE ACCESS METHOD и DROP ACCESS METHOD.

Функция-обработчик индексного метода доступа должна объявляться как принимающая один аргумент типа `internal` и возвращающая псевдотип `index_am_handler`. Аргумент в данном случае фиктивный, и нужен только для того, чтобы эту функцию нельзя было вызывать непосредственно из команд SQL. Возвращать эта функция должна структуру типа `IndexAmRoutine` (в памяти palloc), содержащую всё, что нужно знать коду ядра, чтобы использовать этот метод доступа. Структура `IndexAmRoutine`, также называемая *структурой API* метода доступа, содержит поля, задающие разнообразные предопределённые свойства метода доступа, например, поддерживает ли он составные индексы. Что более важно, она содержит указатели на опорные функции для метода доступа. Это обычные функции на C и они не видны и не могут быть вызваны на уровне SQL. Опорные функции описаны в Разделе 60.2.

Структура `IndexAmRoutine` определяется так:

```
typedef struct IndexAmRoutine
{
    NodeTag     type;

    /*
     * Общее число стратегий (операторов), с которыми возможен поиск/применение
     * этого метода доступа (МД).  Ноль, если у этого МД нет фиксированного набора
     * назначенных стратегий.
     */
    uint16      amstrategies;
```

```
    /* общее число опорных функций, используемых этим МД */
    uint16      amsupport;
    /* номер опорной функции options либо 0 */
    uint16      amoptsprocnum;
    /* поддерживает ли МД упорядочивание (ORDER BY) значений индексированного столбца?
 */
    bool        amcanorder;
    /* поддерживает ли МД упорядочивание (ORDER BY) результата оператора с
 индексированным столбцом? */
    bool        amcanorderbyop;
    /* поддерживает ли МД сканирование в обратном направлении? */
    bool        amcanbackward;
    /* поддерживает ли МД уникальные индексы (UNIQUE)? */
    bool        amcanunique;
    /* поддерживает ли МД индексы с несколькими столбцами? */
    bool        amcanmulticol;
    /* требуется ли для сканирования с МД ограничение первого столбца индекса? */
    bool        amoptionalkey;
    /* воспринимает ли МД условия ScalarArrayOpExpr? */
    bool        amsearcharray;
    /* воспринимает ли МД условия IS NULL/IS NOT NULL? */
    bool        amsearchnulls;
    /* может ли тип, хранящийся в индексе, отличаться от типа столбца? */
    bool        amstorage;
    /* возможна ли кластеризация по индексу этого типа? */
    bool        amclusterable;
    /* обрабатывает ли МД предикатные блокировки? */
    bool        ampredlocks;
    /* поддерживает ли МД параллельное сканирование? */
    bool        amcanparallel;
    /* поддерживает ли МД параллельное построение? */
    bool        amcanbuildparallel;
    /* поддерживает ли МД неключевые столбцы, добавляемые указанием INCLUDE? */
    bool        amcaninclude;
    /* использует ли МД maintenance_work_mem? */
    bool        amusemaintenanceworkmem;
    /* обобщает ли МД кортежи, чтобы все кортежи в блоке
     * попадали в одно обобщение? */
    bool        amsummarizing;
    /* ИЛИ флаги параллельной очистки */
    uint8       amparallelvacuumoptions;
    /* тип данных, хранящихся в индексе, либо InvalidOid, если он переменный */
    Oid         amkeytype;

    /* интерфейсные функции */
    ambuild_function ambuild;
    ambuildempty_function ambuildempty;
    aminsert_function aminsert;
    aminsertcleanup_function aminsertcleanup;   /*может быть NULL */
    ambulkdelete_function ambulkdelete;
    amvacuumcleanup_function amvacuumcleanup;
    amcanreturn_function amcanreturn;   /* может быть NULL */
    amcostestimate_function amcostestimate;
    amoptions_function amoptions;
    amproperty_function amproperty;     /* может быть NULL */
    ambuildphasename_function ambuildphasename;   /* может быть NULL */
    amvalidate_function amvalidate;
    amadjustmembers_function amadjustmembers; /* может быть NULL */
```

```
    ambeginscan_function ambeginscan;
    amrescan_function amrescan;
    amgettuple_function amgettuple;     /* может быть NULL */
    amgetbitmap_function amgetbitmap;   /* может быть NULL */
    amendscan_function amendscan;
    ammarkpos_function ammarkpos;       /* может быть NULL */
    amrestrpos_function amrestrpos;     /* может быть NULL */

    /* интерфейсные функции для поддержки параллельного сканирования по индексу */
    amestimateparallelscan_function amestimateparallelscan;    /* может быть NULL */
    aminitparallelscan_function aminitparallelscan;            /* может быть NULL */
    amparallelrescan_function amparallelrescan;                /* может быть NULL */
} IndexAmRoutine;
```

Чтобы индексный метод доступа применялся, необходимо также определить *семейства операторов* и *классы операторов* в `pg_opfamily`, `pg_opclass`, `pg_amop` и `pg_amproc`. Эти записи позволяют планировщику понять, для каких видов условий запросов могут применяться индексы с данными методом доступа. Семейства и классы операторов описываются в Разделе 36.16; этот материал необходимо изучить, прежде чем читать данную главу.

Отдельный индекс определяется записью в `pg_class`, описывающей его как физическое отношение, и записью в `pg_index`, представляющей логическое содержание индекса — то есть, набор столбцов индекса и семантическое значение этих столбцов, установленное соответствующими классами операторов. Столбцами индекса (значениями ключа) могут быть либо простые столбцы нижележащей таблицы, либо выражения, вычисляемые по строкам таблицы. Для индексного метода доступа обычно не важно, откуда поступают значения ключа индекса (они всегда поступают в вычисленном виде), но очень важна информация о классе операторов в каталоге `pg_index`. Обе эти записи каталогов представлены в составе структуры данных `Relation`, которая передаётся всем функциям, реализующим операции с индексом.

С некоторыми полями флагов в `IndexAmRoutine` связаны неочевидные следствия. Требования индексов с `amcanunique` описаны в Разделе 60.5. Флаг `amcanmulticol` показывает, что метод доступа поддерживает составные индексы, а `amoptionalkey` обозначает, что метод позволяет выполнить сканирование при отсутствии индексируемого ограничивающего условия для первого столбца индекса. Когда `amcanmulticol` равен false, `amoptionalkey` по сути говорит, поддерживает ли метод доступа полное сканирование по индексу без ограничивающего условия. Методы доступа, поддерживающие индексы с несколькими ключевыми столбцами, *должны* поддерживать сканирования при отсутствии ограничений любых или всех столбцов после первого; однако они могут требовать присутствия какого-либо ограничения для первого столбца индекса, и это требование отмечается значением false флага `amoptionalkey`. В `amoptionalkey` для МД может устанавливаться false, например, когда этот метод доступа не индексирует значения. Так как большинство индексируемых операторов — строгие, и поэтому не могут вернуть true для операндов NULL, на первый взгляд кажется заманчивой идея не хранить записи индекса для значений NULL: они всё равно никак не могут быть прочитаны при сканировании индекса. Однако этот аргумент отпадает, когда при сканировании индекса вовсе отсутствует ограничение данного столбца индекса. На практике это означает, что индексы с установленным флагом `amoptionalkey` должны индексировать значения NULL, так как планировщик может склониться к использованию этого индекса вообще без ключей. С этим связано ещё одно ограничение — индексный метод доступа, поддерживающий составные индексы, *должен* поддерживать индексирование значений NULL в столбцах после первого, так как планировщик будет полагать, что индекс можно применять для запросов, в которых эти столбцы не ограничиваются. Например, рассмотрим индекс по (a,b) и запрос с ограничением `WHERE a = 4`. Система будет полагать, что по этому индексу можно просканировать строки с `a = 4`, но это будет неверно, если индекс исключит строки, в которых `b` — NULL. Однако этот индекс вполне может исключить строки, в которых первый столбец содержит NULL. Метод индекса, который индексирует значения NULL, может также установить флаг `amsearchnulls`, отметив тем самым, что он поддерживает в качестве условий поиска `IS NULL` и `IS NOT NULL`.

Флаг `amcaninclude` показывает, поддерживает ли метод доступа «неключевые» столбцы, то есть может ли он сохранить (без обработки) дополнительные столбцы помимо ключевых. Требования в предыдущем абзаце распространяются только на ключевые столбцы. В частности, сочетание `amcanmulticol=false` и `amcaninclude=true` вполне осмысленно: оно означает, что в индексе может быть только один ключевой столбец и при этом несколько дополнительных неключевых столбцов. Кроме того, неключевые столбцы должны допускать значение null вне зависимости от флага `amoptionalkey`.

Флаг `amsummarizing` указывает, обобщает ли метод доступа индексированные кортежи на уровне детализации обобщения, по крайней мере, для каждого блока. Методы доступа, которые указывают не на отдельные кортежи, а на зоны блоков (как BRIN), могут позволить продолжать оптимизацию HOT. Это не относится к атрибутам, на которые ссылаются предикаты индекса: изменение такого атрибута всегда отключает HOT.

60.2. Функции для индексных методов доступа

Индексный метод доступа должен определить в `IndexAmRoutine` следующие функции построения и обслуживания индексов:

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

Строит новый индекс. Отношение индекса уже физически создано, но пока пусто. Оно должно быть наполнено фиксированными данными, которые требуются методу доступа, и записями для всех кортежей, уже существующих в таблице. Обычно функция `ambuild` вызывает `table_index_build_scan()` для поиска в таблице существующих кортежей и для вычисления ключей, которые должны вставляться в этот индекс. Эта функция должна возвращать структуру, выделенную вызовом palloc и содержащую статистику нового индекса. Флаг `amcanbuildparallel` указывает, поддерживает ли метод доступа параллельные построения индексов. Если установлено значение `true`, система попытается выделить параллельные рабочие процессы для построения. Для методов доступа, поддерживающих только непараллельные построения индексов, этот флаг должен иметь значение `false`.

```
void
ambuildempty (Relation indexRelation);
```

Создаёт пустой индекс и записывает его в слой инициализации (`INIT_FORKNUM`) данного отношения. Этот метод вызывается только для нежурналируемых индексов; пустой индекс, записанный в слой инициализации, будет копироваться в основной слой отношения при каждом перезапуске сервера.

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          IndexUniqueCheck checkUnique,
          bool indexUnchanged,
          IndexInfo *indexInfo);
```

Вставляет новый кортеж в существующий индекс. В массивах `values` и `isnull` передаются значения ключа, которые должны быть проиндексированы, а в `heap_tid` — идентификатор индексируемого кортежа (TID). Если метод доступа поддерживает уникальные индексы (флаг `amcanunique` установлен), параметр `checkUnique` указывает, какая проверка уникальности должна выполняться. Это зависит от того, является ли ограничение уникальности откладываемым; за подробностями обратитесь к Разделу 60.5. Обычно параметр `heapRelation` нужен методу доступа только для проверки уникальности (так как он должен обратиться к основным данным, чтобы убедиться в актуальности кортежа).

Булево значение `indexUnchanged` даёт «подсказку» о природе индексируемого кортежа. Когда это значение — true, кортеж является дубликатом некоторого существующего кортежа в индексе. Новый кортеж является логически неизменённым, созданным для новой версии MVCC. Такие кортежи появляются, когда выполняется операция `UPDATE`, которая не меняет никакие столбцы, охватываемые индексом, но тем не менее требует добавления новой версии кортежа в индекс. Опираясь на эту «подсказку», индексный метод может принять решение о выполнении восходящего удаления индексных кортежей в частях индекса, где скапливается много версий одной и той же логической строки. Заметьте, что изменение неключевого столбца не влияет на значение `indexUnchanged`. В коде ядра определяется значение `indexUnchanged` каждого кортежа при использовании подхода с низкими издержками, который допускает как ложные положительные, так и ложные отрицательные результаты. Индексные МД не должны рассматривать `indexUnchanged` как авторитетный источник информации о видимости кортежа или версионировании.

Возвращаемый функцией булев результат имеет значение, только когда параметр `checkUnique` равен `UNIQUE_CHECK_PARTIAL`. В этом случае результат true означает, что новая запись признана уникальной, тогда как false означает, что она может быть неуникальной (и требуется назначить отложенную проверку уникальности). В других случаях рекомендуется возвращать постоянный результат false.

Некоторые индексы могут индексировать не все кортежи. Если кортеж не будет индексирован, `aminsert` должна просто завершиться, не делая ничего.

Если индексный МД хочет кешировать данные между операциями добавления в индекс в одном операторе SQL, он может выделить память в `indexInfo->ii_Context` и сохранить указатель на эти данные в поле `indexInfo->ii_AmCache` (которое изначально равно NULL). Если после вставки индекса необходимо освободить какие-либо ресурсы, кроме памяти, можно использовать функцию `aminsertcleanup`, которая будет вызвана перед освобождением памяти.

```
void
aminsertcleanup (Relation indexRelation,
                 IndexInfo *indexInfo);
```

Очищает состояние, которое поддерживалось в течение последовательных вставок в `indexInfo->ii_AmCache`. Это полезно, если данные требуют дополнительных шагов очистки (например, освобождения закреплённых буферов), а простого освобождения памяти недостаточно.

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

Удаляет кортеж(и) из индекса. Это операция «массового удаления», которая предположительно будет реализована путём сканирования всего индекса и проверки для каждой записи, должна ли она удаляться. Переданная функция `callback` должна вызываться в стиле `callback(`*`TID`*`, callback_state)` с результатом bool, который говорит, должна ли удаляться запись индекса, на которую указывает передаваемый TID. Возвращать эта функция должна NULL или структуру, выделенную вызовом palloc и содержащую статистику результата удаления. NULL можно вернуть, если никакая информация не должна передаваться в `amvacuumcleanup`.

Из-за ограничения `maintenance_work_mem` процедура `ambulkdelete` может вызываться несколько раз, когда удалению подлежит большое количество кортежей. В аргументе `stats` передаётся результат предыдущего вызова для данного индекса (при первом вызове в ходе операции `VACUUM` он содержит NULL). Это позволяет методу доступа накапливать статистику в процессе всей операции. Обычно `ambulkdelete` модифицирует и возвращает одну и ту же структуру, если в `stats` передаётся не NULL.

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
```

```
                IndexBulkDeleteResult *stats);
```

Провести уборку после операции `VACUUM` (до этого `ambulkdelete` могла вызываться несколько или ноль раз). От этой функции не требуется ничего, кроме как выдать статистику по индексу, но она может произвести массовую уборку, например, высвободить пустые страницы индекса. В `stats` ей передаётся структура, возвращённая при последнем вызове `ambulkdelete`, либо NULL, если `ambulkdelete` не вызывалась, так как никакие кортежи удалять не требовалось. Эта функция должна возвращать NULL или структуру, выделенную вызовом palloc. Содержащаяся в этой структуре статистика будет отражена в записи в `pg_class` и попадёт в вывод команды `VACUUM`, если она выполнялась с указанием `VERBOSE`. NULL может возвращаться, если индекс вовсе не изменился в процессе операции `VACUUM`, но в противном случае должна возвращаться корректная статистика.

`amvacuumcleanup` также вызывается в конце операции `ANALYZE`. В этом случае `stats` всегда NULL и любое возвращаемое значение игнорируется. Этот вариант вызова можно распознать, проверив поле `info->analyze_only`. При таком вызове методу доступа рекомендуется ничего не делать, кроме как провести уборку после добавления данных, и только в рабочем процессе автоочистки.

```
bool
amcanreturn (Relation indexRelation, int attno);
```

Проверяет, поддерживается ли *сканирование только индекса* для заданного столбца, когда из индекса можно получить исходное значение столбца. Атрибуты нумеруются с 1, то есть для первого столбца attno равен 1. Возвращает true, если такое сканирование поддерживается, а иначе — false. Эта функция должна всегда возвращать true для неключевых столбцов (если таковые поддерживаются), так как неключевые столбцы, значения которые нельзя извлечь, не имеют смысла. Если индексный метод доступа в принципе не поддерживает сканирование только индекса, в поле `amcanreturn` его структуры `IndexAmRoutine` можно записать NULL.

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

Рассчитывает примерную стоимость сканирования индекса. Эта функция полностью описывается ниже в Разделе 60.6.

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

Разбирает и проверяет массив параметров для индекса. Эта функция вызывается, только когда для индекса задан отличный от NULL массив reloptions. Массив *`reloptions`* состоит из элементов типа `text`, содержащих записи вида *`имя=значение`*. Данная функция должна получить значение типа `bytea`, которое будет скопировано в поле `rd_options` записи индекса в relcache. Содержимое этого значения `bytea` определяется самим методом доступа; большинство стандартных методов доступа помещают в него структуру `StdRdOptions`. Когда параметр *`validate`* равен true, эта функция должна выдать подходящее сообщение об ошибке, если какие-либо параметры нераспознаны или имеют недопустимые значения; если же *`validate`* равен false, некорректные записи должны просто игнорироваться. (В *`validate`* передаётся false, когда параметры уже загружены в `pg_catalog`; при этом неверная запись может быть обнаружена, только если в методе доступа поменялись правила обработки параметров, и в этом случае стоит просто игнорировать такие записи.) NULL можно вернуть, когда нужно получить поведение по умолчанию.

```
bool
amproperty (Oid index_oid, int attno,
            IndexAMProperty prop, const char *propname,
```

```
             bool *res, bool *isnull);
```

Процедура `amproperty` позволяет индексным методам доступа переопределять стандартное поведение функции `pg_index_column_has_property` и связанных с ней. Если метод доступа не проявляет никаких особенностей при запросе свойств индексов, поле `amproperty` в структуре `IndexAmRoutine` может содержать NULL. В противном случае процедура `amproperty` будет вызываться с нулевыми параметрами *`index_oid`* и *`attno`* при вызове `pg_indexam_has_property`, либо с корректным *`index_oid`* и нулевым *`attno`* при вызове `pg_index_has_property`, либо с корректным *`index_oid`* и положительным *`attno`* при вызове `pg_index_column_has_property`. В *`prop`* передаётся значение перечисления, указывающее на проверяемое значение, а в *`propname`* — строка с именем свойства. Если код ядра не распознаёт имя свойства, в *`prop`* передаётся `AMPROP_UNKNOWN`. Методы доступа могут воспринимать нестандартные имена свойств, проверяя *`propname`* на совпадение (для согласованности с кодом ядра используйте для проверки `pg_strcasecmp`); для имён, известных коду ядра, лучше проверять *`prop`*. Если процедура `amproperty` возвращает `true`, это значит, что она установила результат проверки свойства: она должна задать в `*res` возвращаемое логическое значение или установить в `*isnull` значение `true`, чтобы возвратить NULL. (Перед вызовом обе упомянутые переменные инициализируются значением `false`.) Если `amproperty` возвращает `false`, код ядра переключается на обычную логику определения результата проверки свойства.

Методы доступа, поддерживающие операторы упорядочивания, должны реализовывать проверку свойства `AMPROP_DISTANCE_ORDERABLE`, так как код ядра не знает, как это сделать и возвращает NULL. Также может быть полезно реализовать проверку `AMPROP_RETURNABLE`, если это можно сделать проще, чем обращаясь к индексу и вызывая `amcanreturn` (что делает код ядра по умолчанию). Для всех остальных стандартных свойств поведение ядра по умолчанию можно считать удовлетворительным.

```
char *
ambuildphasename (int64 phasenum);
```

Возвращает текстовое название переданной фазы построения индекса. Номера фаз передаются в процессе построения индекса функции `pgstat_progress_update_param`. Названия фаз показываются в представлении `pg_stat_progress_create_index`.

```
bool
amvalidate (Oid opclassoid);
```

Проверяет записи в каталоге для заданного класса операторов, насколько это может сделать метод доступа. Например, это может включать проверку, все ли необходимые опорные функции реализованы. Функция `amvalidate` должна вернуть false, если класс операторов непригоден к использованию. Сообщения о проблеме следует выдать через `ereport`, как правило, на уровне `INFO`.

```
void
amadjustmembers (Oid opfamilyoid,
                 Oid opclassoid,
                 List *operators,
                 List *functions);
```

Проверяет предложенные новые операторы и функции-члены семейства операторов, насколько метод доступа позволяет это сделать, и задаёт виды их зависимостей, если подразумеваемые по умолчанию неудовлетворительны Эта функция вызывается во время выполнения команд `CREATE OPERATOR CLASS` и `ALTER OPERATOR FAMILY ADD`; в последнем случае значение *`opclassoid`* равно `InvalidOid`. В аргументах типа `List` передаются списки элементов структуры `OpFamilyMember`. Проверки, выполняемые данной функцией, обычно являются подмножеством проверок, выполняемых `amvalidate`, поскольку предполагается, что `amadjustmembers` не видит полный набор членов. Например, в этой функции будет разумным проверить сигнатуры опорной функции, но не проверять, предоставляются ли все необходимые опорные функции. О любых проблемах можно сообщить, выдав ошибку. Связанные с зависимостями поля структуры `OpFamilyMember` инициализируются кодом ядра — если выполняется `CREATE OPERATOR CLASS`, создаются жёсткие зависимости от класса операторов, а если выполняется `ALTER OPERATOR FAMILY ADD` — мягкие зависимости от семейства операторов. Функция `amadjustmembers` может скорректировать эти поля, если более уместно другое поведение. Например, GIN, GiST и SP-GiST всегда устанавливают для операторов-членов мягкую зависимость от семейства операторов, поскольку в этих типах индексов связь между оператором и классом оператора относительно слаба; поэтому есть смысл разрешить свободное добавление и удаление членов операторов. Для необязательных опорных функций обычно также устанавливаются мягкие зависимости, чтобы при необходимости их можно было удалить.

Цель индекса, конечно, в том, чтобы поддерживать поиск кортежей, соответствующих индексируемому условию `WHERE`, по *ограничению* или *ключу поиска*. Сканирование индекса описывается более полно ниже, в Разделе 60.3. Индексный метод доступа может поддерживать «простое» сканирование, сканирование по «битовой карте» или и то, и другое. Метод доступа должен или может реализовывать следующие функции, связанные со сканированием:

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             int norderbys);
```

Подготавливает метод к сканированию индекса. В параметрах `nkeys` и `norderbys` задаётся количество операторов условия и сортировки, которые будут задействованы при сканировании; это может быть полезно для выделения памяти. Заметьте, что фактические значения ключей сканирования в этот момент ещё не предоставляются. В результате функция должна выдать структуру, выделенную средствами palloc. В связи с особенностями реализации, метод доступа *должен* создать эту структуру, вызвав `RelationGetIndexScan()`. В большинстве случаев все действия `ambeginscan` сводятся только к выполнению этого вызова и, возможно, получению блокировок; всё самое интересное при запуске сканирования индекса происходит в `amrescan`.

```
void
amrescan (IndexScanDesc scan,
          ScanKey keys,
          int nkeys,
          ScanKey orderbys,
          int norderbys);
```

Запускает или перезапускает сканирование индекса, возможно, с новыми ключами сканирования. (Для перезапуска сканирования с ранее переданными ключами в `keys` и/или `orderbys` передаётся NULL.) Заметьте, что количество ключей или операторов сортировки не может превышать значения, поступившие в `ambeginscan`. На практике возможность перезапуска используется, когда в соединении со вложенным циклом выбирается новый внешний кортеж, так что требуется сравнение с новым ключом, но структура ключей сканирования не меняется.

```
bool
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

Выбирает следующий кортеж в ходе данного сканирования, с передвижением по индексу в заданном направлении (вперёд или назад). Возвращает true, если кортеж был получен, или false, если подходящих кортежей не осталось. В случае успеха в структуре `scan` сохраняется TID кортежа. Заметьте, что под «успехом» здесь подразумевается только, что индекс содержит запись, соответствующую ключам сканирования, а не то, что данный кортеж обязательно существует в данных или оказывается видимым в снимке вызывающего субъекта. При положительном результате `amgettuple` должна также установить для свойства `scan->xs_recheck` значение true или false. Значение false будет означать, что запись индекса точно соответствует ключам сканирования, а true — что есть сомнение в этом, так что условия, представленные ключами сканирования, необходимо ещё раз перепроверить для фактического кортежа, когда он будет получен. Это свойство введено для поддержки «неточных» операторов индексов. Заметьте, что такая перепроверка касается только условий сканирования; предикат частичного индекса (если он имеется) никогда не перепроверяется кодом, вызывающим `amgettuple`.

Если индекс поддерживает сканирование только индекса (то есть `amcanreturn` выдаёт true для каких-либо его столбцов), то в случае успеха метод доступа должен также проверить флаг `scan-`

`>xs_want_itup` и, если он установлен, должен вернуть исходные индексированные данные для этой записи индекса. В столбцах, для которых `amcanreturn` выдаёт false, можно вернуть null. Данные могут возвращаться посредством указателя на `IndexTuple`, сохранённого в `scan->xs_itup`, с дескриптором `scan->xs_itupdesc`; либо посредством указателя на `HeapTuple`, сохранённого в `scan->xs_hitup`, с дескриптором кортежа `scan->xs_hitupdesc`. (Второй вариант должен использоваться при восстановлении данных, которые могут не уместиться в `IndexTuple`.) В любом случае за управление целевой областью данных, определяемой этим указателем, отвечает метод доступа. Данные должны оставаться актуальными как минимум до следующего вызова `amgettuple`, `amrescan` или `amendscan` в процессе сканирования.

Функция `amgettuple` должна быть реализована, только если метод доступа поддерживает «простое» сканирование индекса. В противном случае поле `amgettuple` в структуре `IndexAmRoutine` должно содержать NULL.

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

Выбирает все кортежи для данного сканирования и добавляет их в передаваемую вызывающим кодом структуру `TIDBitmap` (то есть, получает логическое объединение множества TID выбранных кортежей с множеством, уже записанным в битовой карте). Возвращает эта функция число полученных кортежей (это может быть только приблизительная оценка; например, некоторые методы доступа не учитывают повторяющиеся значения). Добавляя идентификаторы кортежей в битовую карту, `amgetbitmap` может обозначить, что для этих кортежей нужно перепроверить условия сканирования. Для этого так же, как и в `amgettuple`, устанавливается выходной параметр `xs_recheck`. Замечание: в текущей реализации эта возможность увязывается с возможностью неточного хранения самих битовых карт, таким образом вызывающий код перепроверяет для отмеченных кортежей и условия сканирования, и предикат частичного индекса (если он имеется). Однако так может быть не всегда. Функции `amgetbitmap` и `amgettuple` не могут использоваться в одном сканировании индекса; есть и другие ограничения в применении `amgetbitmap`, описанные в Разделе 60.3.

Функция `amgetbitmap` должна быть реализована, только если метод доступа поддерживает сканирование индекса «по битовой карте». В противном случае поле `amgetbitmap` в структуре `IndexAmRoutine` должно содержать NULL.

```
void
amendscan (IndexScanDesc scan);
```

Завершает сканирование и освобождает ресурсы. Саму структуру `scan` освобождать не следует, но любые блокировки или закрепления объектов, установленные внутри метода доступа, должны быть сняты.

```
void
ammarkpos (IndexScanDesc scan);
```

Помечает текущую позицию сканирования. Метод доступа должен поддерживать сохранение только одной позиции в процессе сканирования.

Функция `ammarkpos` должна быть реализована, только если метод доступа поддерживает сканирование по порядку. Если это не так, в поле `ammarkpos` в структуре `IndexAmRoutine` можно записать NULL.

```
void
amrestrpos (IndexScanDesc scan);
```

Восстанавливает позицию сканирования, отмеченную последней.

Функция `amrestrpos` должна быть реализована, только если метод доступа поддерживает сканирование по порядку. Если это не так, в поле `amrestrpos` в структуре `IndexAmRoutine` можно записать NULL.

Помимо обычного сканирования некоторые типы индексов могут поддерживать *параллельное сканирование индекса*, что позволяет осуществлять совместное сканирование индекса нескольким обслуживающим процессам. Для этого метод доступа должен организовать работу так, чтобы каждый из взаимодействующих процессов возвращал подмножество кортежей, которое бы возвращалось при обычном, не параллельном сканировании, и таким образом, чтобы объединение этих подмножеств совпадало с множеством кортежей, возвращаемых при обычном сканировании. Более того, чтобы не требовалась глобальная сортировка кортежей, возвращаемых при параллельном сканировании, порядок кортежей в подмножествах, выдаваемых всеми взаимодействующими процессами, должен соответствовать запрошенному. Для поддержки параллельного сканирования по индексу должны быть реализованы следующие функции:

```
Size
amestimateparallelscan (int nkeys,
                        int norderbys);
```

Рассчитывает и возвращает объём (в байтах) в динамической разделяемой памяти, который может потребоваться для осуществления параллельного сканирования. (Этот объём дополняет, а не заменяет объём памяти, затребованный для данных, независимо от МД, в `ParallelIndexScanDescData`.)

Параметры `nkeys` и `norderbys` указывают количество условий и операторов упорядочивания, которые будут использоваться при сканировании, те же значения будут переданы функции `amrescan`. Обратите внимание, что фактические значения ключей сканирования ещё не предоставляются.

Эту функцию можно не реализовывать для методов доступа, которые не поддерживают параллельное сканирование, или для которых объём дополнительно требующейся памяти равен нулю.

```
void
aminitparallelscan (void *target);
```

Эта функция будет вызываться для инициализации области динамической разделяемой памяти в начале параллельного сканирования. Параметр *`target`* будет указывать на область объёма, не меньшего, чем возвратила функция `amestimateparallelscan`, и данная функция может хранить в этой области любые нужные ей данные.

Эту функцию можно не реализовывать для методов доступа, которые не поддерживают параллельное сканирование, или когда выделенная область в разделяемой памяти не требует инициализации.

```
void
amparallelrescan (IndexScanDesc scan);
```

Эта функция, если её реализовать, будет вызываться перед перезапуском параллельного сканирования индекса. Она должна сбросить всё разделяемое состояние, установленное функцией `aminitparallelscan`, с тем, чтобы такое сканирование перезапустилось с начала.

60.3. Сканирование индекса

В процессе сканирования индексный метод доступа отвечает только за выдачу идентификаторов всех кортежей, которые по его представлению соответствуют *ключам сканирования*. Метод доступа *не* участвует в самой процедуре выборки этих кортежей из основной таблицы и не определяет, удовлетворяют ли эти кортежи условиям видимости или другим ограничениям.

Ключом сканирования является внутреннее представление предложения `WHERE` в виде *`ключ_индекса оператор константа`*, где ключ индекса — один из столбцов индекса, а оператор — один из членов семейства операторов, связанного с типом данного столбца. При сканировании по индексу могут задаваться несколько или ноль ключей сканирования, результаты поиска которых должны неявно объединяться операцией AND — ожидается, что возвращаемые кортежи будут удовлетворять всем заданным условиям.

Метод доступа для конкретного запроса может сообщить, что индекс является *неточным* или, другими словами, требует перепроверки. Это подразумевает, что при сканировании индекса будут возвращены все записи, соответствующие ключу сканирования, плюс, возможно, дополнительные записи, которые ему не соответствуют. Внутренний механизм сканирования затем повторно применит условия индекса к кортежу данных, чтобы проверить, нужно ли его выбирать на самом деле. Если признак перепроверки не установлен, при сканировании индекса должны возвращаться только соответствующие ключам записи.

Заметьте, что именно метод доступа должен гарантировать, что корректно будут найдены все и только те записи, которые соответствуют всем переданным ключам сканирования. Также учтите, что ядро системы просто передаёт все предложения `WHERE` с подходящими ключами индекса и семействами операторов, не проводя семантический анализ на предмет их избыточности или противоречивости. Например, с условием `WHERE x > 4 AND x > 14`, где `x` — столбец с индексом-B-деревом, именно самой функции `amrescan` в методе B-дерева предоставляется возможность понять, что первый ключ сканирования избыточный и может быть отброшен. Объём предварительной обработки, которую нужно произвести для этого в `amrescan`, зависит от того, до какой степени метод доступа должен сводить ключи к «нормализованной» форме.

Некоторые методы доступа возвращают записи индекса в чётко определённом порядке, в отличие от других. Фактически есть два различных варианта реализации упорядоченного вывода некоторым методом доступа:

- Для методов доступа, которые всегда возвращают записи в порядке их естественной сортировки (как например, в B-дереве), устанавливается признак `amcanorder`. В настоящее время операторам проверки равенства и упорядочивания при этом должны назначаться номера соответствующих стратегий B-дерева.
- Для методов доступа, которые поддерживают операторы упорядочивания, устанавливается признак `amcanorderbyop`. Он показывает, что индекс может возвращать записи в порядке, определяемом предложением `ORDER BY` *`ключ_индекса оператор константа`*. Модификаторы для такого сканирования могут передаваться в `amrescan`, как описывалось ранее.

У функции `amgettuple` есть аргумент `direction`, который может принимать значение `ForwardScanDirection` (обычный вариант, сканирование вперёд) или `BackwardScanDirection` (сканирование назад). Если в первом вызове после `amrescan` указывается `BackwardScanDirection`, то множество соответствующих записей индекса сканируется от конца к началу, а не в обычном направлении от начала к концу. В этом случае `amgettuple` должна вернуть последний соответствующий кортеж индекса, а не первый как обычно. (Это распространяется только на методы доступа с установленным признаком `amcanorder`.) После первого вызова `amgettuple` должна быть готова продолжать сканирование в любом направлении от записи, выданной последней до этого. (Но если признак `amcanbackward` не установлен, при всех последующих вызовах должно сохраняться то же направление, что было в первом.)

Методы доступа, которые поддерживают упорядоченное сканирование, должны уметь «помечать» позицию сканирования и затем возвращаться к помеченной позиции (возможно, несколько раз к одной и той же позиции). Но запоминаться должна только одна позиция в ходе сканирования; последующий вызов `ammarkpos` переопределяет ранее сохранённую позицию. Метод доступа, не поддерживающий упорядоченное сканирование, не должен определять функции `ammarkpos` и `amrestrpos` в `IndexAmRoutine`; достаточно записать в эти указатели NULL.

И позиция сканирования, и отмеченная позиция (при наличии) должны поддерживаться в согласованном состоянии с учётом одновременных добавлений или удалений записей в индексе. Не будет ошибкой, если только что вставленная запись не будет выдана при сканировании, которое могло бы найти эту запись, если бы она существовала до его начала, либо если сканирование выдаст такую запись после перезапуска или возврата, даже если она не была выдана в первый раз. Подобным образом, параллельное удаление может отражаться, а может и не отражаться в результатах сканирования. Важно только, чтобы при таких операциях добавления или удаления не происходило потерь или дублирования записей, которые в этих операциях не участвовали.

Если индекс сохраняет исходные индексируемые значения данных (а не их искажённое представление), обычно полезно поддержать сканирование только индекса, при котором индекс возвращает фактические данные, а не только TID кортежа данных. Это позволит оптимизировать ввод-вывод, только если карта видимости показывает, что TID относится к полностью видимой странице; в противном случае всё равно придётся посетить кортеж, чтобы проверить его видимость для MVCC. Но это не является заботой метода доступа.

Вместо `amgettuple`, сканирование индекса может осуществляться функцией `amgetbitmap`, которая выбирает все кортежи за один вызов. Это может быть значительно эффективнее `amgettuple`, так как позволяет избежать циклов блокировки/разблокировки в методе доступа. В принципе, `amgetbitmap` должна давать тот же эффект, что и многократные вызовы `amgettuple`, но простоты ради мы накладываем ряд дополнительных ограничений. Во-первых, `amgetbitmap` возвращает все кортежи сразу и не поддерживает пометку позиций и возвращение к ним. Во-вторых, кортежи, возвращаемые в битовой карте, не упорядочиваются каким-либо определённым образом, поэтому `amgetbitmap` не принимает аргумент `direction`. (И операторы упорядочивания никогда не будут передаваться для такого сканирования.) Кроме того, сканирование только индекса с `amgetbitmap` неосуществимо, так как нет никакой возможности возвратить содержимое кортежей индекса. Наконец, `amgetbitmap` не гарантирует, что будут установлены какие-либо блокировки для возвращаемых кортежей, и следствия этого описаны в Разделе 60.4.

Заметьте, что метод доступа может реализовывать только функцию `amgetbitmap`, но не `amgettuple`, и наоборот, если его внутренняя реализация несовместима с одной из этих функций.

60.4. Замечания о блокировке с индексами

Индексные методы доступа должны справляться с параллельными операциями обновления индекса, производимыми несколькими процессами. Ядро системы Postgres Pro получает блокировку `AccessShareLock` для индекса в процессе сканирования и `RowExclusiveLock` при модификации индекса (включая и обычную очистку командой `VACUUM`). Так как эти типы блокировок не конфликтуют, метод доступа должен сам устанавливать более точечные блокировки, которые ему могут потребоваться. Блокировка индекса в целом в режиме `ACCESS EXCLUSIVE` устанавливается только при создании и уничтожении индекса или операции `REINDEX` (вместо этого используется `SHARE UPDATE EXCLUSIVE` вместе с `CONCURRENTLY`).

Реализация типа индекса, поддерживающего параллельные изменения, обычно требует глубокого и всестороннего анализа требуемого поведения.

Помимо собственных внутренних требований индексов к целостности, при параллельном обновлении данных возникают вопросы согласованности родительской таблицы (*основных данных*) и индекса. Вследствие того, что Postgres Pro отделяет чтение и изменение основных данных от чтения и изменения индекса, образуются временные интервалы, в которых индекс может быть несогласованным с данными. Мы решаем эту проблему, применяя следующие правила:

- Новая запись в области данных добавляется до того, как для неё будут созданы записи в индексах. (Таким образом, при параллельном сканировании индекса эта запись в данных скорее всего не будет замечена. Это не проблема, так как читателю индекса всё равно не нужны незафиксированные строки. Но учтите написанное в Разделе 60.5.)
- Когда запись данных удаляется (командой `VACUUM`), сначала должны удалиться все созданные для неё записи в индексах.
- Сканирование индекса должно закрепить страницу индекса, на которой находится элемент, возвращённый последним вызовом `amgettuple`, а `ambulkdelete` не должна удалять записи со страниц, закреплённых другими процессами. Чем обосновано это правило, описывается ниже.

Без третьего правила читатель индекса мог бы увидеть запись индекса за мгновение до того, как она была удалена процедурой `VACUUM`, а затем обратиться к соответствующей записи данных после того, как `VACUUM` удалит и её. Это не приведёт к серьёзным проблемам, если данный элемент остаётся незадействованным, когда к нему обращается читатель, так как пустой слот будет игнорироваться функцией `heap_fetch()`. Но как быть, если третий процесс уже занял этот слот какими-то своими данными? Когда применяется снимок, совместимый с MVCC, и это не проблема, так как эти данные определённо окажутся слишком новыми при проверке видимости для данного снимка.

Однако для снимка несовместимого с MVCC (например, снимка `SnapshotAny`), может так получиться, что будет возвращена строка, на самом деле не соответствующая ключам сканирования. Мы можем защититься от такого исхода, потребовав, чтобы ключи сканирования всегда перепроверялись для строки данных, но это слишком дорогостоящее решение. Вместо этого, мы закрепляем страницу индекса как промежуточный объект, показывающий, что читатель может всё ещё быть «в пути» от записи индекса к соответствующей строке данных. Благодаря тому, что `ambulkdelete` блокируется при обращении к этой закреплённой странице, процедура `VACUUM` не сможет удалить строку данных, пока её извлечение не закончит читатель. Это решение оказывается очень недорогим по времени выполнения, а издержки блокирования привносятся только в редких случаях, когда действительно возникает конфликт.

Такое решение требует, чтобы сканирования индексов выполнялись «синхронно»: мы должны выбирать каждый следующий кортеж данных сразу после того получили соответствущую запись индекса. Это оказывается невыгодно по ряду причин. «Асинхронное» сканирование, при котором мы собираем множество TID из индекса, и обращаемся за кортежами данных только после этого, влечёт гораздо меньше издержек с блокировками и позволяет обращаться к данным более эффективным образом. Согласно проведённому выше анализу, мы должны использовать синхронный подход для снимков, несовместимых с MVCC, но для запросов со снимками MVCC будет работать и асинхронное сканирование.

При сканировании индекса с `amgetbitmap`, метод доступа не закрепляет страницы индекса ни для каких из возвращаемых кортежей. Поэтому такое сканирование можно безопасно применять только со снимками MVCC.

Когда флаг `ampredlocks` не установлен, любое сканирование с данным методом доступа в сериализуемой транзакции будет получать неблокирующую предикатную блокировку для всего индекса. Это будет приводить к конфликту чтения-записи при добавлении любого кортежа в этот индекс параллельной сериализуемой транзакцией. Если среди набора параллельных сериализуемых транзакций выявляются определённые варианты конфликтов чтения-записи, одна из этих транзакций может быть отменена для сохранения целостности данных. Когда данный флаг установлен, это означает, что метод доступа реализует более точную предикатную блокировку, что способствует сокращению частоты отмены транзакций по этой причине.

60.5. Проверки уникальности в индексе

Postgres Pro реализует ограничения уникальности SQL, применяя *уникальные индексы*, то есть такие индексы, которые не принимают несколько записей с одинаковыми ключами. Для метода доступа, поддерживающего это свойство, устанавливается признак `amcanunique`. (В настоящее время это поддерживают только B-деревья.) Столбцы, указанные в предложении `INCLUDE`, не учитываются при контроле уникальности.

Вследствие особенностей MVCC, всегда необходимо допускать физическое сосуществование в индексе дублирующихся записей: такие записи могут относиться к последовательным версиям одной логической строки. На самом деле мы хотим добиться только того, чтобы никакой снимок MVCC не мог содержать две строки с одинаковыми ключами индекса. Из этого вытекают следующие ситуации, которые необходимо отследить, добавляя новую строку в уникальный индекс:

- Если конфликтующая строка была удалена текущей транзакцией, это не проблема. (В частности из-за того, что UPDATE всегда удаляет старую версию строки, прежде чем вставлять новую, операцию UPDATE можно выполнять со строкой, не меняя её ключ.)
- Если конфликтующая строка была добавлена ещё не зафиксированной транзакцией, запрос, претендующий на добавление новой строки, должен подождать, пока эта транзакция не будет зафиксирована. Если она откатывается, конфликт исчезает. Если она фиксируется и при этом оставляет конфликтующую строку, возникает нарушение уникальности. (На практике мы просто ждём завершения другой транзакции и затем пересматриваем проверку видимости.)
- Подобным образом, если конфликтующая строка была удалена ещё не зафиксированной транзакцией, запрос, претендующий на добавление новой строки, должен дождаться фиксации или отката этой транзакции, а затем повторить проверку.

Более того, непосредственно перед тем как сообщать о нарушении уникальности согласно вышеприведённым правилам, метод доступа должен перепроверить, продолжает ли существовать добавляемая строка. Если она признана «мёртвой», о предвиденном нарушении он сообщать не должен. (Такого не должно быть при обычном сценарии добавления строки текущей транзакцией, однако это может произойти в процессе `CREATE UNIQUE INDEX CONCURRENTLY`.)

Мы требуем, чтобы метод доступа выполнял эти проверки сам, и это означает, что он должен обратиться к основным данным и проверить состояние фиксации всех строк, которые согласно содержимого индекса содержат дублирующиеся ключи. Это без сомнения некрасивый и немодульный подход, но он избавляет от излишней работы: если бы мы делали отдельную пробу, то поиск конфликтующей строки по индексу пришлось бы по сути повторять, пытаясь найти место, куда вставить запись для новой строки. Более того, не представляется возможным избежать условий гонки, если проверка конфликта не будет неотъемлемой частью процедуры добавления новой записи индекса.

Если ограничение уникальности откладываемое, возникает дополнительная сложность: нам нужна возможность добавлять запись индекса для новой строки, но отложить выводы о нарушении уникальности до конца оператора или даже позже. Чтобы избежать ненужного повторного поиска по индексу, метод доступа должен произвести предварительную проверку уникальности во время изначального добавления строк. Если при этом окажется, что никакие кортежи не конфликтуют, на этом проверка заканчивается. В противном случае мы планируем перепроверку на время, когда это ограничение начинает действовать. Если во время перепроверки продолжают существовать и вставленный кортеж, и какой-либо другой с тем же ключом, должна выдаваться ошибка. (Заметьте, что в данном случае под «существованием» понимается «существование любого кортежа в цепочке HOT записей индекса».) Для реализации этой схемы в `aminsert` передаётся параметр `checkUnique`, принимающий одно из следующих значений:

- `UNIQUE_CHECK_NO` указывает, что проверка уникальности не должна выполняться (это не уникальный индекс).
- `UNIQUE_CHECK_YES` указывает, что это неоткладываемый уникальный индекс и проверку уникальности нужно выполнить немедленно, как описано выше.
- `UNIQUE_CHECK_PARTIAL` указывает, что это откладываемое ограничение уникальности. Postgres Pro выбирает этот режим для добавления записи индекса для каждой строки. Метод доступа должен допускать добавление в индекс дублирующихся записей и сообщать о возможных конфликтах, возвращая false из `aminsert`. Для каждой такой строки (для которой возвращается false) будет запланирована отложенная перепроверка.

  Метод доступа должен отметить все строки, которые могут нарушать ограничение уникальности, но не будет ошибкой, если он допустит ложное срабатывание. Это позволяет произвести проверку, не дожидаясь завершения других транзакций; конфликты, выявленные на этой стадии, не считаются ошибками и будут перепроверены позже, когда они могут быть уже исчерпаны.
- `UNIQUE_CHECK_EXISTING` указывает, что это отложенная перепроверка строки, которая была отмечена как возможно нарушающая ограничение. Хотя для этой проверки вызывается `aminsert`, метод доступа *не* должен добавлять новую запись индекса в данном случае, так как эта запись уже существует. Вместо этого, метод доступа должен проверить, нет ли в индексе другой такой же записи. Если она находится и соответствующая ей строка продолжает существовать, должна выдаваться ошибка.

  Для варианта `UNIQUE_CHECK_EXISTING` в методе доступа рекомендуется дополнительно проверить, что для целевой строки действительно имеется запись в индексе и сообщить об ошибке, если это не так. Это хорошая идея, так как значения кортежа индекса, передаваемые в `aminsert`, будут рассчитаны заново. Если в определении индекса задействованы функции, которые на самом деле не постоянные, мы можем проверять неправильную область индекса. Дополнительно убедившись в существовании целевой строки при перепроверке, мы можем быть уверены, что сканируются те же значения кортежа, что передавались при изначальном добавлении строки.

60.6. Функции оценки стоимости индекса

Функции `amcostestimate` даётся информация, описывающая возможное сканирование индекса, включая списки предложений WHERE и ORDER BY, которые были выбраны как применимые с данным индексом. Она должна вернуть оценки стоимости обращения к индексу и избирательность предложений WHERE (то есть, процент строк основной таблицы, который будет получен в ходе сканирования индекса). Для простых случаев почти всю работу оценщика стоимости можно произвести, вызывая стандартные процедуры оптимизатора; смысл существования функции `amcostestimate` в том, чтобы индексные методы доступа могли поделиться знаниями, специфичными для типа индекса, когда это может помочь улучшить стандартные оценки.

Каждая функция `amcostestimate` должна иметь такую сигнатуру:

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

Первые три параметра передают входные значения:

`root`

Информация планировщика о выполняемом запросе.

`path`

Рассматриваемый путь доступа к индексу. В нём действительны все поля, кроме значений стоимости и избирательности.

`loop_count`

Число повторений сканирования индекса, которое должно приниматься во внимание при оценке стоимости. Обычно оно будет больше одного, когда при соединении со вложенным циклом планируется параметризованное сканирование. Заметьте, что оценки стоимости тем не менее должны рассчитываться для всего одного сканирования; большие значения *`loop_count`* лишь дают основания предположить, что при многократном сканировании положительное влияние может оказать кеширование.

Последние пять параметров — указатели на переменные для выходных значений:

`*indexStartupCost`

Стоимость выполнения запуска индекса

`*indexTotalCost`

Общая стоимость использования индекса

`*indexSelectivity`

Избирательность индекса

`*indexCorrelation`

Коэффициент корреляции между порядком сканирования индекса и порядком записей в нижележащей таблице

`*indexPages`

Количество страниц индекса на уровне листьев

Заметьте, что функции оценки стоимости должны разрабатываться на C, а не на SQL или другом доступном процедурном языке, так как они должны обращаться к внутренним структурам данным планировщика/оптимизатора.

Стоимости обращения к индексу следует вычислять с использованием параметров: последовательная выборка дискового блока имеет стоимость `seq_page_cost`, непоследовательная выборка — `random_page_cost`, а стоимость обработки одной строки индекса обычно принимается равной `cpu_index_tuple_cost`. Кроме того, за каждый оператор сравнения, вызываемый при обработке индекса, должна взиматься цена `cpu_operator_cost` (особенно за вычисление собственно условий индекса).

«Стоимость запуска» составляет часть общей стоимости сканирования, которая должна быть потрачена, прежде чем можно будет начать чтение первой строки. Для большинства индексов она может считаться нулевой, но для типов индексов с высокими затратами на запуск она может быть больше нуля.

Значение в *`indexSelectivity`* должно показывать, какой процент строк основной таблицы ожидается получить при сканировании таблицы. В случае неточного запроса это обычно будет больше процента строк, действительно удовлетворяющих заданным ограничивающим условиям.

В *`indexCorrelation`* записывается корреляция (в диапазоне от -1.0 до 1.0) между порядком записей в индексе и в таблице. Это значение будет корректировать оценку стоимости выборки строк из основной таблицы.

В *`indexPages`* записывается число страниц на уровне листьев. Это помогает выбрать число исполнителей для параллельного сканирования индекса.

Когда *`loop_count`* больше нуля, возвращаться должны средние значения, ожидаемые для одного сканирования индекса.

Оценка стоимости

Типичная процедура оценки выглядит следующим образом:

1. Рассчитать и вернуть процент строк родительской таблицы, которые будут посещены при заданных ограничивающих условиях. В отсутствие каких-либо знаний, специфичных для типа индекса, использовать стандартную функцию оптимизатора `clauselist_selectivity()`:

   ```
   *indexSelectivity = clauselist_selectivity(root, path->indexquals,
                                              path->indexinfo->rel->relid,
                                              JOIN_INNER, NULL);
   ```

2. Оценить число строк индекса, которые будут посещены при сканировании. Для многих типов индексов это будет произведение *`indexSelectivity`* и числа строк в индексе, но оно может быть и больше. (Заметьте, что размер индекса в страницах и строках можно узнать из структуры `path->indexinfo`.)

3. Рассчитать число страниц индекса, которые будут получены при сканировании. Это может быть просто произведение *`indexSelectivity`* и размера индекса в страницах.

4. Вычислить стоимость обращения к индексу. Универсальный оценщик может сделать следующее:

   ```
   /*
    * Вообще предполагается, что страницы индекса будут считываться последовательно,
    * так что стоимость их чтения cost seq_page_cost, а не random_page_cost.
    * Также мы добавляем стоимость за вычисление условия индекса для каждой строки.
    * Все стоимости считаются пропорционально возрастающими при сканировании.
    */
   cost_qual_eval(&index_qual_cost, path->indexquals, root);
   *indexStartupCost = index_qual_cost.startup;
   *indexTotalCost = seq_page_cost * numIndexPages +
   ```

```
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

Однако при таком расчёте не учитывается амортизация чтения индекса при повторном сканировании.

5. Оценить корреляцию индекса. Для простого упорядоченного индекса по одному полю её можно получить из pg_statistic. Если корреляция неизвестна, вернуть консервативную оценку — ноль (корреляция отсутствует).

Глава 61. Предварительная запись в журнал для расширений

Для некоторых расширений, особенно реализующих пользовательские методы доступа, может потребоваться предварительная запись в журнал для обеспечения безопасности данных при сбое. В Postgres Pro можно достичь этой цели двумя способами.

Первый способ: расширения могут использовать унифицированный тип записей WAL, в котором изменения страниц описываются в общем виде. Этот способ прост в реализации и не требует загрузки библиотеки расширений для применения записей. Однако записи WAL такого типа будут игнорироваться при выполнении логического декодирования.

Второй способ: расширения могут использовать пользовательский менеджер ресурсов. Этот способ более гибкий, поддерживает логическое декодирование и иногда генерирует записи WAL гораздо меньшего размера, чем при использовании унифицированных записей WAL. Однако его сложнее реализовать для расширений.

61.1. Унифицированные записи WAL

Хотя у всех внутренних модулей, взаимодействующих с WAL, имеются собственные типы записей WAL, существует также унифицированный тип записей WAL, описывающий изменения в страницах унифицированным образом.

> **Примечание**
>
> Унифицированные записи WAL игнорируются во время логического декодирования. Если для вашего расширения требуется логическое декодирование, рассмотрите возможность использования пользовательского менеджера ресурсов WAL.

API для конструирования унифицированных записей WAL определён в `access/generic_xlog.h` и реализован в `access/transam/generic_xlog.c`.

Чтобы сформировать запись изменения данных для WAL, применяя механизм унифицированных записей WAL, выполните следующие действия:

1. `state = GenericXLogStart(relation)` — начните формирование унифицированной записи WAL для заданного отношения.
2. `page = GenericXLogRegisterBuffer(state, buffer, flags)` — зарегистрируйте буфер, который будет изменён текущей унифицированной записью WAL. Эта функция возвращает указатель на временную копию страницы буфера, в которой должны производиться изменения. (Модифицировать непосредственно содержимое буфера нельзя.) В третьем аргументе передаётся битовая маска флагов, применимых к этой операции. В настоящее время флаг только один — `GENERIC_XLOG_FULL_IMAGE`, который показывает, что в запись WAL нужно включить образ всей страницы, а не только изменения. Обычно этот флаг должен устанавливаться, когда страница новая или полностью перезаписана. Вызов `GenericXLogRegisterBuffer` можно повторять, если фиксируемое в WAL действие изменяет несколько страниц.
3. Примените изменения к образам страниц, полученным на предыдущем шаге.
4. `GenericXLogFinish(state)` — завершите изменения в буферах и выдайте унифицированную запись WAL.

Формирование записи WAL можно прервать на любом шаге, вызвав `GenericXLogAbort(state)`. При этом будут отменены все изменения, внесённые в копии образов страниц.

Используя механизм унифицированных записей WAL, необходимо учитывать следующее:

- Модифицировать буферы напрямую нельзя! Все изменения должны производиться в копиях, полученных от функции `GenericXLogRegisterBuffer()`. Другими словами, код, формирующий унифицированные записи WAL, никогда не должен сам вызывать `BufferGetPage()`. Однако вызывающий код отвечает за закрепление/открепление и блокировку/разблокировку буферов в подходящие моменты времени. Исключительная блокировка каждого целевого буфера должна удерживаться от вызова `GenericXLogRegisterBuffer()` до `GenericXLogFinish()`.
- Регистрацию буферов (шаг 2) и модификацию образов страниц (шаг 3) можно свободно смешивать, оба этих шага можно повторять в любой последовательности. Но помните, что буферы должны регистрироваться в том же порядке, в каком для них должны получаться блокировки при воспроизведении.
- Максимальное число буферов, которые можно зарегистрировать для унифицированной записи WAL, составляет `MAX_GENERIC_XLOG_PAGES`. При исчерпании этого лимита будет выдана ошибка.
- Унифицированный тип WAL подразумевает, что страницы, подлежащие изменению, имеют стандартную структуру, в частности между `pd_lower` и `pd_upper` нет полезных данных.
- Так как изменяются копии страниц буфера, `GenericXLogStart()` не начинает критическую секцию. Таким образом вы можете безопасно выделять память, выдать ошибку и т. п. между `GenericXLogStart()` и `GenericXLogFinish()`. Единственная фактическая критическая секция присутствует внутри `GenericXLogFinish()`. При выходе по ошибке так же не нужно заботиться о вызове `GenericXLogAbort()`.
- `GenericXLogFinish()` помечает буферы как грязные и устанавливает для них LSN. Вам делать явно это не нужно.
- Для нежурналируемых отношений всё работает так же, за исключением того, что фактически запись в WAL не выдаётся. Таким образом, явно проверять, является ли отношение нежурналируемым, не требуется.
- Функция воспроизведения унифицированных изменений WAL получит исключительные блокировки буферов в том же порядке, в каком они были зарегистрированы. После воспроизведения всех изменений блокировки в том же порядке и освобождаются.
- Если для регистрируемого буфера не задаётся `GENERIC_XLOG_FULL_IMAGE`, унифицированная запись WAL содержит различие между старым и новым образом страницы, которое вычисляется при побайтовом сравнении. Результат оказывается не очень компактным при перемещении данных в странице, но это может быть доработано в будущем.

61.2. Пользовательские менеджеры ресурсов WAL

В этом разделе описывается интерфейс взаимодействия ядра Postgres Pro с пользовательскими менеджерами ресурсов WAL, используя который расширения могут непосредственно интегрировать свои объекты в WAL.

Расширениям, особенно реализующим табличные методы доступа или индексные методы доступа, может потребоваться использовать WAL для восстановления, репликации и/или логического декодирования.

Чтобы создать новый пользовательский менеджер ресурсов WAL, сначала определите структуру `RmgrData` с реализациями методов менеджера ресурсов.

```
/*
 * Таблица методов для менеджеров ресурсов.
 *
 * Эта структура должна синхронизироваться с определением PG_RMGR в
 * файле rmgr.c.
 *
 * Метод rm_identify должен выдать обозначение записи для переданного xl_info
 * (без ссылки на rmid). Например, для XLOG_BTREE_VACUUM он мог бы выдать
 * "VACUUM". Затем может быть вызван метод rm_desc, который должен выдать
```

```
  * дополнительные сведения о записи, если они имеются
  * (например, последний блок).
  *
  * Метод rm_mask принимает на вход страницу, изменённую менеджером ресурсов, и
  * маскирует биты, на изменение которых не должна реагировать проверка
  * wal_consistency_checking.
  *
  * Индексами в RmgrTable[] являются значения RmgrId (см. rmgrlist.h). Если
  * rm_name имеет значение NULL, соответствующая запись RmgrTable считается
  * недействительной.
 */
typedef struct RmgrData
{
    const char *rm_name;
    void        (*rm_redo) (XLogReaderState *record);
    void        (*rm_desc) (StringInfo buf, XLogReaderState *record);
    const char *(*rm_identify) (uint8 info);
    void        (*rm_startup) (void);
    void        (*rm_cleanup) (void);
    void        (*rm_mask) (char *pagedata, BlockNumber blkno);
    void        (*rm_decode) (struct LogicalDecodingContext *ctx,
                              struct XLogRecordBuffer *buf);
} RmgrData;
```

Затем зарегистрируйте новый менеджер ресурсов.

```
/*
 * Регистрация нового пользовательского менеджера ресурсов WAL.
 *
 * Идентификаторы менеджеров ресурсов должны быть глобально уникальными
 * среди всех расширений. Чтобы избежать конфликтов с расширениями других
 * разработчиков, зарезервируйте уникальный RmgrId для вашего расширения на
 * странице https://wiki.postgresql.org/wiki/CustomWALResourceManagers.
 * В процессе разработки следует использовать RM_EXPERIMENTAL_ID, чтобы
 * не резервировать лишние идентификаторы.
 */
extern void RegisterCustomRmgr(RmgrId rmid, const RmgrData *rmgr);
```

Функция `RegisterCustomRmgr` должна вызываться из функции _PG_init модуля расширения. В процессе разработки нового расширения в качестве *`rmid`* следует использовать `RM_EXPERIMENTAL_ID`. Когда расширение для пользователей будет готово к выпуску, зарезервируйте новый идентификатор менеджера ресурсов на странице *Custom WAL Resource Manager* (Пользовательский менеджер ресурсов WAL).

Включите модуль расширения, реализующий пользовательский менеджер ресурсов, в shared_preload_libraries, чтобы он загружался при запуске Postgres Pro.

Примечание

Расширение должно оставаться в `shared_preload_libraries` до тех пор, пока в системе могут быть какие-либо пользовательские записи WAL. В противном случае сервер Postgres Pro не сможет применять или декодировать пользовательские записи WAL, вследствие чего он может не запуститься.

Глава 62. Встроенные индексные методы доступа

62.1. Индексы B-деревья

62.1.1. Введение

Postgres Pro включает реализацию стандартной индексной структуры данных — B-дерева (btree, многонаправленного сбалансированного дерева). В индекс-B-дерево могут быть загружены данные любого типа, которые можно отсортировать в чётко определённом линейном порядке. Единственное его ограничение состоит в том, что размер записи в индексе не может превышать примерно треть страницы (после сжатия TOAST, если оно применяется).

Так как каждый класс операторов btree устанавливает порядок сортировки для своего типа данных, классы операторов btree (или, фактически, семейства операторов) оказались показательными и полезными для представления и понимания семантики сортировки в Postgres Pro. Как следствие, они приобрели некоторые возможности, которые выходят за рамки необходимого минимума для поддержки индексов btree и используются частями системы, довольно далёкими от методов доступа btree.

62.1.2. Поведение классов операторов B-дерева

Как показано в Таблице 36.3, класс операторов btree должен предоставить пять операторов сравнения, `<`, `<=`, `=`, `>=` и `>`. Хотя можно было ожидать, что частью этого класса будет и оператор `<>`, но это не так, потому что использовать `<>` в предложении WHERE для поиска по индексу практически бесполезно. (Для некоторых целей планировщик условно относит оператор `<>` к классу операторов btree, но он находит данный оператор как отрицание оператора `=`, а не обращаясь к `pg_amop`.)

Когда несколько типов данных имеют практически одинаковую семантику сортировки, их классы операторов можно сгруппировать в семейство операторов. Это полезно тем, что позволяет планировщику делать выводы о межтиповых сравнениях. Каждый класс операторов в семействе должен содержать операторы для одного своего типа входных данных (и сопутствующие опорные функции), тогда как межтиповые операторы сравнения и опорные функции являются «слабо» связанными с семейством. В семейство рекомендуется включать полный набор межтиповых операторов, чтобы планировщик мог представить любые условия, которые он может вывести, используя транзитивность.

Семейство операторов btree должно удовлетворять нескольким базовым положениям:

- Оператор `=` должен представлять отношение эквивалентности; то есть для всех отличных от NULL значений *A*, *B*, *C* определённого типа данных:
  - *A* `=` *A* — истина (*рефлексивность*)
  - если *A* `=` *B*, то *B* `=` *A* (*симметрия*)
  - если *A* `=` *B* и *B* `=` *C*, то *A* `=` *C* (*транзитивность*)
- Оператор `<` должен представлять отношение строгого упорядочивания; то есть для всех отличных от NULL значений *A*, *B*, *C*:
  - *A* `<` *A* — ложно (*антирефлексивность*)
  - если *A* `<` *B* и *B* `<` *C*, то *A* `<` *C* (*транзитивность*)
- Более того, упорядочивание действует глобально; то есть для любых отличных от NULL значений *A*, *B*:
  - истинным является ровно одно из условий: *A* `<` *B*, *A* `=` *B* или *B* `<` *A* (*трихотомия*)

(Разумеется, определение функции, осуществляющей сравнение, вытекает из закона трихотомии.)

Остальные три оператора определяются через операторы = и < очевидным образом и должны работать согласованно с последними.

Для семейства операторов, поддерживающего несколько типов данных, вышеперечисленные законы должны выполняться при значениях *A*, *B*, *C*, относящихся к любым типам из семейства. Транзитивность обеспечить сложнее всего, так как в ситуациях с разными типами она требует согласованного поведения двух или трёх различных операторов. Так например, в одном семействе операторов не смогут работать типы `float8` и `numeric`, по крайней мере при текущем подходе, когда значения `numeric` преобразуются во `float8` для сравнения с `float8`. Из-за ограниченной точности типа `float8` различные значения `numeric` могут оказаться равными одному значению `float8`, что нарушит закон транзитивности.

Ещё одно требование для семейства, рассчитанного на несколько типов данных, состоит в том, что любое неявное или двоично-совместимое приведение, которое определено между типами, включёнными в семейство операторов, не должно менять соответствующий порядок сортировки.

Должно быть достаточно понятно, почему индекс-B-дерево требует выполнения этих законов для одного типа данных: без этого упорядочивание ключей невозможно. Кроме того, для поиска в индексе по ключу другого типа данных необходимо, чтобы значения двух типов сравнивались корректно. Расширение семейства до трёх или более типов данных не является обязательным для самого механизма индекса btree, но может быть полезным для планировщика в целях оптимизации.

62.1.3. Опорные функции B-деревьев

Как показано в Таблице 36.9, btree определяет одну необходимую и четыре необязательных опорных функции. Таким образом, пользователь может задать пять методов:

`order`

Для всех комбинаций типов данных, для которых семейство операторов btree предоставляет операторы сравнения, оно должно предоставлять опорную функцию сравнения в `pg_amproc` с номером 1 и с `amproclefttype`/`amprocrighttype`, равными левому и правому типу сравнения (то есть тем же типам данных, с которыми соответствующие операторы зарегистрированы в `pg_amop`). Эта функция сравнения должна принимать два отличных от NULL значения *A* и *B* и возвращать значение `int32`, которое будет `< 0`, `0` или `> 0`, когда *A* `<` *B*, *A* `=` *B* или *A* `>` *B*, соответственно. Результат NULL не допускается: все значения типа данных должны быть сравнимыми.

Если сравниваемые значения имеют сортируемый тип данных, опорной функции сравнения будет передан OID соответствующего правила сортировки через стандартный механизм `PG_GET_COLLATION()`.

`sortsupport`

Дополнительно семейство операторов btree может предоставить функции *поддержки сортировки*, которые регистрируются под номером опорной функции 2. Эти функции позволяют реализовывать сравнения для целей сортировки гораздо эффективнее, чем это возможно при прямолинейном вызове функции поддержки сравнения. Задействованные в этом программные интерфейсы определены в *`src/include/utils/sortsupport.h`*.

`in_range`

Дополнительно семейство операторов btree может предоставить опорные функции *in_range*, которые регистрируются под номером 3. Они не используются в ходе операций с индексом btree; вместо этого они расширяют семантику семейства операторов, чтобы оно могло поддерживать оконные предложения `RANGE` *`смещение`* `PRECEDING` и `RANGE` *`смещение`* `FOLLOWING` (см. Подраздел 4.2.8). По сути они предоставляют дополнительную информацию, позволяющую добавлять или вычитать *`смещение`* в соответствии с порядком сортировки, принятым в семействе.

Функция `in_range` должна иметь сигнатуру

```
in_range(значение type1, база type1, смещение type2, вычитание bool, меньше bool)
returns bool
```

`Значение` и *`база`* должны быть одного типа данных, и этот тип должен поддерживаться семейством операторов (то есть это должен быть тип, для которого реализуется сортировка). Однако *`смещение`* может быть другого типа, который никаким другим образом не поддерживается данным семейством. Например, встроенное семейство `time_ops` предоставляет функцию, для которой *`смещение`* имеет тип `interval`. Семейство может предоставлять функции `in_range` для любых из своих поддерживаемых типов и одного или нескольких типов *`смещений`*. Каждая функция `in_range` должна регистрироваться в `pg_amproc` с полем `amproclefttype`, равным `type1`, и `amprocrighttype`, равным `type2`.

Суть действия функции `in_range` зависит от двух логических флагов. Она должна прибавить или вычесть из *`базы`* *`смещение`*, а затем сравнить *`значение`* с результатом следующим образом:

- если `!`*`вычитание`* и `!`*`меньше`*, возвращается *`значение`* `>=` (*`база`* `+` *`смещение`*)
- если `!`*`вычитание`* и *`меньше`*, возвращается *`значение`* `<=` (*`база`* `+` *`смещение`*)
- если *`вычитание`* и `!`*`меньше`*, возвращается *`значение`* `>=` (*`база`* `-` *`смещение`*)
- если *`вычитание`* и *`меньше`*, возвращается *`значение`* `<=` (*`база`* `-` *`смещение`*)

Прежде чем делать это, функция должна проверить знак *`смещения`* и, если оно отрицательное, выдать ошибку `ERRCODE_INVALID_PRECEDING_OR_FOLLOWING_SIZE` (22013) с текстом ошибки «invalid preceding or following size in window function» (неверная предшествующая или последующая величина в оконной функции). (Это требуется стандартом SQL, но нестандартные семейства операторов могут проигнорировать данное ограничение, так как оно не несёт большой смысловой нагрузки.) Проверка этого требования делегируется функции `in_range`, чтобы коду ядра не требовалось понимать, что означает «меньше нуля» для произвольного типа данных.

Кроме того, функции `in_range`, если это практично, могут не выдавать ошибку, когда операция *`база`* `+` *`смещение`* или *`база`* `-` *`смещение`* приводит к переполнению. Правильный результат сравнения можно получить, даже если это значение выходит за границы допустимого диапазона этого типа данных. Заметьте, что если для типа данных определены такие понятия, как «бесконечность» и «NaN», могут потребоваться дополнительные меры для обеспечения согласованности результатов `in_range` с обычным порядком сортировки данного семейства операторов.

Результаты функции `in_range` должны соответствовать порядку сортировки, устанавливаемому семейством операторов. Точнее говоря, при любых фиксированных аргументах *`смещение`* и *`вычитание`* справедливо:

- Если `in_range` с *`меньше`* = true возвращает true для некоторого *`значения1`* и *`базы`*, true должно возвращаться для каждого *`значения2`* `<=` *`значению1`* с той же *`базой`*.
- Если `in_range` с *`меньше`* = true возвращает false для некоторого *`значения1`* и *`базы`*, false должно возвращаться для любого *`значения2`* `>=` *`значению1`* с той же *`базой`*.
- Если `in_range` с *`меньше`* = true возвращает true для некоторого *`значения`* и *`базы1`*, true должно возвращаться для каждой *`базы2`* `>=` *`базе1`* с тем же *`значением`*.
- Если `in_range` с *`меньше`* = true возвращает false для некоторого *`значения`* и *`базы1`*, false должно возвращаться для любой *`базы2`* `<=` *`базе1`* с тем же *`значением`*.

Аналогичные утверждения с противоположными условиями должны выполняться при *`меньше`* = false.

Если упорядочиваемый тип (`type1`) является сортируемым, функции `in_range` будет передан OID соответствующего правила сортировки через стандартный механизм PG_GET_COLLATION().

Функции `in_range` не должны обрабатывать NULL в аргументах и обычно помечаются как строгие.

`equalimage`

Дополнительно семейство операторов btree может предоставить опорные функции `equalimage` («равенство подразумевает равенство образов»), регистрируемые под номером 4. Эти функции позволяют коду ядра определить, безопасно ли применять исключение дубликатов в B-дереве. В настоящее время функции `equalimage` вызываются только при построении или перестроении индекса.

Функция `equalimage` должна иметь сигнатуру

```
equalimage(opcintype oid) returns bool
```

Её результатом будет статическая информация о классе операторов и правиле сортировки. Результат `true` означает, что функция `order` для класса операторов будет возвращать `0` (признак равенства аргументов), только когда аргументы *`A`* и *`B`* взаимозаменяемы без потери семантической информации. Если функция `equalimage` не определена или она возвращает `false`, рассчитывать на выполнение данного условия нельзя.

В аргументе *`opcintype`* передаётся `pg_type.oid` типа данных, индексируемого данным классом операторов. Это сделано для удобства повторного использования нижележащей функции `equalimage` в разных классах операторов. Если тип *`opcintype`* поддерживает правила сортировки, функции `equalimage` будет передан OID соответствующего правила через стандартный механизм `PG_GET_COLLATION()`.

С точки зрения класса операторов возвращаемое значение `true` означает, что возможно безопасное применение исключения дубликатов (или оно безопасно для правила сортировки, OID которого был передан функции `equalimage`). Однако код ядра будет считать исключение дубликатов безопасным для индекса, только если для *каждого* столбца в этом индексе используется класс операторов, регистрирующий функцию `equalimage`, и все эти функции при вызове возвращают `true`.

Равенство образов *почти* равнозначно простому битовому равенству. Но есть одно небольшое различие: когда индексируется тип данных varlena, представление двух равных образов на диске может отличаться из-за различного применения сжатия TOAST к входным данным. Говоря формально, когда функция `equalimage` класса операторов возвращает `true`, можно полагать, что функция на C `datum_image_eq()` гарантированно будет согласованной с функцией `order` класса операторов (при условии передачи обеим функциям одинакового OID правила сортировки).

Код ядра в принципе не может сделать какие-то выводы о свойстве класса операторов «равенство подразумевает равенство образов» в семействе операторов для множества типов, анализируя другие классы операторов в том же семействе. Также не имеет смысла регистрировать межтиповую функцию `equalimage` для семейства операторов, и при попытке сделать это произойдёт ошибка. Это связано с тем, что свойство «равенство подразумевает равенство образов» зависит не только от семантики сортировки/равенства, определяемой в некоторой степени на уровне семейства операторов. Вообще говоря, это свойство относится к конкретному типу и должно рассматриваться отдельно.

Для классов операторов, поставляемых в базовом продукте Postgres Pro, принято соглашение регистрировать универсальную функцию `equalimage`. Большинство классов операторов регистрируют в качестве такой функции `btequalimage()`, которая устанавливает, что исключение дубликатов безопасно без дополнительных условий. Операторы классов для типов данных, поддерживающих правила сортировки, например, для типа `text`, регистрируют функцию `btvarstrequalimage()`, которая устанавливает, что исключение дубликатов безопасно с детерминированными правилами сортировки. Для сохранения порядка в сторонних расширениях также рекомендуется регистрировать их собственные функции `equalimage`.

`options`

В дополнение семейство операторов btree может предоставить опорные функции `options` («параметры класса операторов»), регистрируемые под номером 5. Эти функции позволяют определить набор видимых пользователю параметров, управляющих поведением класса операторов.

Опорная функция `options` должна иметь сигнатуру

```
options(relopts local_relopts *) returns void
```

Этой функции передаётся указатель на структуру *`local_relopts`*, в которую нужно внести набор параметров, относящихся к классу операторов. Обращаться к этим параметрам из других опорных функций можно с помощью макросов `PG_HAS_OPCLASS_OPTIONS()` и `PG_GET_OPCLASS_OPTIONS()`.

В настоящее время опорная функция `options` не определена ни для одного из классов операторов btree. Сама организация B-дерева не позволяет гибко менять представление ключей, как это возможно с GiST, SP-GiST, GIN и BRIN. Поэтому с существующим методом доступа к индексу-B-дереву для функции `options` нет полезных применений. Тем не менее эта опорная функция была добавлена для B-дерева ради единообразия и не исключено, что она окажется полезной по мере развития реализации B-дерева в Postgres Pro.

62.1.4. Реализация

В этом разделе освещаются детали реализации индекса-B-дерева, знание которых может быть полезно для специалистов.

62.1.4.1. Структура B-дерева

Индексы-B-деревья в Postgres Pro представляют собой многоуровневые иерархические структуры, в которых каждый уровень дерева может использоваться как двусвязный список страниц. Единственная метастраница индекса хранится в фиксированной позиции в начале первого файла сегмента индекса. Все остальные страницы делятся на внутренние и на листовые. Листовые страницы находятся на самом нижнем уровне дерева. Все более высокие уровни состоят из внутренних страниц. Листовая страница содержит кортежи, указывающие на строки в таблице, а внутренняя страница — кортежи, указывающие на следующий уровень в дереве. Обычно листовые страницы составляют около 99% всех страниц индекса. И для тех, и для других страниц используется один стандартный формат, описанный в Разделе 63.6.

Новые листовые страницы добавляются в B-дерево когда существующая листовая страница не может вместить новый поступающий кортеж. При этом выполняется операция *разделения страницы*, освобождающая место на переполнившейся странице, перенося подмножество изначально содержащихся на ней элементов на новую страницу. При разделении страницы в её родительскую страницу также должна быть добавлена *ссылка вниз*, что может потребовать произвести разделение и этой родительской страницы. Разделение страниц «каскадно поднимается вверх» рекурсивным образом. Когда же и корневая страница не может вместить новую ссылку вниз, производится операция *разделения корневой страницы*. При этом в структуру дерева добавляется новый уровень, на котором оказывается новая корневая страница, стоящая над той, что была корневой ранее.

62.1.4.2. Восходящее удаление индексных кортежей

В реализации индексов-B-деревьев не учитывается, что в среде MVCC может быть несколько версий одной логической строки таблицы; для индекса каждый кортеж является независимым объектом, требующим отдельного элемента в индексе. Кортежи «отработанных версий» иногда могут накапливаться, что чревато задержками и замедлением при выполнении запросов. Это обычно происходит при нагрузке с преобладанием `UPDATE`, когда для большинства отдельных операций изменения данных нельзя применить оптимизацию HOT. Изменение значения даже одного индексируемого столбца во время `UPDATE` *всегда* требует добавления нового набора индексных кортежей — отдельного кортежа для *каждого* индекса в таблице. Более того, обратите внимание, что новые кортежи требуются и для тех индексов, которые не были «логически изменены» командой `UPDATE`. Во всех индексах будут нужны дополнительные физические кортежи, указывающие на последнюю версию строки в таблице. Каждый новый кортеж в каждом индексе, как правило, должен сосуществовать с исходным подвергшимся изменению кортежем в течение короткого периода времени (обычно недолго после фиксации транзакции `UPDATE`).

В индексах-B-деревьях постепенно удаляются кортежи отработанных версий в ходе процедуры *восходящего удаления индексных кортежей*. Каждый проход процедуры удаления вызывается, когда ожидается, что произойдёт «разделение страниц из-за отрабатывания версий». Это касается только тех индексов, которые не были логически изменены операторами `UPDATE`, так как именно в них возможно концентрированное накопление устаревших версий на определённых страницах. Реализация обычно старается избежать разделения страниц, хотя вполне возможно, что определённые решения на её уровне не позволят идентифицировать и удалить ни одного мусорного кортежа в индексе (в этом случае проблема размещения нового кортежа на заполненной листовой странице решается путём разделения страницы или в результате исключения дубликатов). Большое количество версий каждой отдельной логической строки, которое нужно прочитать при каждом сканировании индекса, является важным отрицательным фактором общей производительности и скорости отклика системы. Процедура восходящего удаления индексных кортежей выбирает на листовой странице предположительно мусорные кортежи на основании *качественных* характеристик, определяемых логическими строками и версиями. Это отличает её от «нисходящей» уборки индекса, выполняемой процессами автоочистки, которая запускается при превышении определённых *количественных* пороговых значений на уровне таблицы (см. Подраздел 23.1.6).

Примечание

Не все операции удаления, выполняемые в индексах-B-деревьях, реализуются процедурой восходящего удаления. Существует другая категория таких операций: *простое удаление индексных кортежей*. Это отложенная операция обслуживания, удаляющая индексные кортежи, о которых известно, что их можно безопасно стереть (то есть те, для идентификаторов которых установлен бит `LP_DEAD`). Как и восходящее удаление индексных кортежей, простое удаление происходит в тот момент, когда ожидается разделение страницы, для предотвращения этого разделения.

Простое удаление выполняется по возможности в том смысле, что оно может произойти только при условии, что в результате недавних сканирований индекса были установлены биты `LP_DEAD` для обработанных элементов. До Postgres Pro 14 единственной категорией операций удаления в B-дереве было простое удаление. Основное различие между простым и восходящим удалением заключается в том, что только первое обусловлено активностью сканирований индекса, а второе ориентировано именно на активность отрабатывания версий, вызываемую командами `UPDATE`, которые не изменяют логически индексированные столбцы.

При определённой нагрузке восходящее удаление индексных кортежей выполняет основную работу по уборке мусорных кортежей из индексов. Такой эффект ожидается для любого индекса-B-дерева, который в значительной мере затрагивается активностью отрабатывания версий из-за команд `UPDATE`, почти никогда не изменяющих логически столбцы, покрываемые индексом. Среднее и наибольшее количество версий для логической строки может поддерживаться на низком уровне исключительно за счёт постоянных точечных проходов удаления. Вполне возможно, что размер определённых индексов на диске никогда не увеличится ни на одну страницу/блок, несмотря на *постоянное* отрабатывание версий, вызываемое командами `UPDATE`. Но даже в этом случае в конце концов потребуется полная «зачистка» индекса процедурой `VACUUM` (обычно запускаемой в рабочем процессе автоочистки) как часть *совместной* уборки таблицы и всех её индексов.

В отличие от `VACUUM`, восходящее удаление не даёт никаких надёжных гарантий относительно возраста старейшего мусорного индексного кортежа. Ни в одном индексе не допускается сохранение «плавающих мусорных» кортежей, ставших мёртвыми до консервативной точки отсечения, являющейся общей для таблицы и всех её индексов. Этот фундаментальный инвариант на уровне таблицы позволяет обеспечивать безопасную циркуляцию табличных идентификаторов (TID). Именно таким образом разные логические строки могут повторно использовать один и тот же идентификатор с течением времени (хотя это невозможно для двух логических строк, время жизни которых укладывается в один и тот же цикл `VACUUM`).

62.1.4.3. Исключение дубликатов

Дубликатом называется кортеж на листовой странице (кортеж, указывающий на строку таблицы), у которого *все* ключевые столбцы индекса имеют значения, соответствующие значениям столбцов из как минимум одного другого кортежа на листовой странице в том же индексе. Дублирующиеся кортежи довольно часто встречаются на практике. В индексах-B-деревьях такие дубликаты могут представляться особым экономичным образом при включении дополнительного механизма — *исключения дубликатов*.

Работа этого механизма заключается в периодическом объединении групп дублирующихся кортежей и формировании одного кортежа со *списком идентификаторов* для каждой группы. В таком представлении значения ключевых столбцов хранятся в единственном экземпляре, а за ними идёт отсортированный массив идентификаторов TID, указывающих на строки в таблице. Это существенно уменьшает размер хранящихся индексов, в которых каждое значение (или каждое уникальное сочетание значений столбцов) появляется в среднем несколько раз. В результате может значительно увеличиться скорость выполнения запросов, а также могут сократиться издержки, связанные с регулярной очисткой индексов.

Примечание

Исключение дубликатов в B-дереве работает эффективно и с «дубликатами», содержащими значение NULL, несмотря на то, что значения NULL не считаются равными между собой согласно операторам =, входящим в классы операторов btree. Это объясняется тем, что с точки зрения реализации, работающей с внутренним представлением структуры B-дерева, NULL является просто одним из элементов множества всех возможных значений в индексе.

Процесс исключения дубликатов осуществляется по необходимости, когда вставляется новый элемент, не умещающийся на существующей листовой странице, но только тогда, когда удаление индексного кортежа не может освободить достаточно места для нового элемента (обычно удаление недолго рассматривается, а затем пропускается). В отличие от кортежей со списками идентификаторов GIN, в B-дереве эти кортежи не должны расширяться при каждом добавлении нового дубликата; они просто образуют другое физическое представление исходного логического содержимого листовой страницы. При таком подходе обеспечивается стабильная производительность при смешанной нагрузке чтения-записи. Исключение дубликатов должно дать как минимум заметное увеличение производительности для большинства клиентских приложений. По умолчанию оно включено.

Команды `CREATE INDEX` и `REINDEX` также выполняют исключение дубликатов, создавая кортежи со списками идентификаторов, но применяют несколько другую стратегию. Каждая группа дублирующихся обычных кортежей, обнаруженных в отсортированных данных, преобразуется в кортеж со списком идентификаторов *до того*, как данные добавляются в текущую листовую страницу. При этом в каждый такой кортеж упаковывается как можно больше идентификаторов (TID). После этого листовые страницы записываются обычным способом, без дополнительного прохода для исключения дубликатов. Эта стратегия подходит для команд `CREATE INDEX` и `REINDEX`, так как они обрабатывают все данные сразу.

Если же в профиле нагрузки преобладает запись и исключение дубликатов не приносит выигрыша ввиду отсутствия или небольшого числа дублирующихся значений, этот механизм может породить небольшие постоянные издержки (если он не отключён). В таких случаях его можно отключить для отдельных индексов с помощью параметра хранения `deduplicate_items`. При нагрузке только на чтение никакие дополнительные издержки не возникают, так как кортежи со списком идентификаторов читаются так же эффективно, как и кортежи в стандартном представлении. Поэтому чаще всего отключение этого механизма не будет полезным.

Исключение дубликатов иногда может использоваться для уникальных индексов (а также уникальных ограничений). Благодаря этому дополнительные дубликаты отработанных версий могут временно «поглощаться» листовыми страницами. Исключение дубликатов в уникальных индексах дополняет восходящее удаление индексных кортежей, особенно в тех случаях, когда длительная транзакция держит снимок, препятствующий сборке мусора. Тем самым выигрывается время для восстановления эффективности стратегии восходящего удаления индексных кортежей. Откладывание разделения страниц до естественного завершения одной длительной транзакции позволяет успешно произвести проход восходящего удаления, который ранее был невозможен.

Подсказка

Для определения необходимости провести процедуру исключения дубликатов в уникальном индексе применяются дополнительные соображения. В таких индексах как правило можно перейти сразу к разделению листовой страницы, не расходуя лишние циклы на бесполезные проходы в поиске дубликатов. Если вас беспокоят возможные издержки, которые могут быть связаны с исключением дубликатов, вы можете установить значение `deduplicate_items = off` для отдельных индексов. Однако его вполне можно оставить включённым и для уникальных индексов.

Исключение дубликатов может применяться не всегда ввиду ограничений на уровне реализации. Возможность его применения определяется во время выполнения `CREATE INDEX` или `REINDEX`.

Учтите, что исключение дубликатов считается небезопасным и не может применяться в следующих случаях, когда возможны семантические различия равных значений:

- Исключение дубликатов не может применяться с типами `text`, `varchar` и `char` в случае использования *недетерминированных* правил сортировки, так как в равных значениях должны сохраняться возможные различия в регистре и диакритических знаках.
- Исключение дубликатов невозможно с типом `numeric`, так как для равных значений должен сохраняться числовой масштаб, который может быть разным.
- Исключение дубликатов не может применяться с типом `jsonb`, так как внутри класса операторов B-дерева `jsonb` используется тип `numeric`.
- Исключение дубликатов невозможно для типов `float4` и `float8`. В этих типах имеются разные представления значений `-0` и `0`, которые при этом считаются равными. Однако отличие между ними должно сохраняться.

Имеется ещё одно ограничение на уровне реализации, которое может быть снято в будущих версиях Postgres Pro:

- Исключение дубликатов невозможно с типами-контейнерами (это составные, диапазонные типы, а также массивы).

Есть ещё одно ограничение на уровне реализации, действующее вне зависимости от применяемого класса операторов или правила сортировки:

- Исключение дубликатов не может применяться в индексах с `INCLUDE`.

62.2. Индексы GiST

62.2.1. Введение

GiST расшифровывается как «Generalized Search Tree» (Обобщённое поисковое дерево). Это сбалансированный иерархический метод доступа, который представляет собой базовый шаблон, на основе которого могут реализовываться произвольные схемы индексации. На базе GiST могут быть реализованы B-деревья, R-деревья и многие другие схемы индексации.

Ключевым преимуществом GiST является то, что он позволяет разрабатывать дополнительные типы данных с соответствующими методами доступа экспертам в предметной области типа данных, а не специалистам по СУБД.

Представленная здесь информация частично позаимствована с *сайта* Проекта индексации GiST Калифорнийского университета в Беркли и из диссертации Марселя Корнакера *Методы доступа для СУБД следующего поколения*. Сопровождением реализации GiST в Postgres Pro в основном занимаются Фёдор Сигаев и Олег Бартунов; дополнительные сведения можно получить на их *сайте*.

62.2.2. Встроенные классы операторов

В базовый дистрибутив Postgres Pro включены классы операторов GiST, перечисленные в Таблице 62.1. (Некоторые дополнительные модули, описанные в Приложении F, добавляют другие классы операторов GiST.)

Таблица 62.1. Встроенные классы операторов GiST

Имя	Индексируемые операторы	Операторы упорядочивания
`box_ops`	`<< (box, box)`	`<-> (box, point)`
	`&< (box, box)`	
	`&& (box, box)`	
	`&> (box, box)`	
	`>> (box, box)`	
	`~= (box, box)`	
	`@> (box, box)`	
	`<@ (box, box)`	
	`&<| (box, box)`	
	`<<| (box, box)`	
	`|>> (box, box)`	
	`|&> (box, box)`	
`circle_ops`	`<< (circle, circle)`	`<-> (circle, point)`
	`&< (circle, circle)`	
	`&> (circle, circle)`	
	`>> (circle, circle)`	
	`<@ (circle, circle)`	
	`@> (circle, circle)`	
	`~= (circle, circle)`	
	`&& (circle, circle)`	
	`|>> (circle, circle)`	
	`<<| (circle, circle)`	
	`&<| (circle, circle)`	
	`|&> (circle, circle)`	
`inet_ops`	`<< (inet, inet)`	
	`<<= (inet, inet)`	
	`>> (inet, inet)`	
	`>>= (inet, inet)`	
	`= (inet, inet)`	
	`<> (inet, inet)`	
	`< (inet, inet)`	

Имя	Индексируемые операторы	Операторы упорядочивания
	`<= (inet, inet)`	
	`> (inet, inet)`	
	`>= (inet, inet)`	
	`&& (inet, inet)`	
`multirange_ops`	`= (anymultirange, anymultirange)`	
	`&& (anymultirange, anymultirange)`	
	`&& (anymultirange, anyrange)`	
	`@> (anymultirange, anyelement)`	
	`@> (anymultirange, anymultirange)`	
	`@> (anymultirange, anyrange)`	
	`<@ (anymultirange, anymultirange)`	
	`<@ (anymultirange, anyrange)`	
	`<< (anymultirange, anymultirange)`	
	`<< (anymultirange, anyrange)`	
	`>> (anymultirange, anymultirange)`	
	`>> (anymultirange, anyrange)`	
	`&< (anymultirange, anymultirange)`	
	`&< (anymultirange, anyrange)`	
	`&> (anymultirange, anymultirange)`	
	`&> (anymultirange, anyrange)`	
	`-|- (anymultirange, anymultirange)`	
	`-|- (anymultirange, anyrange)`	
`point_ops`	`|>> (point, point)`	`<-> (point, point)`
	`<< (point, point)`	
	`>> (point, point)`	
	`<<| (point, point)`	
	`~= (point, point)`	
	`<@ (point, box)`	
	`<@ (point, polygon)`	
	`<@ (point, circle)`	
`poly_ops`	`<< (polygon, polygon)`	`<-> (polygon, point)`
	`&< (polygon, polygon)`	
	`&> (polygon, polygon)`	
	`>> (polygon, polygon)`	
	`<@ (polygon, polygon)`	
	`@> (polygon, polygon)`	
	`~= (polygon, polygon)`	
	`&& (polygon, polygon)`	
	`<<| (polygon, polygon)`	

Имя	**Индексируемые операторы**	**Операторы упорядочивания**
	`&<| (polygon, polygon)`	
	`|&> (polygon, polygon)`	
	`|>> (polygon, polygon)`	
`range_ops`	`= (anyrange, anyrange)`	
	`&& (anyrange, anyrange)`	
	`&& (anyrange, anymultirange)`	
	`@> (anyrange, anyelement)`	
	`@> (anyrange, anyrange)`	
	`@> (anyrange, anymultirange)`	
	`<@ (anyrange, anyrange)`	
	`<@ (anyrange, anymultirange)`	
	`<< (anyrange, anyrange)`	
	`<< (anyrange, anymultirange)`	
	`>> (anyrange, anyrange)`	
	`>> (anyrange, anymultirange)`	
	`&< (anyrange, anyrange)`	
	`&< (anyrange, anymultirange)`	
	`&> (anyrange, anyrange)`	
	`&> (anyrange, anymultirange)`	
	`-|- (anyrange, anyrange)`	
	`-|- (anyrange, anymultirange)`	
`tsquery_ops`	`<@ (tsquery, tsquery)`	
	`@> (tsquery, tsquery)`	
`tsvector_ops`	`@@ (tsvector, tsquery)`	

По историческим причинам класс операторов `inet_ops` не является классом по умолчанию для типов `inet` и `cidr`. Чтобы использовать его, укажите имя класса в `CREATE INDEX`, например:

```
CREATE INDEX ON my_table USING GIST (my_inet_column inet_ops);
```

62.2.3. Расширяемость

Реализация нового метода доступа индекса традиционно была большой и сложной задачей. Чтобы её решить, необходимо было понимать внутреннее устройство базы данных, в частности работу менеджера блокировок и журнала предзаписи. Но с интерфейсом GiST, реализующим высокий уровень абстракции, разработчик метода доступа должен реализовать только смысловое наполнение индексируемого типа данных. Уровень GiST берёт на себя заботу о параллельном доступе, поддержке журнала и поиске в структуре дерева.

Эту расширяемость не следует путать с расширяемостью других стандартных деревьев поиска в смысле поддержки различных типов данных. Например, Postgres Pro поддерживает расширяемость B-деревьев и индексов по хешу. Это означает, что в Postgres Pro вы можете построить B-дерево или хеш-таблицу по любому желаемому типу данных. Но такие B-деревья будут поддерживать только предикаты сравнений (<, =, >), а индексы по хешу только запросы с равенством.

Поэтому, если вы проиндексируете в Postgres Pro в B-дереве, например, коллекцию изображений, вы сможете выполнять только проверки вида «равны ли изображения X и Y», «меньше ли изображение X изображения Y» и «больше ли изображение X изображения Y». Это может быть полезно, в зависимости от того, как вы определите операции «равно», «меньше» и «больше». Однако используя индекс на базе GiST, возможно удовлетворять и запросы из предметной области, например, «найти все изображения лошадей» или «найти все пересвеченные изображения».

Всё, что нужно, чтобы получить работающий метод доступа GiST — это реализовать несколько методов, определяющих поведение ключей в дереве. Конечно, эти методы должны быть довольно изощрёнными, чтобы поддерживать изощрённые запросы, но для всех стандартных запросов (B-деревьев, R-деревьев и т. д.) они относительно просты. Словом, GiST сочетает расширяемость с универсальностью, повторным использованием кода и аккуратным интерфейсом.

Класс операторов индекса GiST должен предоставить пять методов и может дополнительно предоставлять ещё шесть. Корректность индекса обеспечивается реализацией методов `same`, `consistent` и `union`, а его эффективность (по размеру и скорости) будет зависеть от методов `penalty` и `picksplit`. Два необязательных метода, `compress` и `decompress`, позволяют реализовать внутреннее представление данных дерева, не совпадающее с типом индексируемых данных. Данные листьев индекса должны быть индексируемого типа, тогда как в остальных узлах дерева могут быть произвольные структуры C (но при этом должны соблюдаться правила, предъявляемые Postgres Pro к типам данных; прочитайте о `varlena` для данных переменного размера). Если внутренний тип данных дерева существует на уровне SQL, в команде `CREATE OPERATOR CLASS` можно использовать указание `STORAGE`. Необязательный восьмой метод `distance` нужно реализовать, только если класс операторов желает поддерживать упорядоченные сканирования (поиск ближайших соседей). Необязательный девятый метод `fetch` требуется, если класс операторов должен поддерживать сканирование только индекса и при этом предоставляется метод `compress`. Необязательный десятый метод `options` необходим, если класс операторов содержит определяемые пользователем параметры. Необязательный одиннадцатый метод `sortsupport` используется для ускорения построения индекса GiST.

`consistent`

Для переданной записи индекса `p` и значения запроса `q` эта функция определяет, является ли запись индекса «соответствующей» запросу; то есть, может ли предикат «*индексированный_столбец* *индексируемый_оператор* `q`» удовлетворяться для какой-либо строки, представленной данной записью индекса? Для записей на уровне листьев это равносильно проверке индексируемого условия, тогда как для внутреннего узла дерева требуется определить, нужно ли сканировать поддерево индекса, относящееся к данному узлу. Когда результат `true`, также должен возвращаться флаг `recheck`, показывающий, точно ли удовлетворяется предикат или это лишь потенциально возможно. Если `recheck` = `false`, это означает, что индекс проверил условие предиката в точности, тогда как при `recheck` = `true` проверяемая строка будет только кандидатом на совпадение. В этом случае система автоматически перепроверит *индексируемый_оператор* с действительным значением строки, чтобы окончательно определить, соответствует ли оно запросу. Благодаря этому GiST поддерживает индексы как точной, так и неточной структуры.

В SQL эта функция должна объявляться примерно так:

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid,
 internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

А соответствующий код в модуле C может реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
```

```
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry->key);
    bool        retval;

    /*
     * Определить возвращаемое значение как функцию стратегии, ключа и запроса.
     *
     * Вызовите GIST_LEAF(entry), чтобы узнать текущую позицию в дереве индекса,
     * что удобно, например для поддержки оператора = (вы можете проверить
     * равенство в листьях дерева и непустое пересечение в остальных
     * узлах).
     */

    *recheck = true;        /* или false, если проверка точная */

    PG_RETURN_BOOL(retval);
}
```

Здесь `key` — это элемент в индексе, а `query` — значение, искомое в индексе. Параметр `StrategyNumber` показывает, какой оператор из класса операторов применяется — он соответствует одному из номеров операторов, заданных в команде `CREATE OPERATOR CLASS`.

В зависимости от того, какие операторы включены в класс, тип данных `query` может быть разным для разных операторов, так как это будет тот тип, что фигурирует в правой части оператора, и он может отличаться от индексируемого типа данных, фигурирующего слева. (В показанном выше шаблоне предполагается, что допускается только один тип; в противном случае получение значения `query` зависело бы от оператора.) В SQL-объявлении функции `consistent` для аргумента `query` рекомендуется установить индексированный тип данного класса операторов, хотя фактический тип может быть каким-то другим, в зависимости от оператора.

`union`

Этот метод консолидирует информацию в дереве. Получив набор записей, он должен сгенерировать в индексе новую запись, представляющие все эти записи.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS storage_type
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec->vector;
    data_type  *out,
               *tmp,
               *old;
    int         numranges,
                i = 0;

    numranges = entryvec->n;
    tmp = DatumGetDataType(ent[0].key);
```

```
    out = tmp;

    if (numranges == 1)
    {
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i < numranges; i++)
    {
        old = out;
        tmp = DatumGetDataType(ent[i].key);
        out = my_union_implementation(out, tmp);
    }

    PG_RETURN_DATA_TYPE_P(out);
}
```

Как можно заметить, в этом шаблоне мы имеем дело с типом данных, для которого `union(X, Y, Z) = union(union(X, Y), Z)`. Достаточно просто можно поддержать и такие типы данных, для которых это не выполняется, реализовав соответствующий алгоритм объединения в этом опорном методе GiST.

Результатом функции `union` должно быть значение типа хранения индекса, каким бы он ни был (он может совпадать с типом индексированного столбца, а может и отличаться от него). Функция, реализующая `union`, должна возвращать указатель на память, выделенную вызовом `palloc()`. Она не может просто вернуть полученное значение как есть, даже если оно имеет тот же тип.

Как показано выше, первый аргумент `internal` функции `union` на самом деле представляет указатель `GistEntryVector`. Во втором аргументе (его можно игнорировать) передаётся указатель на целочисленную переменную. (Раньше требовалось, чтобы функция `union` сохраняла в этой переменной размер результирующего значения, но теперь такого требования нет.)

`compress`

Преобразует элемент данных в формат, подходящий для физического хранения в странице индекса. Если метод `compress` не реализован, элементы данных хранятся в индексе без модификации.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *retval;

    if (entry->leafkey)
    {
        /* заменить entry->key сжатой версией */
```

```
        compressed_data_type *compressed_data =
 palloc(sizeof(compressed_data_type));

        /* заполнить *compressed_data из entry->key ... */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset, FALSE);
    }
    else
    {
        /* обычно с записями внутренних узлов ничего делать не нужно */
        retval = entry;
    }

    PG_RETURN_POINTER(retval);
}
```

Разумеется, *`compressed_data_type`* (тип сжатых данных) нужно привести к нужному типу, при преобразовании в который будут сжиматься узлы на уровне листьев.

`decompress`

Преобразует сохранённое представление данных в формат, с которым смогут работать другие методы GiST в классе операторов. Если метод `decompress` опускается, подразумевается, что эти методы могут работать непосредственно с форматом хранения данных. (Метод `decompress` не обязательно будет обратным к `compress`; в частности, если функция `compress` сохраняет данные с потерями, `decompress` не сможет восстановить в точности исходные данные. Поэтому метод `decompress` в общем случае неравнозначен `fetch`, так как другим методам GiST может не потребоваться восстанавливать данные полностью.)

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

Этот шаблон подходит для случая, когда преобразовывать данные не нужно. (Но, разумеется, ещё проще и в большинстве случаев рекомендуется вовсе опустить этот метод.)

`penalty`

Возвращает значение, выражающее «стоимость» добавления новой записи в конкретную ветвь дерева. Элементы будут вставляться по тому направлению в дереве, для которого значение `penalty` минимально. Результаты `penalty` должны быть неотрицательными; если возвращается отрицательное значение, оно воспринимается как ноль.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
```

```
LANGUAGE C STRICT;  -- в некоторых случаях функции стоимости не должны быть строгими
```

И соответствующий код в модуле C может реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry->key);
    data_type  *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

По историческим причинам функция `penalty` не просто возвращает результат типа `float`; вместо этого она должна сохранить его значение по адресу, указанному третьим аргументом. Собственно возвращаемое значение игнорируется, хотя в нём принято возвращать этот же адрес.

Функция `penalty` важна для хорошей производительности индекса. Она будет вызываться во время добавления записи, чтобы выбрать ветвь для дальнейшего движения, когда в дерево нужно добавить новый элемент. Это имеет значение во время запроса, так как чем более сбалансирован индекс, тем быстрее будет поиск в нём.

`picksplit`

Когда необходимо разделить страницу индекса, эта функция решает, какие записи должны остаться в старой странице, а какие нужно перенести в новую.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C может реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY  *ent = entryvec->vector;
    int         i,
                nbytes;
    OffsetNumber *left,
               *right;
    data_type  *tmp_union;
    data_type  *unionL;
    data_type  *unionR;
    GISTENTRY **raw_entryvec;

    maxoff = entryvec->n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);
```

```
    v->spl_left = (OffsetNumber *) palloc(nbytes);
    left = v->spl_left;
    v->spl_nleft = 0;

    v->spl_right = (OffsetNumber *) palloc(nbytes);
    right = v->spl_right;
    v->spl_nright = 0;

    unionL = NULL;
    unionR = NULL;

    /* Инициализировать чистый вектор записи. */
    raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec->vector[i]);

    for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    {
        int         real_index = raw_entryvec[i] - entryvec->vector;

        tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
        Assert(tmp_union != NULL);

        /*
         * Выбрать, куда помещать записи индекса и изменить unionL и unionR
         * соответственно. Добавить записи в v->spl_left или
         * v->spl_right и увеличить счётчики.
         */

        if (my_choice_is_left(unionL, curl, unionR, curr))
        {
            if (unionL == NULL)
                unionL = tmp_union;
            else
                unionL = my_union_implementation(unionL, tmp_union);

            *left = real_index;
            ++left;
            ++(v->spl_nleft);
        }
        else
        {
            /*
             * То же самое с правой стороной
             */
        }
    }

    v->spl_ldatum = DataTypeGetDatum(unionL);
    v->spl_rdatum = DataTypeGetDatum(unionR);
    PG_RETURN_POINTER(v);
}
```

Заметьте, что результат функции `picksplit` доставляется через полученную на вход структуру `v`. Собственно возвращаемое значение игнорируется, хотя в нём принято возвращать адрес `v`.

Как и `penalty`, функция `picksplit` важна для хорошей производительности индекса. Сложность создания быстродействующих индексов GiST заключается как раз в разработке подходящих реализаций `penalty` и `picksplit`.

`same`

Возвращает true, если два элемента индекса равны, и false в противном случае. («Элемент индекса» — это значение типа хранения индекса, а не обязательно исходного типа индексируемого столбца.)

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_same(storage_type, storage_type, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C может реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool        *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

По историческим причинам функция `same` не просто возвращает результат булевого типа; вместо этого она должна сохранить флаг по адресу, указанному третьим аргументом. Собственно возвращаемое значение игнорируется, хотя в нём принято возвращать этот же адрес.

`distance`

Для переданной записи индекса `p` и значения запроса `q` эта функция определяет «дистанцию» от записи индекса до значения в запросе. Эта функция должна быть представлена, если класс операторов содержит какие-либо операторы упорядочивания. Запрос с оператором упорядочивания будет выполняться так, чтобы записи индекса с наименьшей «дистанцией» возвращались первыми, так что результаты должны согласовываться со смысловым значением оператора. Для записи на уровне листьев результат представляет только дистанцию до этой записи, а для внутреннего узла дерева это будет минимальная дистанция, которая может быть получена среди всех его потомков.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid, internal)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

И соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    /* bool *recheck = (bool *) PG_GETARG_POINTER(4); */
    data_type  *key = DatumGetDataType(entry->key);
    double      retval;
```

```
    /*
     * определить возвращаемое значение как функцию стратегии, ключа и запроса.
     */

    PG_RETURN_FLOAT8(retval);
}
```

Функции `distance` передаются те же аргументы, что и функции `consistent`.

При определении дистанции допускается некоторая неточность, если результат никогда не будет превышать действительную дистанцию до элемента. Так, например, в геометрических приложениях бывает достаточно определить дистанцию до описанного прямоугольника. Для внутреннего узла дерева результат не должен превышать дистанцию до любого из его дочерних узлов. Если возвращаемая дистанция неточная, функция должна установить флаг `*recheck`. (Это необязательно для внутренних узлов дерева; для них результат всегда считается неточным.) В этом случае исполнитель вычислит точную дистанцию, выбрав кортеж из кучи, и переупорядочит кортежи при необходимости.

Если функция distance возвращает `*recheck = true` для любого узла на уровне листьев, типом результата исходного оператора упорядочивания должен быть `float8` или `float4`, и значения результата функции distance должны быть сравнимы с результатами исходного оператора упорядочивания, так как исполнитель будет выполнять сортировку, используя и результаты функции distance, и уточнённые результаты оператора упорядочивания. В противном случае значениями результата distance могут быть любые конечные значения `float8`, при условии, что относительный порядок значений результата соответствует порядку, который даёт оператор упорядочивания. (Значения бесконечность и минус бесконечность применяются внутри для особых случаев, например, представления NULL, поэтому возвращать такие значения из функций `distance` не рекомендуется.)

`fetch`

Преобразует сжатое представление элемента данных в индексе в исходный тип данных, для сканирования только индекса. Возвращаемые данные должны быть точной, не примерной копией изначально проиндексированного значения.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_fetch(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

В качестве аргумента ей передаётся указатель на структуру `GISTENTRY`. При вызове её поле `key` содержит данные листа в сжатой форме (не NULL). Возвращаемое значение — ещё одна структура `GISTENTRY`, в которой поле `key` содержит те же данные в исходной, развёрнутой форме. Если функция `compress` класса операторов не делает с данными листьев ничего, метод `fetch` может возвратить аргумент без изменений. Либо, если класс операторов не имеет функции `compress`, метод `fetch` тоже может быть опущен, так как он в любом случае не должен ничего делать.

Соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_fetch);

Datum
my_fetch(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    input_data_type *in = DatumGetPointer(entry->key);
    fetched_data_type *fetched_data;
    GISTENTRY  *retval;
```

```
    retval = palloc(sizeof(GISTENTRY));
    fetched_data = palloc(sizeof(fetched_data_type));

    /*
     * Преобразовать структуру 'fetched_data' в Datum исходного типа данных.
     */

    /* Заполнить *retval из fetch_data. */
    gistentryinit(*retval, PointerGetDatum(converted_datum),
                  entry->rel, entry->page, entry->offset, FALSE);

    PG_RETURN_POINTER(retval);
}
```

Если метод сжатия является неточным для записей уровня листьев, такой класс операторов не может поддерживать сканирование только индекса и не должен определять функцию `fetch`.

`options`

Позволяет определить видимые пользователю параметры, управляющие поведением класса операторов.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_options(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Этой функции передаётся указатель на структуру `local_relopts`, в которую нужно внести набор параметров, относящихся к классу операторов. Обращаться к этим параметрам из других опорных функций можно с помощью макросов `PG_HAS_OPCLASS_OPTIONS()` и `PG_GET_OPCLASS_OPTIONS()`.

Ниже показан пример реализации функции my_options() и использования параметров из других опорных функций:

```
typedef enum MyEnumType
{
    MY_ENUM_ON,
    MY_ENUM_OFF,
    MY_ENUM_AUTO
} MyEnumType;

typedef struct
{
    int32   vl_len_;    /* заголовок varlena (не меняйте его напрямую!) */
    int     int_param;  /* целочисленный параметр */
    double  real_param; /* параметр с плавающей точкой */
    MyEnumType enum_param; /* параметр-перечисление */
    int     str_param;  /* строковый параметр */
} MyOptionsStruct;

/* Строковое представление значений в перечислении */
static relopt_enum_elt_def myEnumValues[] =
{
    {"on", MY_ENUM_ON},
    {"off", MY_ENUM_OFF},
    {"auto", MY_ENUM_AUTO},
    {(const char *) NULL}   /* завершающий элемент списка */
};
```

```
static char *str_param_default = "default";

/*
 * Пример проверочной функции: проверяет, что строка не длиннее 8 байт.
 */
static void
validate_my_string_relopt(const char *value)
{
    if (strlen(value) > 8)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
                 errmsg("str_param must be at most 8 bytes")));
}

/*
 * Пример функции-заполнителя: переводит символы в нижний регистр.
 */
static Size
fill_my_string_relopt(const char *value, void *ptr)
{
    char   *tmp = str_tolower(value, strlen(value), DEFAULT_COLLATION_OID);
    int     len = strlen(tmp);

    if (ptr)
        strcpy((char *) ptr, tmp);

    pfree(tmp);
    return len + 1;
}

PG_FUNCTION_INFO_V1(my_options);

Datum
my_options(PG_FUNCTION_ARGS)
{
    local_relopts *relopts = (local_relopts *) PG_GETARG_POINTER(0);

    init_local_reloptions(relopts, sizeof(MyOptionsStruct));
    add_local_int_reloption(relopts, "int_param", "integer parameter",
                            100, 0, 1000000,
                            offsetof(MyOptionsStruct, int_param));
    add_local_real_reloption(relopts, "real_param", "real parameter",
                             1.0, 0.0, 1000000.0,
                             offsetof(MyOptionsStruct, real_param));
    add_local_enum_reloption(relopts, "enum_param", "enum parameter",
                             myEnumValues, MY_ENUM_ON,
                             "Valid values are: \"on\", \"off\" and \"auto\".",
                             offsetof(MyOptionsStruct, enum_param));
    add_local_string_reloption(relopts, "str_param", "string parameter",
                               str_param_default,
                               &validate_my_string_relopt,
                               &fill_my_string_relopt,
                               offsetof(MyOptionsStruct, str_param));

    PG_RETURN_VOID();
}
```

```
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    int     int_param = 100;
    double  real_param = 1.0;
    MyEnumType enum_param = MY_ENUM_ON;
    char   *str_param = str_param_default;

    /*
     * Обычно когда в классе операторов определён метод 'options', полученные через
 него
     * параметры всегда передаются опорным функциям.  Однако, если вы добавите метод
 'options' в
     * существующий класс операторов, в ранее созданных индексах параметров не
 будет, поэтому
     * необходима следующая проверка.
     */
    if (PG_HAS_OPCLASS_OPTIONS())
    {
        MyOptionsStruct *options = (MyOptionsStruct *) PG_GET_OPCLASS_OPTIONS();

        int_param = options->int_param;
        real_param = options->real_param;
        enum_param = options->enum_param;
        str_param = GET_STRING_RELOPTION(options, str_param);
    }

    /* продолжение реализации опорной функции */
}
```

Так как в GiST представление ключа допускает гибкость, могут быть полезны параметры для настройки этого индекса. Например, можно задать длину ключа сигнатуры. В качестве примера рассмотрите функцию `gtsvector_options()`.

`sortsupport`

Возвращает функцию-компаратор для сортировки данных с сохранением локальности. Она используется командами `CREATE INDEX` и `REINDEX`. Качество созданного индекса зависит от того, насколько хорошо порядок сортировки, определённый функцией-компаратором, сохраняет локальность вводимых данных.

Метод `sortsupport` не является обязательным. Если он не предоставляется, `CREATE INDEX` создаёт индекс, вставляя каждый кортеж в дерево с помощью функций `penalty` и `picksplit`, но этот способ гораздо медленнее.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_sortsupport(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

В аргументе передаётся указатель на структуру `SortSupport`. Как минимум, данная функция должна заполнить в ней поле `comparator`. Компаратору передаются три аргумента: два элемента Datum для сравнения и указатель на структуру `SortSupport`. В данном случае в Datum передаются индексированные значения в том формате, в котором они хранятся в индексе; то есть в формате, возвращаемом методом `compress`. Полный API определён в файле *src/include/utils/sortsupport.h*.

Соответствующий код в модуле C должен реализовываться по такому шаблону:

```
PG_FUNCTION_INFO_V1(my_sortsupport);

static int
my_fastcmp(Datum x, Datum y, SortSupport ssup)
{
  /* establish order between x and y by computing some sorting value z */

  int z1 = ComputeSpatialCode(x);
  int z2 = ComputeSpatialCode(y);

  return z1 == z2 ? 0 : z1 > z2 ? 1 : -1;
}

Datum
my_sortsupport(PG_FUNCTION_ARGS)
{
  SortSupport ssup = (SortSupport) PG_GETARG_POINTER(0);

  ssup->comparator = my_fastcmp;
  PG_RETURN_VOID();
}
```

Все опорные методы GiST обычно вызываются в кратковременных контекстах памяти; то есть, `CurrentMemoryContext` сбрасывается после обработки каждого кортежа. Таким образом можно не заботиться об освобождении любых блоков памяти, выделенных функцией palloc. Однако в некоторых случаях для опорного метода полезно кешировать какие-либо данные между вызовами. Для этого нужно разместить долгоживущие данные в контексте `fcinfo->flinfo->fn_mcxt` и сохранить указатель на них в `fcinfo->flinfo->fn_extra`. Такие данные смогут просуществовать всё время операции с индексом (например, одно сканирование индекса GiST, построение индекса или добавление кортежа в индекс). Не забудьте вызвать pfree для предыдущего значения, заменяя значение в `fn_extra`, чтобы не допустить накопления утечек памяти в ходе операции.

62.2.4. Реализация

62.2.4.1. Способы построения индексов GiST

Самый простой способ построить индекс GiST — просто добавлять все записи одну за другой. Но для больших индексов это может занять много времени, поскольку если индексные кортежи разбросаны по всему индексу и он настолько велик, что не помещается в кеше, потребуется выполнить множество неупорядоченных операций ввода-вывода. Postgres Pro поддерживает два альтернативных метода начального построения индекса GiST: *с сортировкой* и *с буферизацией*.

Построение индекса с сортировкой возможно, только если все классы операторов, используемые индексом, предоставляют функцию `sortsupport`, как описано в Подраздел 62.2.3. В таком случае данный метод используется по умолчанию, как оптимальный.

При построении индекса с буферизацией кортежи не вставляются в индекс немедленно. Это позволяет кардинально сократить число операций произвольного доступа, требующихся при обработке неупорядоченных наборов данных. Для хорошо упорядоченных наборов выигрыш может быть минимальным или вообще отсутствовать, так как всего несколько страниц будут принимать новые кортежи в один момент времени и эти страницы будут умещаться в кеше, даже если весь индекс очень большой.

При построении индекса с буферизацией приходится гораздо чаще вызывать функцию `penalty` по сравнению с обычным построением, на что уходят дополнительные ресурсы процессора. Кроме того, буферам требуется временное место на диске, вплоть до размера результирующего индекса. Буферизация также может повлиять на качество результирующего индекса, как в положительную, так и в отрицательную сторону. Это влияние зависит от различных факторов, например, от распределения поступающих данных и от реализации класса операторов.

Если сортировка невозможна, по умолчанию при построении индекса GiST включается буферизация, когда размер индекса достигает значения effective_cache_size. Буферизацию можно принудительно включить или отключить вручную с помощью параметра `buffering` команды CREATE INDEX. Поведение по умолчанию достаточно эффективно в большинстве случаев, но если входные данные упорядочены, выключив буферизацию, можно получить некоторое ускорение.

62.2.5. Примеры

В настоящее время ядро системы Postgres Pro обеспечивает поддержку текстового поиска (индексацию типов `tsvector` и `tsquery`), а также функциональность R-дерева для некоторых встроенных геометрических типов данных. Классы операторов GiST содержатся также и в следующих дополнительных модулях (contrib):

`btree_gist`	Функциональность B-дерева для различных типов данных
`cube`	Индексирование для многомерных кубов
`hstore`	Модуль для хранения пар (ключ, значение)
`intarray`	RD-дерево для одномерных массивов значений int4
`ltree`	Индексирование древовидных структур
`pg_trgm`	Схожесть текста на основе статистики триграмм
`seg`	Индексирование «диапазонов чисел с плавающей точкой»

62.3. Индексы SP-GiST

62.3.1. Введение

Аббревиатура SP-GiST расшифровывается как «Space-Partitioned GiST» (GiST с секционированием пространства). SP-GiST поддерживает деревья поиска с секционированием, что облегчает разработку широкого спектра различных несбалансированных структур данных, в том числе деревьев квадрантов, а также k-мерных и префиксных деревьев. Общей характеристикой этих структур является то, что они последовательно разбивают пространство поиска на сегменты, которые не обязательно должны быть равного размера. При этом поиск, хорошо соответствующий правилу секционирования, с таким индексом может быть очень быстрым.

Эти популярные структуры данных изначально конструировались для работы в памяти. При таком применении они обычно представляются в виде набора динамически выделяемых узлов, связываемых указателями. Однако подобную схему нельзя в таком виде перенести на диск, так как цепочки указателей могут быть довольно длинными, и поэтому потребуется слишком много обращений к диску. Структуры данных для хранения на диске, напротив, должны иметь большую разветвлённость для минимизации объёма ввода-вывода. Для решения этой задачи SP-GiST сопоставляет узлы дерева поиска со страницами на диске так, чтобы при поиске требовалось обращаться только к нескольким страницам на диске, даже если при этом нужно просмотреть множество узлов.

Как и GiST, SP-GiST призван дать возможность разрабатывать дополнительные типы данных с соответствующими методами доступа экспертам в предметной области типа данных, а не специалистам по СУБД.

Представленная здесь информация частично позаимствована с *сайта* Проекта индексации SP-GiST Университета Пердью. Сопровождением реализации SP-GiST в Postgres Pro в основном занимаются Фёдор Сигаев и Олег Бартунов; дополнительные сведения можно получить на их *сайте*.

62.3.2. Встроенные классы операторов

В базовый дистрибутив Postgres Pro включены классы операторов SP-GiST, перечисленные в Таблице 62.2.

Таблица 62.2. Встроенные классы операторов SP-GiST

Имя	Индексируемые операторы	Операторы упорядочивания
`box_ops`	`<< (box,box)`	`<-> (box,point)`
	`&< (box,box)`	
	`&> (box,box)`	
	`>> (box,box)`	
	`<@ (box,box)`	
	`@> (box,box)`	
	`~= (box,box)`	
	`&& (box,box)`	
	`<<| (box,box)`	
	`&<| (box,box)`	
	`|&> (box,box)`	
	`|>> (box,box)`	
`inet_ops`	`<< (inet,inet)`	
	`<<= (inet,inet)`	
	`>> (inet,inet)`	
	`>>= (inet,inet)`	
	`= (inet,inet)`	
	`<> (inet,inet)`	
	`< (inet,inet)`	
	`<= (inet,inet)`	
	`> (inet,inet)`	
	`>= (inet,inet)`	
	`&& (inet,inet)`	
`kd_point_ops`	`|>> (point,point)`	`<-> (point,point)`
	`<< (point,point)`	
	`>> (point,point)`	
	`<<| (point,point)`	
	`~= (point,point)`	
	`<@ (point,box)`	
`poly_ops`	`<< (polygon,polygon)`	`<-> (polygon,point)`

Имя	Индексируемые операторы	Операторы упорядочивания
	`&< (polygon,polygon)`	
	`&> (polygon,polygon)`	
	`>> (polygon,polygon)`	
	`<@ (polygon,polygon)`	
	`@> (polygon,polygon)`	
	`~= (polygon,polygon)`	
	`&& (polygon,polygon)`	
	`<<| (polygon,polygon)`	
	`&<| (polygon,polygon)`	
	`|>> (polygon,polygon)`	
	`|&> (polygon,polygon)`	
`quad_point_ops`	`|>> (point,point)`	`<-> (point,point)`
	`<< (point,point)`	
	`>> (point,point)`	
	`<<| (point,point)`	
	`~= (point,point)`	
	`<@ (point,box)`	
`range_ops`	`= (anyrange,anyrange)`	
	`&& (anyrange,anyrange)`	
	`@> (anyrange,anyelement)`	
	`@> (anyrange,anyrange)`	
	`<@ (anyrange,anyrange)`	
	`<< (anyrange,anyrange)`	
	`>> (anyrange,anyrange)`	
	`&< (anyrange,anyrange)`	
	`&> (anyrange,anyrange)`	
	`-|- (anyrange,anyrange)`	
`text_ops`	`= (text,text)`	
	`< (text,text)`	
	`<= (text,text)`	
	`> (text,text)`	
	`>= (text,text)`	
	`~<~ (text,text)`	
	`~<=~ (text,text)`	
	`~>=~ (text,text)`	
	`~>~ (text,text)`	
	`^@ (text,text)`	

Из двух классов операторов для типа `point` классом по умолчанию является `quad_point_ops`. Класс `kd_point_ops` поддерживает те же операторы, но использует другую структуру данных индекса, которая может дать выигрыш в скорости для некоторых приложений.

Классы операторов `quad_point_ops`, `kd_point_ops` и `poly_ops` поддерживают оператор упорядочивания `<->`, позволяющий выполнить поиск k ближайших соседей (`k-NN`) по индексированному набору точек или многоугольников.

62.3.3. Расширяемость

SP-GiST предлагает интерфейс с высоким уровнем абстракции и таким образом требует от разработчика метода доступа реализовать только методы, специфичные для конкретного типа данных. Ядро SP-GiST отвечает за эффективную схему обращений к диску и поиск в структуре дерева, а также берёт на себя заботу о параллельном доступе и поддержке журнала.

Кортежи в листьях дерева SP-GiST обычно содержат значения того же типа данных, что и индексируемый столбец, но могут содержать и неточное представление индексируемого столбца. На верхнем уровне эти кортежи содержат исходное индексируемое значение данных, но на более нижних могут содержать только часть значения, например, суффикс. В этом случае опорные функции класса операторов должны уметь восстанавливать исходное значение, собирая его из внутренних кортежей, которые нужно пройти для достижения уровня конкретного листа.

Когда создаётся индекс SP-GiST с неключевыми столбцами (`INCLUDE`), значения этих столбцов также будут храниться в кортежах уровня листьев. Класс операторов SP-GiST не обращает внимания на неключевые столбцы, поэтому здесь они рассматриваться не будут.

Внутренние кортежи устроены сложнее, так как они представляют собой точки разветвления в дереве поиска. Каждый внутренний кортеж содержит набор из одного или нескольких *узлов*, представляющих группы сходных значений листьев. Узел содержит ответвление, приводящее либо к другому, внутреннему кортежу нижнего уровня, либо к короткому списку кортежей в листьях, лежащих в одной странице индекса. Для каждого узла обычно задаётся *метка*, описывающая его; например, в префиксном дереве меткой может быть очередной символ в строковом значении. (С другой стороны, класс операторов может опускать метки узлов, если он имеет дело с фиксированным набором узлов во всех внутренних кортежах; см. Подраздел 62.3.4.2.) Дополнительно внутренний кортеж может хранить *префикс*, описывающий все его члены. В префиксном дереве это может быть общий префикс всех представленных ниже строк. Значением префикса не обязательно должен быть префикс, а могут быть любые данные, требующиеся классу операторов; например, в дереве квадрантов это может быть центральная точка, от которой отмеряются четыре квадранта. В этом случае внутренний кортеж дерева квадрантов будет также содержать четыре узла, соответствующие квадрантам вокруг этой центральной точки.

Некоторые алгоритмы деревьев требует знания уровня (или глубины) текущего кортежа, так что ядро SP-GiST даёт возможность классам операторов контролировать число уровней при спуске по дереву. Также имеется поддержка пошагового восстановления представленного значения, когда это требуется, и передачи вниз дополнительных данных (так называемых *переходящих значений*) при спуске.

Примечание

Ядро SP-GiST берёт на себя заботу о значениях NULL. Хотя в индексах SP-GiST не хранятся записи для NULL в индексируемых столбцах, это скрыто от кода класса операторов; записи индексов или условия поиска с NULL никогда не передаются методам класса операторов. (Предполагается, что операторы SP-GiST строгие и не могут возвращать положительный результат для значений NULL.) Поэтому значения NULL здесь больше обсуждаться не будут.

Класс операторов индекса для SP-GiST должен предоставить пять методов и может дополнительно предоставить ещё два. Все пять обязательных методов должны по единому соглашению принимать два аргумента `internal`, первым из которых будет указатель на структуру C, содержащую входные значения для опорного метода, а вторым — указатель на структуру C, в которую должны помещаться выходные значения. Четыре из этих методов должны возвращать просто `void`, так как их результаты помещаются в выходную структуру; однако `leaf_consistent` возвращает результат `boolean`. Эти методы не должны менять никакие поля в их входных структурах. Выходная структура всегда обнуляется перед вызовом пользовательского метода. Необязательный шестой метод `compress` принимает в единственном аргументе значение `datum`, подлежащее индексированию, и возвращает значение, подходящее для физического хранения в кортеже уровня листьев. Необязательный седьмой метод `options` принимает указатель типа `internal` на структуру C, в которую должны помещаться параметры для класса операторов, и возвращает `void`.

Пользователь должен определить следующие пять обязательных методов:

`config`

Возвращает статическую информацию о реализации индекса, включая OID типов данных префикса и метки узла.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_config(internal, internal) RETURNS void ...
```

В первом аргументе передаётся указатель на структуру `spgConfigIn` языка C, содержащие входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgConfigOut` языка C, в которую функция должна поместить результат.

```
typedef struct spgConfigIn
{
    Oid         attType;        /* Индексируемый тип данных */
} spgConfigIn;

typedef struct spgConfigOut
{
    Oid         prefixType;     /* Тип данных префикса во внутренних кортежах */
    Oid         labelType;      /* Тип данных метки узла во внутренних кортежах */
    Oid         leafType;       /* Тип данных в кортежах уровня листьев */
    bool        canReturnData;  /* Класс операторов может восстановить исходные
 данные */
    bool        longValuesOK;   /* Класс может принимать значения, не умещающиеся на
 одной странице */
} spgConfigOut;
```

Поле `attType` передаётся для поддержки полиморфных классов операторов; для обычных классов операторов с фиксированным типом оно будет всегда содержать одно значение и поэтому его можно просто игнорировать.

Для классов операторов, не использующих префиксы, в `prefixType` можно установить `VOIDOID`. Подобным образом, для классов операторов, не использующих метки узлов, в `labelType` тоже можно установить `VOIDOID`. Признак `canReturnData` следует установить, если класс операторов может восстановить изначально переданное в индекс значение. Признак `longValuesOK` должен устанавливаться, только если `attType` переменной длины и класс операторов может фрагментировать длинные значения, повторяя суффиксы (см. Подраздел 62.3.4.1).

Значение `leafType` должно соответствовать типу хранения индекса, заданному в поле `opckeytype` записи, которая описывает этот класс операторов в каталоге. (Обратите внимание, что `opckeytype` может быть нулевым, тогда тип хранения определяется входным типом класса операторов, как бывает чаще всего.) Для обеспечения обратной совместимости методу `config` разрешается задать `leafType` другое значение (которое и будет использоваться); в результате содержимое индекса нельзя будет правильно идентифицировать в каталогах, поэтому данная возможность считается устаревшей. Кроме того, допускается оставить `leafType` неинициализированным (нулевым), что будет означать, что тип хранения индекса определяется значением `opckeytype`.

Когда `attType` и `leafType` различаются, должен предоставляться необязательный метод `compress`. Метод `compress` отвечает за преобразование данных, подлежащих индексации, из типа `attType` в тип `leafType`.

`choose`

Выбирает метод для добавления нового значения во внутренний кортеж.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_choose(internal, internal) RETURNS void ...
```

В первом аргументе передаётся указатель на структуру `spgChooseIn` языка C, содержащую входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgChooseOut`, в которую функция должна поместить результат.

```
typedef struct spgChooseIn
{
    Datum       datum;          /* исходное значение, которое должно индексироваться
 */
    Datum       leafDatum;      /* текущее значение, которое должно сохраниться в
 листе */
    int         level;          /* текущий уровень (начиная с нуля) */

    /* Данные из текущего внутреннего кортежа */
    bool        allTheSame;     /* кортеж с признаком все-равны? */
    bool        hasPrefix;      /* у кортежа есть префикс? */
    Datum       prefixDatum;    /* если да, то это значение префикса */
    int         nNodes;         /* число узлов во внутреннем кортеже */
    Datum      *nodeLabels;     /* значения меток узлов (NULL, если их нет) */
} spgChooseIn;

typedef enum spgChooseResultType
{
    spgMatchNode = 1,           /* спуститься в существующий узел */
    spgAddNode,                 /* добавить узел во внутренний кортеж */
    spgSplitTuple               /* разделить внутренний кортеж (изменить его
 префикс) */
} spgChooseResultType;

typedef struct spgChooseOut
{
    spgChooseResultType resultType;     /* код действия, см. выше */
    union
    {
        struct                  /* результаты для spgMatchNode */
        {
            int         nodeN;      /* спуститься к этому узлу (нумерация с 0) */
            int         levelAdd;   /* шаг увеличения уровня */
            Datum       restDatum;  /* новое значение листа */
        }           matchNode;
        struct                  /* результаты для spgAddNode */
        {
            Datum       nodeLabel;  /* метка нового узла */
            int         nodeN;      /* куда вставлять её (нумерация с 0) */
        }           addNode;
        struct                  /* результаты для spgSplitTuple */
        {
            /* Информация для формирования нового внутреннего кортежа верхнего
 уровня с одним дочерним кортежем */
            bool        prefixHasPrefix;    /* кортеж должен иметь префикс? */
            Datum       prefixPrefixDatum;  /* если да, его значение */
            int         prefixNNodes;       /* число узлов */
            Datum      *prefixNodeLabels;   /* их метки (или NULL, если
```

```
                                             * меток нет) */
            int         childNodeN;         /* узел, который получит дочерний кортеж
 */

            /* Информация для формирования нового внутреннего кортежа нижнего уровня
 со всеми старыми узлами */
            bool        postfixHasPrefix;   /* кортеж должен иметь префикс? */
            Datum       postfixPrefixDatum; /* если да, его значение */
        }           splitTuple;
    }           result;
} spgChooseOut;
```

В `datum` передаётся исходное значение типа `spgConfigIn.attType`, которое должно быть вставлено в индекс. В `leafDatum` содержится значение типа `spgConfigOut.leafType`, изначально представляющее собой результат метода `compress`, применённого к `datum`, если метод `compress` реализован, а иначе — собственно значение `datum`. `leafDatum` может быть другим на нижних уровнях дерева, если его изменят функции `choose` или `picksplit`. Когда поиск места добавления достигает страницы уровня листа, в создаваемом кортеже листа сохраняется текущее значение `leafDatum`. В `level` задаётся текущий уровень внутреннего кортежа, начиная с нуля для уровня корня. Признак `allTheSame` устанавливается, если текущий внутренний кортеж содержит несколько равнозначных узлов (см. Подраздел 62.3.4.3). Признак `hasPrefix` устанавливается, если текущий внутренний кортеж содержит префикс; в этом случае в `prefixDatum` задаётся его значение. Поле `nNodes` задаёт число дочерних узлов, содержащихся во внутреннем кортеже, а `nodeLabels` представляет массив их меток или NULL, если меток у них нет.

Функция `choose` может определить, соответствует ли новое значение одному из существующих дочерних узлов, или что нужно добавить новый дочерний узел, или что новое значение не согласуется с префиксом кортежа и внутренний кортеж нужно разделить, чтобы получить менее ограничивающий префикс.

Если новое значение соответствует одному из существующих дочерних узлов, установите в `resultType` значение `spgMatchNode`. Установите в `nodeN` номер этого узла в массиве узлов (нумерация начинается с нуля). Установите в `levelAdd` значение, на которое должен увеличиваться уровень (`level`) при спуске через этот узел, либо оставьте его нулевым, если класс операторов не отслеживает уровни. Установите `restDatum`, равным `leafDatum`, если класс операторов не меняет значения данных от уровня к уровню, а в противном случае запишите в него изменённое значение, которое должно использоваться в качестве `leafDatum` на следующем уровне.

Если нужно добавить новый дочерний узел, установите в `resultType` значение `spgAddNode`. В `nodeLabel` задайте метку для нового узла, а в `nodeN` позицию (отсчитываемую от нуля), в которую должен вставляться узел в массиве узлов. После того как узел будет добавлен, функция `choose` вызывается снова с изменённым внутренним кортежем; в результате этого вызова должен быть получен результат `spgMatchNode`.

Если новое значение не согласуется с префиксом кортежа, установите в `resultType` значение `spgSplitTuple`. Это действие приводит к перемещению всех существующих узлов в новый внутренний кортеж нижнего уровня и замене существующего внутреннего кортежа кортежем, содержащим одну ссылку вниз на новый внутренний кортеж. Установите признак `prefixHasPrefix`, чтобы указать, должен ли новый верхний кортеж иметь префикс, и если да, задайте в `prefixPrefixDatum` значение префикса. Это новое значение префикса должно быть в достаточной мере менее ограничивающим, чем исходное, чтобы в индекс было принято новое значение. Запишите в `prefixNNodes` число требующихся узлов в новом кортеже, а в `prefixNodeLabels` — указатель на выделенный через palloc массив с их метками или NULL, если метки узлов не нужны. Заметьте, что общий размер нового кортежа верхнего уровня не должен превышать общий размер кортежа, который он замещает; это ограничивает длины нового префикса и новых меток. Установите в `childNodeN` индекс (начиная с нуля) узла, который будет ссылаться на новый внутренний кортеж нижнего уровня. Установите признак `postfixHasPrefix`, чтобы указать, должен ли новый внутренний кортеж нижнего уровня иметь префикс, и если да, задайте в `postfixPrefixDatum` значение префикса. Сочетание этих двух префиксов и метки узла, ссылающегося вниз, (если она есть) должно иметь то же значение, что и исходный префикс, так как нет возможности ни изменить метки узлов, перемещённых в новый кортеж нижнего уровня, ни изменить какие-либо нижние записи индекса. После того как узел разделён, функция `choose` будет вызвана снова с заменяемым внутренним кортежем. При этом вызове может быть возвращён результат `spgAddNode`, если подходящий узел не был создан действием `spgSplitTuple`. В конце концов `choose` должна вернуть `spgMatchNode`, чтобы операция добавления могла перейти на следующий уровень.

`picksplit`

Выбирает, как создать новый внутренний кортеж по набору кортежей в листьях.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_picksplit(internal, internal) RETURNS void ...
```

В первом аргументе передаётся указатель на структуру `spgPickSplitIn` языка C, содержащую входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgPickSplitOut` языка C, в которую функция должна поместить результат.

```
typedef struct spgPickSplitIn
{
    int         nTuples;        /* число кортежей в листьях */
    Datum      *datums;         /* их значения (массив длины nTuples) */
    int         level;          /* текущий уровень (отсчитывая от 0) */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{
    bool        hasPrefix;      /* новый внутренний кортеж должен иметь префикс? */
    Datum       prefixDatum;    /* если да, его значение */

    int         nNodes;         /* число узлов для нового внутреннего кортежа */
    Datum      *nodeLabels;     /* их метки (или NULL, если их нет) */

    int        *mapTuplesToNodes;   /* номер узла для каждого кортежа в листе */
    Datum      *leafTupleDatums;    /* значения, помещаемые в каждый новый кортеж */
} spgPickSplitOut;
```

В `nTuples` задаётся число предоставленных кортежей уровня листьев, а в `datums` — массив их значений типа `spgConfigOut.leafType`. В `level` указывается текущий уровень, который должны разделять все кортежи листьев, и который станет уровнем нового внутреннего кортежа.

Установите признак `hasPrefix`, чтобы указать, должен ли новый внутренний кортеж иметь префикс, и если да, задайте в `prefixDatum` значение префикса. Установите в `nNodes` количество узлов, которые будут содержаться во внутреннем кортеже, а в `nodeLabels` — массив значений их меток либо NULL, если узлам не нужны метки. Поместите в `mapTuplesToNodes` указатель на массив, назначающий номера узлов (начиная с нуля) каждому кортежу листа. В `leafTupleDatums` передайте массив значений, которые должны быть сохранены в новых кортежах листьев (они будут совпадать со входными значениями (`datums`), если класс операторов не изменяет значения от уровня к следующему). Заметьте, что функция `picksplit` сама должна выделить память, используя palloc, для массивов `nodeLabels`, `mapTuplesToNodes` и `leafTupleDatums`.

Если передаётся несколько кортежей листьев, ожидается, что функция `picksplit` классифицирует их и разделит на несколько узлов; иначе нельзя будет разнести кортежи листьев по разным страницам, что является конечной целью этой операции. Таким образом, если `picksplit` в итоге помещает все кортежи листьев в один узел, ядро SP-GiST меняет это решение и создаёт внутренний кортеж, в котором кортежи листьев связываются случайным образом с несколькими узлами с одинаковыми метками. Такой кортеж помечается флагом `allTheSame`, показывающим, что все узлы равны. Функции `choose` и `inner_consistent` должны работать с такими внутренними кортежами особым образом. За дополнительными сведениями обратитесь к Подразделу 62.3.4.3.

`picksplit` может применяться к одному кортежу на уровне листьев, только когда функция `config` установила в `longValuesOK` значение true и было передано входное значение, большее страницы. В этом случае цель операции — отделить префикс и получить новое, более короткое значение для листа. Этот вызов будет повторяться, пока значение уровня листа не уменьшится настолько, чтобы уместиться в странице. За дополнительными сведениями обратитесь к Подразделу 62.3.4.1.

`inner_consistent`

Возвращает набор узлов (ветвей), по которым надо продолжать поиск.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_inner_consistent(internal, internal) RETURNS void ...
```

В первом аргументе передаётся указатель на структуру `spgInnerConsistentIn` языка C, содержащую входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgInnerConsistentOut` языка C, в которую функция должна поместить результат.

```
typedef struct spgInnerConsistentIn
{
    ScanKey     scankeys;       /* массив операторов и искомых значений */
    ScanKey     orderbys;       /* массив операторов упорядочивания и
                                 * сравниваемых значений */
    int         nkeys;          /* длина массива scankeys */
    int         norderbys;      /* длина массива orderbys */

    Datum       reconstructedValue;     /* значение, восстановленное для родителя */
    void       *traversalValue; /* переходящее значение, специфичное для класса
 операторов */
    MemoryContext traversalMemoryContext;   /* переходящие значения нужно помещать
 сюда */
    int         level;          /* текущий уровень (отсчитывается от нуля) */
    bool        returnData;     /* нужно ли возвращать исходные данные? */

    /* Данные из текущего внутреннего кортежа */
    bool        allTheSame;     /* кортеж с признаком все-равны? */
    bool        hasPrefix;      /* у кортежа есть префикс? */
    Datum       prefixDatum;    /* если да, то это значение префикса */
    int         nNodes;         /* число узлов во внутреннем кортеже */
    Datum      *nodeLabels;     /* значения меток узлов (NULL, если их нет) */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
{
    int         nNodes;         /* число дочерних узлов, которые нужно посетить */
    int        *nodeNumbers;    /* их номера в массиве узлов */
    int        *levelAdds;      /* шаги увеличения уровня для этих узлов */
    Datum      *reconstructedValues;    /* связанные восстановленные значения */
    void      **traversalValues;        /* переходящие значения, специфичные для
 класса операторов */
    double    **distances;              /* связанные расстояния */
} spgInnerConsistentOut;
```

Массив `scankeys` длины `nkeys` описывает условия поиска по индексу. Эти условия объединяются операцией И — найдены должны быть только те записи, которые удовлетворяют всем условиям. (Заметьте, что с `nkeys` = 0 подразумевается, что запросу удовлетворяют все записи в индексе.) Обычно эту функцию интересуют только поля `sk_strategy` и `sk_argument` в каждой записи массива, в которых определяется соответственно индексируемый оператор и искомое значение. В частности, нет необходимости проверять `sk_flags`, чтобы распознать NULL в искомом значении, так как ядро SP-GiST отфильтрует такие условия. Массив `orderbys` длины `norderbys` подобным образом описывает упорядочивающие операторы (если они есть). В `reconstructedValue` передаётся значение, восстановленное для родительского кортежа; это может быть `(Datum) 0` на уровне корня или если функция `inner_consistent` не установила значение на предыдущем уровне. В `traversalValue` передаётся указатель на переходящие данные, полученные из предыдущего вызова `inner_consistent` для родительского кортежа индекса, либо NULL на уровне корня. Поле `traversalMemoryContext` указывает на контекст памяти, в котором нужно сохранить выходные переходящие данные (см. ниже). В `level` передаётся уровень текущего внутреннего кортежа (уровень корня считается нулевым). Флаг `returnData` устанавливается, когда для этого запроса нужно получить восстановленные данные; это возможно, только если функция `config` установила признак `canReturnData`. Признак `allTheSame` устанавливается, если текущий внутренний кортеж имеет пометку «все-равны»; в этом случае все узлы имеют одну метку (если имеют) и значит, либо все они, либо никакой не соответствует запросу (см. Подраздел 62.3.4.3). Признак `hasPrefix` устанавливается, если текущий внутренний кортеж содержит префикс; в этом случае в `prefixDatum` находится его значение. В `nNodes` задаётся число дочерних узлов, содержащихся во внутреннем кортеже, а в `nodeLabels` — массив их меток либо NULL, если они не имеют меток.

В `nNodes` нужно записать число дочерних узлов, которые потребуется посетить при поиске, а в `nodeNumbers` — массив их индексов. Если класс операторов отслеживает уровни, в `levelAdds` нужно передать массив с шагами увеличения уровня при посещении каждого узла. (Часто шаг будет одним для всех узлов, но может быть и по-другому, поэтому применяется массив.) Если потребовалось восстановить значения, поместите в `reconstructedValues` массив значений, восстановленных для каждого дочернего узла, который нужно посетить; в противном случае оставьте `reconstructedValues` равным NULL. Предполагается, что восстановленные значения имеют тип `spgConfigOut.leafType`. (Однако ядро системы ничего не будет делать с ними, кроме как, возможно, копировать, поэтому они должны иметь такие же `typlen` и `typbyval`, что и `leafType`.) Если выполняется поиск с упорядочиванием, поместите в `distances` массив расстояний в соответствии с массивом `orderbys` (узлы с меньшими расстояниями будут обрабатываться первыми). В противном случае оставьте в этом поле NULL. Если желательно передать дополнительные данные («переходящие значения») на нижние уровни при поиске по дереву, поместите в `traversalValues` указатель на массив соответствующих переходящих значений, по одному для каждого дочернего узла, который нужно посетить; в противном случае оставьте в `traversalValues` значение NULL. Заметьте, что функция `inner_consistent` сама должна выделять память, используя palloc, для массивов `nodeNumbers`, `levelAdds`, `distances`, `reconstructedValues` и `traversalValues` в текущем контексте памяти. Однако выходные переходящие значения, на которые указывает массив `traversalValues`, должны размещаться в контексте `traversalMemoryContext`. При этом каждое переходящее значения должно располагаться в отдельном блоке памяти palloc.

`leaf_consistent`

Возвращает true, если кортеж листа удовлетворяет запросу.

В SQL эта функция должна объявляться так:

```
CREATE FUNCTION my_leaf_consistent(internal, internal) RETURNS bool ...
```

В первом аргументе передаётся указатель на структуру `spgLeafConsistentIn` языка C, содержащую входные данные для функции. Во втором аргументе передаётся указатель на структуру `spgLeafConsistentOut` языка C, в которую функция должна поместить результат.

```
typedef struct spgLeafConsistentIn
{
    ScanKey     scankeys;       /* массив операторов и искомых значений */
    ScanKey     orderbys;       /* массив операторов упорядочивания и
```

```
                                  * сравниваемых значений */
    int         nkeys;          /* длина массива scankeys */
    int         norderbys;      /* длина массива orderbys */

    Datum       reconstructedValue;     /* значение, восстановленное для родителя */
    void       *traversalValue; /* переходящее значение, специфичное для класса
 операторов */
    int         level;          /* текущий уровень (отсчитывая от нуля) */
    bool        returnData;     /* нужно ли возвращать исходные данные? */

    Datum       leafDatum;      /* значение в кортеже листа */
} spgLeafConsistentIn;

typedef struct spgLeafConsistentOut
{
    Datum       leafValue;        /* восстановленные исходные данные, при наличии */
    bool        recheck;          /* true, если оператор нужно перепроверить */
    bool        recheckDistances; /* true, если расстояния нужно перепроверить */
    double     *distances;        /* связанные расстояния */
} spgLeafConsistentOut;
```

Массив `scankeys` длины `nkeys` описывает условия поиска по индексу. Эти условия объединяются операцией И — запросу удовлетворяют только те записи в индексе, которые удовлетворяют всем этим условиям. (Заметьте, что с `nkeys` = 0 подразумевается, что запросу удовлетворяют все записи в индексе.) Обычно эту функцию интересуют только поля `sk_strategy` и `sk_argument` в каждой записи массива, в которых определяются соответственно индексируемый оператор и искомое значение. В частности, нет необходимости проверять `sk_flags`, чтобы распознать NULL в искомом значении, так как ядро SP-GiST отфильтрует такие условия. Массив `orderbys` длины `norderbys` подобным образом описывает упорядочивающие операторы. В `reconstructedValue` передаётся значение, восстановленное для родительского кортежа; это может быть `(Datum) 0` на уровне корня или если функция `inner_consistent` не установила значение на предыдущем уровне. В `traversalValue` передаётся указатель на переходящие данные, полученные из предыдущего вызова `inner_consistent` для родительского кортежа индекса, либо NULL на уровне корня. В `level` передаётся уровень текущего внутреннего кортежа (уровень корня считается нулевым). Флаг `returnData` устанавливается, когда для этого запроса нужно получить восстановленные данные; это возможно, только если функция `config` установила признак `canReturnData`. В `leafDatum` передаётся значение ключа, записанное в текущем кортеже листа.

Эта функция должна вернуть `true`, если кортеж листа соответствует запросу, или `false` в противном случае. В случае положительного результата, если в поле `returnData` передано `true`, нужно поместить в `leafValue` значение (типа `spgConfigIn.attType`), изначально переданное для индексации в этот кортеж. Кроме того, флагу `recheck` можно присвоить `true`, если соответствие неточное, так что для установления точного результата проверки нужно повторно применить оператор(ы) к собственно кортежу данных. Если выполняется упорядочивающий поиск, поместите в `distances` массив со значениями расстояния, соответствующими массиву `orderbys`. В противном случае оставьте в этом поле NULL. Если хотя бы одно из возвращаемых расстояний определено неточно, присвойте true полю `recheckDistances`. В этом случае исполнитель вычислит точные расстояния после получения кортежа из кучи и переупорядочит кортежи, если потребуется.

Дополнительно пользователь может определить методы:

`Datum compress(Datum in)`

Преобразует элемент данных в формат, подходящий для физического хранения в кортеже уровня листьев в индексе. Эта функция принимает значение типа `spgConfigIn.attType` и возвращает значение типа `spgConfigOut.leafType`. Возвращаемое значение не должно содержать указатель на внешние TOAST-данные.

Обратите внимание, что метод `compress` применяется только к сохраняемым значениям. Методы, оценивающие согласованность, получают сканируемые в запросе ключи (`scankeys`) неизменёнными, без обработки функцией `compress`.

`options`

Определяет набор видимых пользователю параметров, управляющих поведением класса операторов.

В SQL эта функция должна объявляться так:

```
CREATE OR REPLACE FUNCTION my_options(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Этой функции передаётся указатель на структуру `local_relopts`, в которую нужно внести набор параметров, относящихся к классу операторов. Обращаться к этим параметрам из других опорных функций можно с помощью макросов `PG_HAS_OPCLASS_OPTIONS()` и `PG_GET_OPCLASS_OPTIONS()`.

Так как в SP-GiST представление ключа допускает гибкость, могут быть полезны параметры для настройки этого индекса.

Все опорные методы SP-GiST обычно вызываются в кратковременных контекстах памяти; то есть `CurrentMemoryContext` сбрасывается после обработки каждого кортежа. Таким образом, можно не заботиться об освобождении любых блоков памяти, выделенных функцией palloc. (Метод `config` является исключением: в нём нужно не допускать утечек памяти. Но обычно метод `config` не делает ничего, кроме как присваивает константы переданной структуре параметров.)

Если индексируемый столбец имеет сортируемый тип данных, правило сортировки индекса будет передаваться всем опорным методам, используя стандартный механизм `PG_GET_COLLATION()`.

62.3.4. Реализация

В этом разделе освещаются тонкости реализации и особенности, о которых полезно знать тем, кто будет реализовывать классы операторов SP-GiST.

62.3.4.1. Ограничения SP-GiST

Отдельные кортежи листьев и внутренние кортежи должны умещаться в одной странице индекса (по умолчанию её размер 8 Кбайт). Таким образом при индексировании значений типов данных переменной длины большие значения могут поддерживаться только такими схемами, как префиксные деревья, в которых каждый уровень дерева включает префикс, достаточно короткий для помещения в страницу, и на конечном уровне листьев содержится суффикс, который также достаточно мал, чтобы поместиться в странице. Класс операторов должен устанавливать признак `longValuesOK`, только если он готов организовывать такую структуру. Если этот признак не установлен, ядро SP-GiST не примет запрос на индексацию значения, которое слишком велико для одной страницы индекса.

Также класс операторов должен отвечать за то, чтобы внутренние кортежи при расширении не выходили за пределы страницы индекса; это ограничивает число дочерних узлов, которые могут принадлежать одному внутреннему кортежу, а также максимальный размер значения префикса.

Ещё одно ограничение состоит в том, что когда узел внутреннего кортежа указывает на набор кортежей листьев, все эти кортежи должны находиться в одной странице индекса. (Это конструктивное ограничение введено для оптимизации позиционирования и экономии места на ссылках, связывающих такие кортежи вместе.) Если набор кортежей листьев оказывается слишком большим для одной страницы, выполняется разделение и вставляется промежуточный внутренний кортеж. Чтобы устранить возникшую проблему, новый внутренний кортеж *должен* разделять набор значений в листе на несколько групп узлов. Если функция `picksplit` класса операторов не может сделать это, ядро SP-GiST переходит к чрезвычайным мерам, описанным в Подразделе 62.3.4.3.

Когда `longValuesOK` имеет значение true, ожидается, что последующие уровни дерева SP-GiST будут включать всё больше и больше информации в префиксы и метки узлов внутренних кортежей, постепенно уменьшая значение, которое нужно сохранить на уровне листьев, чтобы оно в конце концов уместилось на странице. Чтобы в случае ошибки в классах операторах исключить бесконечные циклы при добавлении записи, ядро SP-GiST выдаст ошибку, если значение для листа не уменьшится за 10 вызовов метода `choose`.

62.3.4.2. SP-GiST без меток узлов

В некоторых древовидных схемах каждый внутренний кортеж содержит фиксированный набор узлов; например, в дереве квадрантов это всегда четыре узла, соответствующие четырём квадрантам вокруг центральной точки внутреннего кортежа. В таком случае код обычно работает с узлами по номерам и необходимости в явных метках узлов нет. Чтобы убрать метки узлов (и таким образом сэкономить место), функция `picksplit` может возвратить NULL вместо массива `nodeLabels`, а функция `choose` аналогично может возвратить NULL вместо массива `prefixNodeLabels` во время действия `spgSplitTuple`. В результате при последующих вызовах функций `choose` и `inner_consistent` им вместо `nodeLabels` будет передаваться NULL. В принципе метки узлов могут применяться для одних внутренних кортежей и отсутствовать у других в том же индексе.

Когда внутренний кортеж содержит узлы без меток, функция `choose` не может выбрать действие `spgAddNode`, так как в этом случае предполагается, что набор узлов фиксированный.

62.3.4.3. Внутренние кортежи «все-равны»

Ядро SP-GiST может переопределить результаты функции `picksplit` класса операторов, когда эта функция не может разделить поступившие значения листьев на минимум две категории узлов. Когда это происходит, создаётся новый внутренний кортеж с несколькими узлами, каждый из которых имеет одну метку (если имеет), которую `picksplit` дала одному узлу, а значения листьев распределяются случайно между этими равнозначными узлами. Для этого внутреннего кортежа устанавливается флаг `allTheSame`, который предупреждает функции `choose` и `inner_consistent`, что кортеж не содержит набор узлов, который они обычно ожидают.

Когда обрабатывается кортеж с флагом `allTheSame`, выбранное функцией `choose` действие `spgMatchNode` воспринимается как указание, что новое значение можно присвоить одному из равнозначных узлов; код ядра будет игнорировать полученное значение `nodeN` и спустится в один из узлов, выбранный случайно (чтобы дерево было сбалансированным). Будет считаться ошибкой, если `choose` выберет действие `spgAddNode`, так как при этом не все узлы окажутся равны; если добавляемое значение не соответствует существующим узлам, должно выбираться действие `spgSplitTuple`.

Также, когда обрабатывается кортеж с флагом `allTheSame`, функция `inner_consistent` должна вернуть все или не возвращать никакие узлы для продолжения поиска по индексу, так как все узлы равнозначны. Для этого может потребоваться, а может и не потребоваться код обработки особого случая, в зависимости от того, как `inner_consistent` обычно воспринимает узлы.

62.4. Индексы GIN

62.4.1. Введение

GIN расшифровывается как «Generalized Inverted Index» (Обобщённый инвертированный индекс). GIN предназначается для случаев, когда индексируемые значения являются составными, а запросы, на обработку которых рассчитан индекс, ищут значения элементов в этих составных объектах. Например, такими объектами могут быть документы, а запросы могут выполнять поиск документов, содержащих определённые слова.

Здесь мы используем термин *объект*, говоря о составном значении, которое индексируется, и термин *ключ*, говоря о включённом в него элементе. GIN всегда хранит и ищет ключи, а не объекты как таковые.

Индекс GIN сохраняет набор пар (ключ, список идентификаторов), где *список идентификаторов* содержит идентификаторы строк, в которых находится ключ. Один и тот же идентификатор строки может фигурировать в нескольких списках, так как объект может содержать больше одного ключа. Значение каждого ключа хранится только один раз, так что индекс GIN очень компактен в случаях, когда один ключ встречается много раз.

GIN является обобщённым в том смысле, что код метода доступа GIN не должен знать о конкретных операциях, которые он ускоряет. Вместо этого задаются специальные стратегии для конкретных типов данных. Стратегия определяет, как извлекаются ключи из индексируемых объектов и условий запросов, и как установить, действительно ли удовлетворяет запросу строка, содержащая некоторые значения ключей.

Ключевым преимуществом GIN является то, что он позволяет разрабатывать дополнительные типы данных с соответствующими методами доступа экспертам в предметной области типа данных, а не специалистам по СУБД. В этом аспекте он похож на GiST.

Сопровождением реализации GIN в Postgres Pro в основном занимаются Фёдор Сигаев и Олег Бартунов. Дополнительные сведения о GIN можно получить на их *сайте*.

62.4.2. Встроенные классы операторов

В базовый дистрибутив Postgres Pro включены классы операторов GIN, перечисленные в Таблице 62.3. (Некоторые дополнительные модули, описанные в Приложении F, добавляют другие классы операторов GIN.)

Таблица 62.3. Встроенные классы операторов GIN

Имя	Индексируемые операторы
`array_ops`	`&& (anyarray,anyarray)`
	`@> (anyarray,anyarray)`
	`<@ (anyarray,anyarray)`
	`= (anyarray,anyarray)`
`jsonb_ops`	`@> (jsonb,jsonb)`
	`@? (jsonb,jsonpath)`
	`@@ (jsonb,jsonpath)`
	`? (jsonb,text)`
	`?| (jsonb,text[])`
	`?& (jsonb,text[])`
`jsonb_path_ops`	`@> (jsonb,jsonb)`
	`@? (jsonb,jsonpath)`
	`@@ (jsonb,jsonpath)`
`tsvector_ops`	`@@ (tsvector,tsquery)`

Из двух классов операторов для типа `jsonb` классом по умолчанию является `jsonb_ops`. Класс `jsonb_path_ops` поддерживает меньше операторов, но обеспечивает для них большую производительность. За подробностями обратитесь к Подразделу 8.14.4.

62.4.3. Расширяемость

Интерфейс GIN характеризуется высоким уровнем абстракции и таким образом требует от разработчика метода доступа реализовать только смысловое наполнение обрабатываемого типа данных.

Уровень GIN берёт на себя заботу о параллельном доступе, поддержке журнала и поиске в структуре дерева.

Всё, что нужно, чтобы получить работающий метод доступа GIN — это реализовать несколько пользовательских методов, определяющих поведение ключей в дереве и отношения между ключами, индексируемыми объектами и поддерживаемыми запросами. Словом, GIN сочетает расширяемость с универсальностью, повторным использованием кода и аккуратным интерфейсом.

Класс операторов должен предоставить GIN следующие три метода:

`Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)`

Возвращает массив ключей (выделенный через palloc) для индексируемого объекта. Число возвращаемых ключей должно записываться в `*nkeys`. Если какой-либо из ключей может быть NULL, нужно так же выделить через palloc массив из `*nkeys` полей `bool`, записать его адрес в `*nullFlags` и установить эти флаги NULL как требуется. В `*nullFlags` можно оставить значение `NULL` (это начальное значение), если все ключи отличны от NULL. Эта функция может возвратить `NULL`, если объект не содержит ключей.

`Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer **extra_data, bool **nullFlags, int32 *searchMode)`

Возвращает массив ключей (выделенный через palloc) для запрашиваемого значения; то есть, в `query` поступает значение, находящееся по правую сторону индексируемого оператора, по левую сторону которого указан индексируемый столбец. Аргумент `n` задаёт номер стратегии оператора в классе операторов (см. Подраздел 36.16.2). Часто функция `extractQuery` должна проанализировать `n`, чтобы определить тип данных аргумента `query` и выбрать метод для извлечения значений ключей. Число возвращаемых ключей должно быть записано в `*nkeys`. Если какие-либо ключи могут быть NULL, нужно так же выделить через palloc массив из `*nkeys` полей `bool`, сохранить его адрес в `*nullFlags`, и установить эти флаги NULL как требуется. В `*nullFlags` можно оставить значение `NULL` (это начальное значение), если все ключи отличны от NULL. Эта функция может возвратить `NULL`, если `query` не содержит ключей.

Выходной аргумент `searchMode` позволяет функции `extractQuery` выбрать режим, в котором должен выполняться поиск. Если `*searchMode` имеет значение `GIN_SEARCH_MODE_DEFAULT` (это значение устанавливается перед вызовом), подходящими кандидатами считаются только те объекты, которые соответствуют минимум одному из возвращённых ключей. Если в `*searchMode` установлено значение `GIN_SEARCH_MODE_INCLUDE_EMPTY`, то в дополнение к объектам с минимум одним совпадением ключа, подходящими кандидатами будут считаться и объекты, вообще не содержащие ключей. (Этот режим полезен для реализации, например, операторов А-является-подмножеством-В.) Если в `*searchMode` установлено значение `GIN_SEARCH_MODE_ALL`, подходящими кандидатами считаются все отличные от NULL объекты в индексе, независимо от того, встречаются ли в них возвращаемые ключи. (Этот режим намного медленнее двух других, так как он по сути требует сканирования всего индекса, но он может быть необходим для корректной обработки крайних случаев. Оператор, который выбирает этот режим в большинстве ситуаций, скорее всего не подходит для реализации в классе операторов GIN.) Символы для этих значений режима определены в `access/gin.h`.

Выходной аргумент `pmatch` используется, когда поддерживается частичное соответствие. Чтобы использовать его, `extractQuery` должна выделить массив из `*nkeys` логических элементов и сохранить его адрес в `*pmatch`. Элемент этого массива должен содержать true, если соответствующий ключ требует частичного соответствия, и false в противном случае. Если переменная `*pmatch` содержит `NULL`, GIN полагает, что частичное соответствие не требуется. В эту переменную записывается `NULL` перед вызовом, так что этот аргумент можно просто игнорировать в классах операторов, не поддерживающих частичное соответствие.

Выходной аргумент `extra_data` позволяет функции `extractQuery` передать дополнительные данные методам `consistent` и `comparePartial`. Чтобы использовать его, `extractQuery` должна выделить массив из `*nkeys` указателей и сохранить его адрес в `*extra_data`, а затем сохранить всё, что ей требуется, в отдельных указателях. В эту переменную записывается `NULL` перед вызовом, поэтому данный аргумент может просто игнорироваться классами операторов, которым не нужны дополнительные данные. Если массив `*extra_data` задан, он целиком передаётся в метод `consistent`, а в `comparePartial` передаётся соответствующий его элемент.

Класс операторов должен также предоставить функцию для проверки, соответствует ли индексированный объект запросу. Поддерживаются две её вариации: булева `consistent` и троичная `triConsistent`. Функция `triConsistent` покрывает функциональность обоих, так что достаточно реализовать только её. Однако если вычисление булевой вариации оказывается значительно дешевле, может иметь смысл реализовать их обе. Если представлена только булева вариация, некоторые оптимизации, построенные на отбраковывании объектов до выборки всех ключей, отключаются.

`bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])`

Возвращает true, если индексированный объект удовлетворяет оператору запроса с номером стратегии `n` (или потенциально удовлетворяет, когда возвращается указание перепроверки). Эта функция не имеет прямого доступа к значению индексированного объекта, так как GIN не хранит сами объекты. Вместо этого, она знает о значениях ключей, извлечённых из запроса и встречающихся в данном индексированном объекте. Массив `check` имеет длину `nkeys`, что равняется числу ключей, ранее возвращённых функцией `extractQuery` для данного значения `query`. Элемент массива `check` равняется true, если индексированный объект содержит соответствующий ключ запроса; то есть, если (check[i] == true), то i-ый ключ в массиве результата `extractQuery` присутствует в индексированном объекте. Исходное значение `query` передаётся на случай, если оно потребуется методу `consistent`; с той же целью ему передаются массивы `queryKeys[]` и `nullFlags[]`, ранее возвращённые функцией `extractQuery`. В аргументе `extra_data` передаётся массив дополнительных данных, возвращённый функцией `extractQuery`, или `NULL`, если дополнительных данных нет.

Когда `extractQuery` возвращает ключ NULL в `queryKeys[]`, соответствующий элемент `check[]` содержит true, если индексированный объект содержит ключ NULL; то есть можно считать, что элементы `check[]` отражают условие `IS NOT DISTINCT FROM`. Функция `consistent` может проверить соответствующий элемент `nullFlags[]`, если ей нужно различать соответствие с обычным значением и соответствие с NULL.

В случае успеха в `*recheck` нужно записать true, если кортеж данных нужно перепроверить с оператором запроса, либо false, если проверка по индексу была точной. То есть результат false гарантирует, что кортеж данных не соответствует запросу; результат true со значением `*recheck`, равным false, гарантирует, что кортеж данных соответствует запросу; а результат true со значением `*recheck`, равным true, означает, что кортеж данных может соответствовать запросу, поэтому его нужно выбрать и перепроверить, применив оператор запроса непосредственно к исходному индексированному элементу.

`GinTernaryValue triConsistent(GinTernaryValue check[], StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[], Datum queryKeys[], bool nullFlags[])`

Функция `triConsistent` подобна `consistent`, но вместо булевых значений в векторе `check` ей передаются три варианта сравнений для каждого ключа: `GIN_TRUE`, `GIN_FALSE` и `GIN_MAYBE`. `GIN_FALSE` и `GIN_TRUE` имеют обычное логическое значение, тогда как `GIN_MAYBE` означает, что присутствие ключа неизвестно. Когда присутствуют значения `GIN_MAYBE`, функция должна возвращать `GIN_TRUE`, только если объект удовлетворяет запросу независимо от того, содержит ли индекс соответствующие ключи запроса. Подобным образом, функция должна возвращать `GIN_FALSE`, только если объект не удовлетворяет запросу независимо от того, содержит ли он ключи `GIN_MAYBE`. Если результат зависит от элементов `GIN_MAYBE`, то есть соответствие нельзя утверждать или отрицать в зависимости от известных ключей запроса, функция должна вернуть `GIN_MAYBE`.

Когда в векторе `check` нет элементов `GIN_MAYBE`, возвращаемое значение `GIN_MAYBE` равнозначно установленному флагу `recheck` в булевой функции `consistent`.

Кроме того, GIN нужно каким-то образом сортировать значения ключа, хранящиеся в индексе. Класс операторов может определить порядок сортировки, указав метод сравнения:

`int compare(Datum a, Datum b)`

Сравнивает два ключа (не индексированные объекты!) и возвращает целое меньше нуля, ноль или целое больше нуля, показывающее, что первый ключ меньше, равен или больше второго. Ключи NULL никогда не передаются этой функции.

Если же класс операторов не определяет метод `compare`, GIN попытается найти класс операторов B-дерева по умолчанию для типа данных ключа индекса и воспользоваться его функцией сравнения. Если класс операторов GIN предназначен только для одного типа данных, рекомендуется задавать функцию сравнения в этом классе операторов, так как поиск класса операторов B-дерева занимает несколько циклов. Однако для полиморфных классов операторов GIN (например, `array_ops`) задать одну функцию сравнения обычно не представляется возможным.

Дополнительно класс операторов для GIN может предоставить следующие методы:

`int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)`

Сравнивает ключ запроса с частичным соответствием с ключом индекса. Возвращает целое число, знак которого отражает результат сравнения: отрицательное число означает, что ключ индекса не соответствует запросу, но нужно продолжать сканирование индекса; ноль означает, что ключ индекса соответствует запросу; положительное число означает, что сканирование индекса нужно прекратить, так как других соответствий не будет. Функции передаётся номер стратегии `n` оператора, сформировавшего запрос частичного соответствия, на случай, если нужно знать его смысл, чтобы определить, когда прекращать сканирование. Кроме того, ей передаётся `extra_data` — соответствующий элемент массива дополнительных данных, сформированного функцией `extractQuery`, либо `NULL`, если дополнительных данных нет. Значения NULL этой функции никогда не передаются.

`void options(local_relopts *relopts)`

Определяет набор видимых пользователю параметров, управляющих поведением класса операторов.

Функции `options` передаётся указатель на структуру `local_relopts`, в которую нужно внести набор параметров, относящихся к классу операторов. Обращаться к этим параметрам из других опорных функций можно с помощью макросов `PG_HAS_OPCLASS_OPTIONS()` и `PG_GET_OPCLASS_OPTIONS()`.

Так как в GIN и извлечение ключа из индексируемых значений, и его представление допускают гибкость, могут быть полезны параметры для настройки этого индекса.

Для поддержки проверок на «частичное соответствие» класс операторов должен предоставить метод `comparePartial`, а метод `extractQuery` должен устанавливать параметр `pmatch`, когда встречается запрос на частичное соответствие. За подробностями обратитесь к Подразделу 62.4.4.2.

Фактические типы данных различных значений `Datum`, упоминаемых выше, зависят от класса операторов. Значения объектов, передаваемые в `extractValue`, всегда имеют входной тип класса операторов, а все значения ключей должны быть типа, заданного параметром `STORAGE` для класса. Типом аргумента `query`, передаваемого функциям `extractQuery`, `consistent` и `triConsistent`, будет тот тип, что указан в качестве типа правого операнда оператора-члена класса, определяемого по номеру стратегии. Это не обязательно должен быть индексируемый тип, достаточно лишь, чтобы из него можно было извлечь значения ключей, имеющие нужный тип. Однако рекомендуется, чтобы в SQL-объявлениях этих трёх опорных функций для аргумента `query` назначался индексируемый тип класса операторов, даже несмотря на то, что фактический тип может быть другим, в зависимости от оператора.

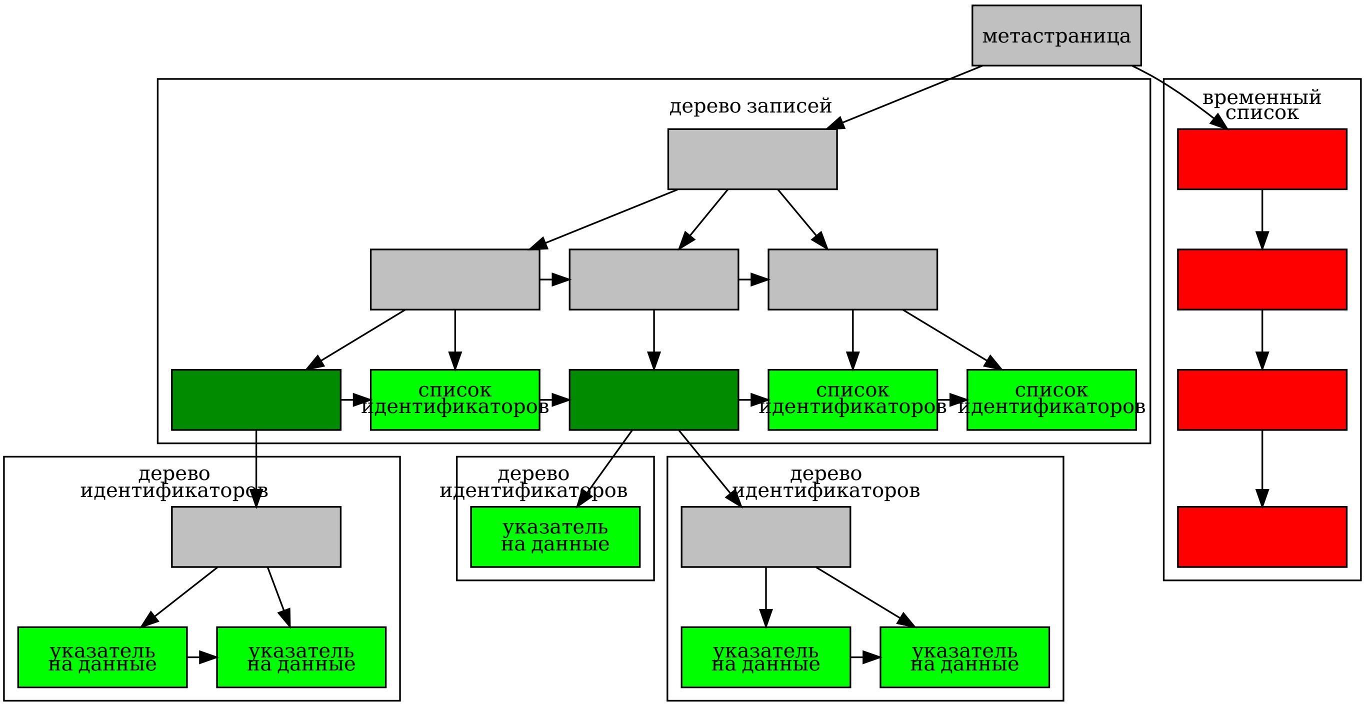
62.4.4. Реализация

Внутри индекс GIN содержит B-дерево, построенное по ключам, где каждый ключ является элементом одного или нескольких индексированных объектов (например, членом массива) и где каждый кортеж на страницах листьев содержит либо указатель на B-дерево указателей на данные («дерево идентификаторов»), либо простой список указателей на данные («список идентификаторов»), когда этот список достаточно мал, чтобы уместиться в одном кортеже индекса вместе со значением ключа. Эти компоненты GIN-индекса показаны на Рисунке 62.1.

Начиная с PostgreSQL версии 9.1, в индекс могут быть включены значения ключей, равные NULL. Кроме того, в индекс вставляются NULL для индексируемых объектов, равных NULL или не содержащих ключей, согласно функции `extractValue`. Это позволяет находить при поиске пустые объекты, когда они должны быть найдены.

Составные индексы GIN реализуются в виде одного B-дерева по составным значениям (номер столбца, значение ключа). Значения ключей для различных столбцов могут быть разных типов.

Рисунок 62.1. Внутреннее устройство GIN



62.4.4.1. Методика быстрого обновления GIN

Природа инвертированного индекса такова, что обновление GIN обычно медленная операция: при добавлении или изменении одной строки данных может потребоваться выполнить множество добавлений записей в индекс (для каждого ключа, извлечённого из индексируемого объекта). GIN может отложить большой объём этой работы, вставляя новые кортежи во временный, несортированный список записей, ожидающих индексации. Когда таблица очищается, автоматически анализируется, вызывается функция `gin_clean_pending_list` или размер этого списка временного списка становится больше чем gin_pending_list_limit, записи переносятся в основную структуру данных GIN теми же методами массового добавления данных, что и при начальном создании индекса. Это значительно увеличивает скорость обновления индекса GIN, даже с учётом дополнительных издержек при очистке. К тому же эту дополнительную работу можно выполнить в фоновом процессе, а не в процессе, непосредственно выполняющем запросы.

Основной недостаток такого подхода состоит в том, что при поиске необходимо не только проверить обычный индекс, но и просканировать список ожидающих записей, так что если этот список большой, поиск значительно замедляется. Ещё один недостаток состоит в том, что хотя в основном изменения производятся быстро, изменение, при котором этот список оказывается «слишком большим», влечёт необходимость немедленной очистки и поэтому выполняется гораздо дольше остальных изменений. Минимизировать эти недостатки можно, правильно организовав автоочистку.

Если выдержанность времени операций важнее скорости обновления, применение списка ожидающих записей можно отключить, выключив параметр хранения `fastupdate` для индекса GIN. За подробностями обратитесь к CREATE INDEX.

62.4.4.2. Алгоритм частичного соответствия

GIN может поддерживать проверки «частичного соответствия», когда запрос выявляет не точное соответствие одному или нескольким ключам, а возможные соответствия, попадающие в достаточно узкий диапазон значений ключей (при порядке сортировки ключей, определённым опорным методом `compare`). В этом случае метод `extractQuery` возвращает не значение ключа, которое должно соответствовать точно, а значение, определяющее нижнюю границу искомого диапазона, и устанавливает флаг `pmatch`. Затем диапазон ключей сканируется методом `comparePartial`. Метод `comparePartial` должен вернуть ноль при соответствии ключа индекса, отрицательное значение, если соответствия нет, но нужно продолжать проверку диапазона, и положительное значение, если ключ индекса оказался за искомым диапазоном.

62.4.5. Приёмы и советы по применению GIN

Создание или добавление

Добавление объектов в индекс GIN может выполняться медленно, так как для каждого объекта скорее всего потребуется добавлять множество ключей. Поэтому при массовом добавлении данных в таблицу рекомендуется удалить индекс GIN и пересоздать его по окончании добавления.

Когда для GIN включён режим `fastupdate` (за подробностями обратитесь к Подразделу 62.4.4.1), издержки меньше, чем когда он отключён. Но при очень большом объёме изменений всё же лучше удалить и заново создать индекс.

maintenance_work_mem

Время построения индекса GIN очень сильно зависит от параметра `maintenance_work_mem`; не стоит экономить на рабочей памяти при создании индекса.

gin_pending_list_limit

В процессе последовательных добавлений в существующий индекс GIN с включённым режимом `fastupdate`, система будет очищать список ожидающих индексации записей, когда его размер будет превышать `gin_pending_list_limit`. Во избежание значительных колебаний конечного времени ответа имеет смысл проводить очистку этого списка в фоновом режиме (то есть, применяя автоочистку). Избежать операций очистки на переднем плане можно, увеличив `gin_pending_list_limit` или проводя автоочистку более активно. Однако если вследствие увеличения порога операции очистки запустится очистка на переднем плане, она будет выполняться ещё дольше.

Значение `gin_pending_list_limit` можно переопределить для отдельных индексов GIN, изменив их параметры хранения, что позволяет задавать для каждого индекса GIN свой порог очистки. Например, можно увеличить порог только для часто обновляемых индексов GIN и оставить его низким для остальных.

gin_fuzzy_search_limit

Основной целью разработки индексов GIN было обеспечить поддержку хорошо расширяемого полнотекстового поиска в Postgres Pro, а при полнотекстовом поиске нередко возникают ситуации, когда возвращается очень большой набор результатов. Однако чаще всего так происходит, когда запрос содержит очень часто встречающиеся слова, так что полученный результат всё равно оказывается бесполезным. Так как чтение множества записей с диска и последующая сортировка их может занять много времени, это неприемлемо в производственных условиях. (Заметьте, что поиск по индексу при этом выполняется очень быстро.)

Для управляемого выполнения таких запросов в GIN введено настраиваемое мягкое ограничение сверху для числа возвращаемых строк: конфигурационный параметр `gin_fuzzy_search_limit`. По умолчанию он равен 0 (то есть ограничение отсутствует). Если он имеет ненулевое значение, возвращаемый набор строк будет случайно выбранным подмножеством всего набора результатов.

«Мягким» оно называется потому, что фактическое число возвращаемых строк может несколько отличаться от заданного значения, в зависимости от запроса и качества системного генератора случайных чисел.

Из практики, со значениями в несколько тысяч (например, 5000 — 20000) получаются приемлемые результаты.

62.4.6. Ограничения

GIN полагает, что индексируемые операторы являются строгими. Это означает, что функция `extractValue` вовсе не будет вызываться для значений NULL (вместо этого будет автоматически создаваться пустая запись в индексе), так же как и `extractQuery` не будет вызываться с искомым значением NULL (при этом сразу предполагается, что запрос не удовлетворяется). Заметьте однако, что при этом поддерживаются ключи NULL, содержащиеся в составных объектах или искомых значениях.

62.4.7. Примеры

В базовый дистрибутив Postgres Pro включены классы операторов GIN, перечисленные ранее в Таблице 62.3. Также классы операторов GIN содержатся в следующих модулях `contrib`:

`btree_gin`

Функциональность B-дерева для различных типов данных

`hstore`

Модуль для хранения пар (ключ, значение)

`intarray`

Расширенная поддержка `int[]`

`pg_trgm`

Схожесть текста на основе статистики триграмм

62.5. Индексы BRIN

62.5.1. Введение

BRIN расшифровывается как «Block Range Index» (Индекс зон блоков). BRIN предназначается для обработки очень больших таблиц, в которых определённые столбцы некоторым естественным образом коррелируют с их физическим расположением в таблице.

В BRIN используется понятие *зоны блоков* (или «зоны страниц»). *Зоной блоков* называется группа страниц, физически расположенных в таблице рядом; для каждой зоны в индексе сохраняется некоторая сводная информация. Например, в таблице заказов магазина может содержаться поле с датой добавления заказа, и практически всегда записи более ранних заказов и в таблице будут размещены ближе к началу; в таблице, содержащей столбец с почтовым индексом, также естественным образом могут группироваться записи по городам.

Индексы BRIN могут удовлетворять запросы, выполняя обычное сканирование по битовой карте, и будут возвращать все кортежи во всех страницах каждой зоны, если сводные данные, сохранённые в индексе, *соответствуют* условиям запроса. Исполнитель запроса должен перепроверить эти кортежи и отбросить те, что не соответствуют условиям запроса — другими словами, эти индексы неточные. Так как индекс BRIN очень маленький, сканирование индекса влечёт мизерные издержки по сравнению с последовательным сканированием, но может избавить от необходимости сканирования больших областей таблицы, которые определённо не содержат подходящие кортежи.

Конкретные данные, которые будут хранится в индексе BRIN, а также запросы, которые сможет поддержать этот индекс, зависят от класса операторов, выбранного для каждого столбца индекса. Например, типы данных с линейным порядком сортировки могут иметь классы операторов, хранящие минимальное и максимальное значение для каждой зоны блоков; для геометрических типов может храниться прямоугольник, вмещающий все объекты в зоне блоков.

Размер зоны блоков определяется в момент создания индекса параметром хранения `pages_per_range`. Число записей в индексе будет равняться размеру отношения в страницах, делённому на установленное значение `pages_per_range`. Таким образом, чем меньше это число, тем больше становится индекс (так как в нём требуется хранить больше элементов), но в то же время сводные данные могут быть более точными и большее число блоков данных может быть пропущено при сканировании индекса.

62.5.1.1. Обслуживание индекса

Во время создания индекса сканируются все существующие страницы, и в результате в индексе создаётся сводный кортеж для каждой зоны, в том числе, возможно неполной зоны в конце. По мере того, как данными наполняются новые страницы, если они оказываются в зонах, для которых уже есть сводная информация, она будет обновлена с учётом данных из новых кортежей. Если же создаётся новая страница, не попадающая в последнюю зону, для зоны, в которую попадает новая страница, сводная запись не рассчитывается автоматически; кортежи на таких страницах остаются неучтёнными, пока не будет проведён начальный расчёт сводных данных для этой зоны.

Начальный расчёт сводки для зоны страниц может производиться несколькими способами. Во-первых, сводка рассчитывается в ходе процедуры VACUUM, запущенной вручную или из процесса autovacuum, для всех существующих зон. Во-вторых, если включён параметр autosummarize для индекса (по умолчанию он отключён), при автоочистке этой базы данных производится расчёт сводки для всех зон без сводной информации, охватывающих недавно заполненные страницы таблицы, независимо от того, выполняется ли автоочистка именно этой таблицы; подробнее см. ниже.

Наконец, можно использовать следующие функции (пока они выполняются, search_path временно меняется на `pg_catalog, pg_temp`):

`brin_summarize_new_values(regclass)` , рассчитывающую сводку для зон без сводных данных;
`brin_summarize_range(regclass, bigint)` , рассчитывающую сводку для зоны без сводных данных, содержащей указанную страницу.

Когда включён режим автопересчёта сводки, при первом добавлении данных в первую страницу новой зоны процессу `autovacuum` передаётся запрос на расчёт сводки для этой зоны. Этот запрос будет выполнен при следующей обработке этой базы рабочим процессом автоочистки. Если очередь запросов переполнена, запрос в неё не записывается и в журнал сервера выводится соответствующее сообщение:

```
LOG:  request for BRIN range summarization for index "brin_wi_idx" page 128 was not
 recorded
```

В этом случае зона останется без сводного значения, а чтобы его рассчитать, нужно вручную выполнить очистку этой таблицы или вызвать одну из вышеуказанных функций.

И наоборот, можно удалить сводное значение для зоны, вызвав функцию `brin_desummarize_range(regclass, bigint)`, что может быть полезно, когда этот кортеж в индексе становится не очень хорошим представлением соответствующих данных, так как они изменились. За дополнительной информацией обратитесь к Подразделу 9.28.8.

62.5.2. Встроенные классы операторов

В базовый дистрибутив Postgres Pro включены классы операторов BRIN, перечисленные в Таблице 62.4.

Классы операторов *minmax* хранят минимальные и максимальные значения, встречающиеся в индексированном столбце в определённой зоне. Классы операторов *inclusion* хранят значение, в котором содержатся значения индексированного столбца в определённой зоне. Классы операторов *bloom* строят фильтр Блума для всех значений в определённой зоне. Классы операторов *minmax-*

multi хранят несколько минимальных и максимальных значений, встречающихся в индексированном столбце в определённой зоне.

Таблица 62.4. Встроенные классы операторов BRIN

Имя	Индексируемые операторы
`bit_minmax_ops`	`= (bit,bit)`
	`< (bit,bit)`
	`> (bit,bit)`
	`<= (bit,bit)`
	`>= (bit,bit)`
`box_inclusion_ops`	`@> (box,point)`
	`<< (box,box)`
	`&< (box,box)`
	`&> (box,box)`
	`>> (box,box)`
	`<@ (box,box)`
	`@> (box,box)`
	`~= (box,box)`
	`&& (box,box)`
	`<<| (box,box)`
	`&<| (box,box)`
	`|&> (box,box)`
	`|>> (box,box)`
`bpchar_bloom_ops`	`= (character,character)`
`bpchar_minmax_ops`	`= (character,character)`
	`< (character,character)`
	`<= (character,character)`
	`> (character,character)`
	`>= (character,character)`
`bytea_bloom_ops`	`= (bytea,bytea)`
`bytea_minmax_ops`	`= (bytea,bytea)`
	`< (bytea,bytea)`
	`<= (bytea,bytea)`
	`> (bytea,bytea)`
	`>= (bytea,bytea)`
`char_bloom_ops`	`= ("char","char")`
`char_minmax_ops`	`= ("char","char")`
	`< ("char","char")`
	`<= ("char","char")`
	`> ("char","char")`
	`>= ("char","char")`
`date_bloom_ops`	`= (date,date)`

Имя	**Индексируемые операторы**
date_minmax_ops	= (date,date)
	< (date,date)
	<= (date,date)
	> (date,date)
	>= (date,date)
date_minmax_multi_ops	= (date,date)
	< (date,date)
	<= (date,date)
	> (date,date)
	>= (date,date)
float4_bloom_ops	= (float4,float4)
float4_minmax_ops	= (float4,float4)
	< (float4,float4)
	> (float4,float4)
	<= (float4,float4)
	>= (float4,float4)
float4_minmax_multi_ops	= (float4,float4)
	< (float4,float4)
	> (float4,float4)
	<= (float4,float4)
	>= (float4,float4)
float8_bloom_ops	= (float8,float8)
float8_minmax_ops	= (float8,float8)
	< (float8,float8)
	<= (float8,float8)
	> (float8,float8)
	>= (float8,float8)
float8_minmax_multi_ops	= (float8,float8)
	< (float8,float8)
	<= (float8,float8)
	> (float8,float8)
	>= (float8,float8)
inet_inclusion_ops	<< (inet,inet)
	<<= (inet,inet)
	>> (inet,inet)
	>>= (inet,inet)
	= (inet,inet)
	&& (inet,inet)
inet_bloom_ops	= (inet,inet)
inet_minmax_ops	= (inet,inet)

Имя	Индексируемые операторы
	`< (inet,inet)`
	`<= (inet,inet)`
	`> (inet,inet)`
	`>= (inet,inet)`
`inet_minmax_multi_ops`	`= (inet,inet)`
	`< (inet,inet)`
	`<= (inet,inet)`
	`> (inet,inet)`
	`>= (inet,inet)`
`int2_bloom_ops`	`= (int2,int2)`
`int2_minmax_ops`	`= (int2,int2)`
	`< (int2,int2)`
	`> (int2,int2)`
	`<= (int2,int2)`
	`>= (int2,int2)`
`int2_minmax_multi_ops`	`= (int2,int2)`
	`< (int2,int2)`
	`> (int2,int2)`
	`<= (int2,int2)`
	`>= (int2,int2)`
`int4_bloom_ops`	`= (int4,int4)`
`int4_minmax_ops`	`= (int4,int4)`
	`< (int4,int4)`
	`> (int4,int4)`
	`<= (int4,int4)`
	`>= (int4,int4)`
`int4_minmax_multi_ops`	`= (int4,int4)`
	`< (int4,int4)`
	`> (int4,int4)`
	`<= (int4,int4)`
	`>= (int4,int4)`
`int8_bloom_ops`	`= (bigint,bigint)`
`int8_minmax_ops`	`= (bigint,bigint)`
	`< (bigint,bigint)`
	`> (bigint,bigint)`
	`<= (bigint,bigint)`
	`>= (bigint,bigint)`
`int8_minmax_multi_ops`	`= (bigint,bigint)`
	`< (bigint,bigint)`
	`> (bigint,bigint)`

Имя	**Индексируемые операторы**
	`<= (bigint,bigint)`
	`>= (bigint,bigint)`
`interval_bloom_ops`	`= (interval,interval)`
`interval_minmax_ops`	`= (interval,interval)`
	`< (interval,interval)`
	`<= (interval,interval)`
	`> (interval,interval)`
	`>= (interval,interval)`
`interval_minmax_multi_ops`	`= (interval,interval)`
	`< (interval,interval)`
	`<= (interval,interval)`
	`> (interval,interval)`
	`>= (interval,interval)`
`macaddr_bloom_ops`	`= (macaddr,macaddr)`
`macaddr_minmax_ops`	`= (macaddr,macaddr)`
	`< (macaddr,macaddr)`
	`<= (macaddr,macaddr)`
	`> (macaddr,macaddr)`
	`>= (macaddr,macaddr)`
`macaddr_minmax_multi_ops`	`= (macaddr,macaddr)`
	`< (macaddr,macaddr)`
	`<= (macaddr,macaddr)`
	`> (macaddr,macaddr)`
	`>= (macaddr,macaddr)`
`macaddr8_bloom_ops`	`= (macaddr8,macaddr8)`
`macaddr8_minmax_ops`	`= (macaddr8,macaddr8)`
	`< (macaddr8,macaddr8)`
	`<= (macaddr8,macaddr8)`
	`> (macaddr8,macaddr8)`
	`>= (macaddr8,macaddr8)`
`macaddr8_minmax_multi_ops`	`= (macaddr8,macaddr8)`
	`< (macaddr8,macaddr8)`
	`<= (macaddr8,macaddr8)`
	`> (macaddr8,macaddr8)`
	`>= (macaddr8,macaddr8)`
`name_bloom_ops`	`= (name,name)`
`name_minmax_ops`	`= (name,name)`
	`< (name,name)`
	`<= (name,name)`
	`> (name,name)`

Имя	Индексируемые операторы
	`>= (name,name)`
`numeric_bloom_ops`	`= (numeric,numeric)`
`numeric_minmax_ops`	`= (numeric,numeric)`
	`< (numeric,numeric)`
	`<= (numeric,numeric)`
	`> (numeric,numeric)`
	`>= (numeric,numeric)`
`numeric_minmax_multi_ops`	`= (numeric,numeric)`
	`< (numeric,numeric)`
	`<= (numeric,numeric)`
	`> (numeric,numeric)`
	`>= (numeric,numeric)`
`oid_bloom_ops`	`= (oid,oid)`
`oid_minmax_ops`	`= (oid,oid)`
	`< (oid,oid)`
	`> (oid,oid)`
	`<= (oid,oid)`
	`>= (oid,oid)`
`oid_minmax_multi_ops`	`= (oid,oid)`
	`< (oid,oid)`
	`> (oid,oid)`
	`<= (oid,oid)`
	`>= (oid,oid)`
`pg_lsn_bloom_ops`	`= (pg_lsn,pg_lsn)`
`pg_lsn_minmax_ops`	`= (pg_lsn,pg_lsn)`
	`< (pg_lsn,pg_lsn)`
	`> (pg_lsn,pg_lsn)`
	`<= (pg_lsn,pg_lsn)`
	`>= (pg_lsn,pg_lsn)`
`pg_lsn_minmax_multi_ops`	`= (pg_lsn,pg_lsn)`
	`< (pg_lsn,pg_lsn)`
	`> (pg_lsn,pg_lsn)`
	`<= (pg_lsn,pg_lsn)`
	`>= (pg_lsn,pg_lsn)`
`range_inclusion_ops`	`= (anyrange,anyrange)`
	`< (anyrange,anyrange)`
	`<= (anyrange,anyrange)`
	`>= (anyrange,anyrange)`
	`> (anyrange,anyrange)`
	`&& (anyrange,anyrange)`

Имя	Индексируемые операторы
	@> (anyrange,anyelement)
	@> (anyrange,anyrange)
	<@ (anyrange,anyrange)
	<< (anyrange,anyrange)
	>> (anyrange,anyrange)
	&< (anyrange,anyrange)
	&> (anyrange,anyrange)
	-|- (anyrange,anyrange)
text_bloom_ops	= (text,text)
text_minmax_ops	= (text,text)
	< (text,text)
	<= (text,text)
	> (text,text)
	>= (text,text)
tid_bloom_ops	= (tid,tid)
tid_minmax_ops	= (tid,tid)
	< (tid,tid)
	> (tid,tid)
	<= (tid,tid)
	>= (tid,tid)
tid_minmax_multi_ops	= (tid,tid)
	< (tid,tid)
	> (tid,tid)
	<= (tid,tid)
	>= (tid,tid)
timestamp_bloom_ops	= (timestamp,timestamp)
timestamp_minmax_ops	= (timestamp,timestamp)
	< (timestamp,timestamp)
	<= (timestamp,timestamp)
	> (timestamp,timestamp)
	>= (timestamp,timestamp)
timestamp_minmax_multi_ops	= (timestamp,timestamp)
	< (timestamp,timestamp)
	<= (timestamp,timestamp)
	> (timestamp,timestamp)
	>= (timestamp,timestamp)
timestamptz_bloom_ops	= (timestamptz,timestamptz)
timestamptz_minmax_ops	= (timestamptz,timestamptz)
	< (timestamptz,timestamptz)
	<= (timestamptz,timestamptz)

Имя	Индексируемые операторы
	> (timestamptz,timestamptz)
	>= (timestamptz,timestamptz)
timestamptz_minmax_multi_ops	= (timestamptz,timestamptz)
	< (timestamptz,timestamptz)
	<= (timestamptz,timestamptz)
	> (timestamptz,timestamptz)
	>= (timestamptz,timestamptz)
time_bloom_ops	= (time,time)
time_minmax_ops	= (time,time)
	< (time,time)
	<= (time,time)
	> (time,time)
	>= (time,time)
time_minmax_multi_ops	= (time,time)
	< (time,time)
	<= (time,time)
	> (time,time)
	>= (time,time)
timetz_bloom_ops	= (timetz,timetz)
timetz_minmax_ops	= (timetz,timetz)
	< (timetz,timetz)
	<= (timetz,timetz)
	> (timetz,timetz)
	>= (timetz,timetz)
timetz_minmax_multi_ops	= (timetz,timetz)
	< (timetz,timetz)
	<= (timetz,timetz)
	> (timetz,timetz)
	>= (timetz,timetz)
uuid_bloom_ops	= (uuid,uuid)
uuid_minmax_ops	= (uuid,uuid)
	< (uuid,uuid)
	> (uuid,uuid)
	<= (uuid,uuid)
	>= (uuid,uuid)
uuid_minmax_multi_ops	= (uuid,uuid)
	< (uuid,uuid)
	> (uuid,uuid)
	<= (uuid,uuid)
	>= (uuid,uuid)

Имя	Индексируемые операторы
`varbit_minmax_ops`	`= (varbit,varbit)`
	`< (varbit,varbit)`
	`> (varbit,varbit)`
	`<= (varbit,varbit)`
	`>= (varbit,varbit)`

62.5.2.1. Параметры класса операторов

Для некоторых встроенных классов операторов можно задавать параметры, влияющие на их поведение, а именно: для классов операторов `bloom` и `minmax-multi`. У разных классов операторов разный набор допустимых параметров.

Классы операторов bloom принимают эти параметры:

`n_distinct_per_range`

Определяет предполагаемое количество различных значений (без учёта NULL) в зоне блоков, в зависимости от которого выбирается размер фильтра Блума в BRIN-индексе. Работает аналогично параметру `n_distinct` для ALTER TABLE. Положительное значение непосредственно задаёт предполагаемое количество. Минимальное количество различных значений (без учёта NULL) значений — `16`. Если заданное значение отрицательно, то количество таких значений линейно зависит от максимального возможного числа кортежей в зоне блоков (один блок может содержать до 290 строк). Коэффициентом зависимости будет модуль этого значения, который не должен превышать 1. Значение по умолчанию — `-0.1`.

`false_positive_rate`

Определяет ожидаемую долю ложных положительных срабатываний, в зависимости от которой выбирается размер фильтра Блума в BRIN-индексе. Принимаются значения от 0,0001 до 0,25. Значение по умолчанию — 0,01, то есть вероятность ложных положительных срабатываний равна 1%.

Классы операторов minmax-multi принимают эти параметры:

`values_per_range`

Определяет максимальное количество значений, хранимых BRIN-индексами типа minmax для обобщения информации по зоне блоков. Каждое значение может представлять собой либо точку, либо границу интервала. Принимаются значения от 8 до 256, а значение по умолчанию — 32.

62.5.3. Расширяемость

Интерфейс BRIN характеризуется высоким уровнем абстракции и таким образом требует от разработчика метода доступа реализовать только смысловое наполнение обрабатываемого типа данных. Уровень BRIN берёт на себя заботу о параллельном доступе, поддержке журнала и поиске в структуре индекса.

Всё, что нужно, чтобы получить работающий метод доступа BRIN — это реализовать несколько пользовательских методов, определяющих поведение сводных значений, хранящихся в индексе, и их взаимоотношения с ключами сканирования. Словом, BRIN сочетает расширяемость с универсальностью, повторным использованием кода и аккуратным интерфейсом.

Класс операторов для BRIN должен предоставлять четыре метода:

`BrinOpcInfo *opcInfo(Oid type_oid)`

Возвращает внутреннюю информацию о сводных данных индексированных столбцов. Возвращаемое значение должно указывать на `BrinOpcInfo` (в памяти palloc) со следующим определением:

```
typedef struct BrinOpcInfo
{
    /* Число полей, хранящихся в столбце индекса этого класса операторов */
    uint16      oi_nstored;

    /* Непрозрачный указатель для внутреннего использования классом операторов */
    void       *oi_opaque;

    /* Элементы кеша типов для сохранённых столбцов */
    TypeCacheEntry *oi_typcache[FLEXIBLE_ARRAY_MEMBER];
} BrinOpcInfo;
```

Поле `BrinOpcInfo.oi_opaque` могут использовать подпрограммы класса операторов для передачи информации опорным функциям при сканировании индекса.

`bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey *keys, int nkeys)`

Показывает, соответствуют ли все значения ScanKey заданным индексированным значениям некоторой зоны. Номер целевого атрибута передаётся в составе ключа сканирования. Для одного атрибута может передаваться несколько ключей сканирования. Количество записей определяется параметром `nkeys`.

`bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey key)`

Показывает, соответствует ли значение ScanKey заданным индексированным значениям некоторой зоны. Номер целевого атрибута передаётся в составе ключа сканирования. Этот обратно совместимый вариант функции consistent устарел.

`bool addValue(BrinDesc *bdesc, BrinValues *column, Datum newval, bool isnull)`

Для заданного кортежа индекса и индексируемого значения изменяет выбранный атрибут кортежа, чтобы он дополнительно охватывал новое значение. Если в кортеж вносятся какие-либо изменения, возвращается `true`.

`bool unionTuples(BrinDesc *bdesc, BrinValues *a, BrinValues *b)`

Консолидирует два кортежа индекса. Получая два кортежа, изменяет выбранный атрибут первого из них, что он охватывал оба кортежа. Второй кортеж не изменяется.

Дополнительно класс операторов для BRIN может предоставить следующий метод:

`void options(local_relopts *relopts)`

Определяет набор видимых пользователю параметров, управляющих поведением класса операторов.

Функции `options` передаётся указатель на структуру `local_relopts`, в которую нужно внести набор параметров, относящихся к классу операторов. Обращаться к этим параметрам из других опорных функций можно с помощью макросов `PG_HAS_OPCLASS_OPTIONS()` и `PG_GET_OPCLASS_OPTIONS()`.

Так как в BRIN и извлечение ключа из индексируемых значений, и его представление допускают гибкость, могут быть полезны параметры для настройки этого индекса.

Основной дистрибутив включает поддержку четырёх типов классов операторов: minmax, minmax-multi, inclusion и bloom. Определения классов операторов, использующие их, представлены для встроенных типов данных, насколько это уместно. Пользователь может определить дополнительные классы операторов для других типов данных, применяя аналогичные определения, и обойтись таким образом без написания кода; достаточно будет объявить нужные записи в каталоге. Заметьте, что предположения о семантике стратегий операторов зашиты в исходном коде опорных функций.

Также возможно создать классы операторов, воплощающие полностью другую семантику, разработав реализации четырёх основных опорных функций, описанных выше. Заметьте, что обратная совместимость между разными основными версиями не гарантируется: к примеру, в следующих выпусках могут потребоваться дополнительные опорные функции.

При написании класса операторов для типа данных, представляющего полностью упорядоченное множество, можно использовать опорные функции minmax вместе с соответствующими операторами, как показано в Таблице 62.5. Все члены класса операторов (функции и операторы) являются обязательными.

Таблица 62.5. Номера стратегий и опорных функций для классов операторов minmax

Член класса операторов	**Объект**
Опорная функция 1	внутренняя функция `brin_minmax_opcinfo()`
Опорная функция 2	внутренняя функция `brin_minmax_add_value()`
Опорная функция 3	внутренняя функция `brin_minmax_consistent()`
Опорная функция 4	внутренняя функция `brin_minmax_union()`
Стратегия оператора 1	оператор меньше
Стратегия оператора 2	оператор меньше-или-равно
Стратегия оператора 3	оператор равно
Стратегия оператора 4	оператор больше-или-равно
Стратегия оператора 5	оператор больше

При написании класса операторов для сложного типа данных, значения которого включаются в другой тип, можно использовать опорные функции inclusion вместе с соответствующими операторами, как показано в Таблице 62.6. Для этого требуется одна дополнительная функция, которую можно написать на любом языке. Для расширенной функциональности можно определить другие функции. Все операторы являются необязательными. Некоторые из них требует наличия других, что показано в таблице как зависимости.

Таблица 62.6. Номера стратегий и опорных функций для классов операторов inclusion

Член класса операторов	**Объект**	**Зависимость**
Опорная функция 1	внутренняя функция `brin_inclusion_opcinfo()`	
Опорная функция 2	внутренняя функция `brin_inclusion_add_value()`	
Опорная функция 3	внутренняя функция `brin_inclusion_consistent()`	
Опорная функция 4	внутренняя функция `brin_inclusion_union()`	
Опорная функция 11	функция для слияния двух элементов	
Опорная функция 12	необязательная функция для проверки возможности слияния двух элементов	
Опорная функция 13	необязательная функция для проверки, содержится ли один элемент в другом	
Опорная функция 14	необязательная функция для проверки, является ли элемент пустым	
Стратегия оператора 1	оператор левее	Стратегия оператора 4
Стратегия оператора 2	оператор не-простирается-правее	Стратегия оператора 5

Член класса операторов	**Объект**	**Зависимость**
Стратегия оператора 3	оператор перекрывается	
Стратегия оператора 4	оператор не-простирается-левее	Стратегия оператора 1
Стратегия оператора 5	оператор правее	Стратегия оператора 2
Стратегия оператора 6, 18	оператор то-же-или-равно	Стратегия оператора 7
Стратегия оператора 7, 16, 24, 25	оператор содержит-или-равно	
Стратегия оператора 8, 26, 27	оператор содержится-в-или-равно	Стратегия оператора 3
Стратегия оператора 9	оператор не-простирается-выше	Стратегия оператора 11
Стратегия оператора 10	оператор ниже	Стратегия оператора 12
Стратегия оператора 11	оператор выше	Стратегия оператора 9
Стратегия оператора 12	оператор не-простирается-ниже	Стратегия оператора 10
Стратегия оператора 20	оператор меньше	Стратегия оператора 5
Стратегия оператора 21	оператор меньше-или-равно	Стратегия оператора 5
Стратегия оператора 22	оператор больше	Стратегия оператора 1
Стратегия оператора 23	оператор больше-или-равно	Стратегия оператора 1

Номера опорных функций от 1 до 10 зарезервированы для внутренних функций BRIN, так что функции уровня SQL начинаются с номера 11. Опорная функция номер 11 является основной, необходимой для построения индекса. Она должна принимать два аргумента того же типа данных, что и целевой тип класса, и возвращать их объединение. Класс операторов inclusion может сохранять значения объединения в различных типах данных, в зависимости от параметра `STORAGE`. Возвращаемое функцией объединения значение должно соответствовать типу данных `STORAGE`.

Опорные функции под номерами 12 и 14 предоставляются для поддержки нерегулярностей встроенных типов данных. Функция номер 12 применяется для поддержки работы с сетевыми адресами из различных семейств, которые нельзя объединять. Функция номер 14 применяется для поддержки зон с пустыми значениями. Функция номер 13 является необязательной, но рекомендуемой; она проверяет новое значение, прежде чем оно будет передано функции объединения. Инфраструктура BRIN может соптимизировать некоторые операции, когда объединение не меняется, поэтому применение этой функции может способствовать увеличению быстродействия индекса.

При написании класса операторов для типа данных, который реализует только оператор равенства и поддерживает хеширование, можно использовать опорные процедуры bloom вместе с соответствующими операторами, как показано в Таблица 62.7. Все члены класса операторов (процедуры и операторы) являются обязательными.

Таблица 62.7. Номера стратегий и опорных процедур для классов операторов bloom

Член класса операторов	**Объект**
Опорная процедура 1	внутренняя функция `brin_bloom_opcinfo()`

Член класса операторов	Объект
Опорная процедура 2	внутренняя функция `brin_bloom_add_value()`
Опорная процедура 3	внутренняя функция `brin_bloom_consistent()`
Опорная процедура 4	внутренняя функция `brin_bloom_union()`
Опорная процедура 5	внутренняя функция `brin_bloom_options()`
Опорная процедура 11	функция для вычисления хеша элемента
Стратегия оператора 1	оператор равно

Номера опорных процедур 1-10 зарезервированы для внутренних функций BRIN, так что функции уровня SQL начинаются с номера 11. Опорная функция номер 11 является основной, необходимой для построения индекса. Она должна принимать один аргумент того же типа данных, что и класс оператора, и возвращать хеш значения.

Класс операторов minmax-multi также используется для типов данных, представляющих полностью упорядоченное множество, и может рассматриваться просто как расширение класса операторов minmax. Класс операторов minmax для каждого блока обобщает информацию по значениям в один непрерывный интервал, а minmax-multi позволяет обобщить информацию по значениям в несколько меньших интервалов для более эффективной обработки выпадающих значений. При написании класса операторов можно использовать опорные процедуры minmax-multi вместе с соответствующими операторами, как показано в Таблица 62.8. Все члены класса операторов (процедуры и операторы) являются обязательными.

Таблица 62.8. Номера стратегий и опорных процедур для классов операторов minmax-multi

Член класса операторов	Объект
Опорная процедура 1	внутренняя функция `brin_minmax_multi_opcinfo()`
Опорная процедура 2	внутренняя функция `brin_minmax_add_value()`
Опорная процедура 3	внутренняя функция `brin_minmax_consistent()`
Опорная процедура 4	внутренняя функция `brin_minmax_union()`
Опорная процедура 5	внутренняя функция `brin_minmax_multi_options()`
Опорная процедура 11	функция для вычисления расстояния между двумя значениями (размер зоны)
Стратегия оператора 1	оператор меньше
Стратегия оператора 2	оператор меньше-или-равно
Стратегия оператора 3	оператор равно
Стратегия оператора 4	оператор больше-или-равно
Стратегия оператора 5	оператор больше

Классы операторов minmax и inclusion поддерживают межтиповые операторы, хотя с ними зависимости становятся более сложными. Класс minmax требует, чтобы для двух аргументов одного типа определялся полный набор операторов. Это позволяет поддерживать дополнительные типы данных, определяя дополнительные наборы операторов. Стратегии операторов класса inclusion могут зависеть от других стратегий, как показано в Таблице 62.6, или от своих собственных стратегий. Для них требуется, чтобы был определён необходимый оператор с типом данных `STORAGE` для левого аргумента и другим поддерживаемым типом для правого аргумента реализуемого оператора.

См. определение `float4_minmax_ops` в качестве примера для minmax и `box_inclusion_ops` в качестве примера для inclusion.

62.6. Хеш-индексы

62.6.1. Обзор

Postgres Pro поддерживает реализацию хеш-индексов, которые хранятся на диске и могут полностью восстанавливаться после сбоя. Хеш-индекс может быть построен по данным любого типа, в том числе типа, для которого не определён линейный порядок. Хеш-индексы хранят только хеш-значение индексируемых данных, поэтому размер индексируемого столбца данных неограничен.

Хеш-индексы могут строиться только по одному столбцу и не позволяют проверять уникальность.

Хеш-индексы поддерживают только оператор `=`, поэтому для предложений WHERE, в которых фигурируют проверки интервалов, хеш-индексы будут бесполезны.

Каждый кортеж хеш-индекса хранит только 4-байтовое хеш-значение, а не фактическое значение столбца. В результате хеш-индексы могут быть намного меньше, чем B-деревья, при индексировании более длинных элементов данных, таких как UUID, URL-адреса и т. д. Поскольку значения столбцов в хеш-индексе отсутствуют, при его сканировании результаты будут неточными. Хеш-индексы могут участвовать в сканировании индекса по битовой карте и обратном сканировании.

Хеш-индексы предназначены в первую очередь для нагрузки с большим количеством операций SELECT и UPDATE, которые выполняют сканирование с проверкой равенства для больших таблиц. В индексе-B-дереве поиск должен проходить по дереву до тех пор, пока не будет найдена листовая страница. В таблицах с миллионами строк такой «спуск» может увеличить время доступа к данным. Листовым страницам в хеш-индексе соответствуют страницы ячеек. В отличие от индекса-B-дерева хеш-индекс позволяет напрямую обращаться к страницам ячеек, тем самым потенциально сокращая время доступа к индексу в больших таблицах. Это сокращение «логического ввода-вывода» становится ещё более заметным для индексов/данных, которые не умещаются в общих буферах/ОЗУ.

Конструкция хеш-индексов в принципе приспособлена для индексирования данных с неравномерным распределением хешей, однако наиболее эффективен в них прямой доступ к страницам ячеек, осуществляемый при равномерном распределении значений хеша. Когда при добавлении нового значения оказывается, что страница ячеек заполнена, к этой странице привязываются дополнительные страницы переполнения, локально расширяющие хранилище для индексных кортежей, соответствующих этому значению хеша. Затем в ходе выполнения последующих запросов при сканировании ячейки хеша потребуется просканировать все страницы переполнения. Таким образом, несбалансированный хеш-индекс для некоторых данных фактически может оказаться хуже индекса-B-дерева по количеству требуемых обращений к блокам.

Ввиду неэффективности хеш-индексов в случаях переполнения можно сказать, что они лучше всего подходят для уникальных данных, данных с низкой степенью уникальности или данных с небольшим количеством строк в одной ячейке хеша. Один из возможных способов избежать проблем — исключить из индекса значения с низкой степенью уникальности, определив частичный индекс по условию, но такой вариант может подойти далеко не всегда.

Как и B-деревья, хеш-индексы выполняют простое удаление индексных кортежей. Это отложенная операция обслуживания, удаляющая индексные кортежи, о которых известно, что их можно безопасно стереть (то есть те, для идентификаторов которых установлен бит LP_DEAD). Если при добавлении нового кортежа обнаруживается, что на странице нет свободного места, данная операция пытается предотвратить создание новой страницы переполнения, удаляя мёртвые индексные кортежи (что возможно, только если страница не закреплена). Удаление мёртвых индексных указателей также происходит во время операции VACUUM.

Если возможно, операция VACUUM также попытается уместить индексные кортежи в наименьшем количестве страниц переполнения, минимизируя цепочку переполнения. Если страницы переполнения становятся пустыми, они могут быть переработаны для повторного использования другими ячейками, но они никогда не возвращаются операционной системе. В настоящее время единственная возможность уменьшить размер хеш-индекса — перестроить его командой REINDEX. Также на данный момент невозможно уменьшить количество ячеек.

Хеш-индексы могут расширяться за счёт увеличения количества страниц ячеек по мере увеличения числа индексируемых строк. Сопоставление значения хеша с номером ячейки выбирается таким образом, чтобы индекс мог расширяться последовательно. Когда в индекс нужно будет добавить новую ячейку, «разделить» придётся ровно одну существующую ячейку, при этом некоторые из её кортежей будут перенесены в новую ячейку в соответствии с изменённым сопоставлением.

Это расширение выполняется на переднем плане, и из-за него может увеличиваться время добавления данных для пользователей. Поэтому хеш-индексы могут не подходить для таблиц с быстро увеличивающимся числом строк.

62.6.2. Реализация

В хеш-индексе есть четыре типа страниц: метастраница (нулевая страница), содержащая статически размещённую управляющую информацию, основные страницы ячеек, страницы переполнения и страницы битовой карты, в которых отслеживаются освободившиеся страницы переполнения, пригодные для повторного использования. Страницы битовой карты для целей адресации рассматриваются как подмножество страниц переполнения.

Как сканирование индекса, так и добавление кортежей требует вычисления ячейки, в которой должен располагаться данный кортеж. Для этого нужно знать количество ячеек и значения `highmask` и `lowmask` из метастраницы; однако для оптимальной производительности нежелательно блокировать и закреплять метастраницу при каждой такой операции. Поэтому в кеше отношений каждого обслуживающего процесса имеется кешированная копия метастраницы. Благодаря этому обеспечивается правильное сопоставление ячеек при условии, что целевая ячейка не была разделена с момента последнего обновления кеша.

Основные страницы ячеек и страницы переполнения выделяются независимо, поскольку для каждого конкретного индекса может потребоваться больше или меньше страниц переполнения относительно количества ячеек в нём. В расчёте хеш-кодов используется интересный набор правил, позволяющий поддерживать переменное количество страниц переполнения, не требуя перемещения основных страниц ячеек после их создания.

Каждая проиндексированная строка в таблице представлена в хеш-индексе одним индексным кортежем. Кортежи хеш-индекса хранятся в страницах ячеек и страницах переполнения (если они есть). Для ускорения поиска элементы индекса размещаются в каждой конкретной странице упорядоченными по хеш-коду, что позволяет использовать в рамках страницы двоичный поиск. Однако заметьте, что никаких предположений об относительном порядке хеш-кодов в разных страницах, относящихся к одной ячейки, делать нельзя.

Алгоритмы разделения ячеек для расширения хеш-индекса слишком сложны, чтобы упоминать о них здесь. Алгоритм разделения защищён от сбоев и может быть перезапущен, если разделение не было успешно завершено.

Глава 63. Физическое хранение базы данных

В данной главе рассматривается формат физического хранения, используемый базами данных Postgres Pro.

63.1. Размещение файлов базы данных

В данном разделе описывается формат хранения на уровне файлов и каталогов.

Файлы конфигурации и файлы данных, используемые кластером базы данных, традиционно хранятся вместе в каталоге данных, который обычно называют `PGDATA` (по имени переменной среды, которую можно использовать для его определения). Обычно `PGDATA` находится в `/var/lib/pgpro/std-17/data`. На одной и той же машине может находиться множество кластеров, управляемых различными экземплярами сервера.

В каталоге `PGDATA` содержится несколько подкаталогов и управляющих файлов, как показано в Таблице 63.1. В дополнение к этим обязательным элементам конфигурационные файлы кластера `postgresql.conf`, `pg_hba.conf` и `pg_ident.conf` традиционно хранятся в `PGDATA`, хотя их можно разместить и в другом месте.

Таблица 63.1. Содержание `PGDATA`

Элемент	Описание
`PG_VERSION`	Файл, содержащий номер основной версии Postgres Pro
`base`	Подкаталог, содержащий подкаталоги для каждой базы данных
`current_logfiles`	Файл, в котором отмечается, в какие файлы журналов производит запись сборщик сообщений
`global`	Подкаталог, содержащий общие таблицы кластера, такие как `pg_database`
`pg_commit_ts`	Подкаталог, содержащий данные о времени фиксации транзакций
`pg_dynshmem`	Подкаталог, содержащий файлы, используемые подсистемой динамической разделяемой памяти
`pg_logical`	Подкаталог, содержащий данные о состоянии для логического декодирования
`pg_multixact`	Подкаталог, содержащий данные о состоянии мультитранзакций (используемые для разделяемой блокировки строк)
`pg_notify`	Подкаталог, содержащий данные состояния прослушивания и уведомлений (LISTEN/NOTIFY)
`pg_replslot`	Подкаталог, содержащий данные слота репликации
`pg_serial`	Подкаталог, содержащий информацию о выполненных сериализуемых транзакциях.
`pg_snapshots`	Подкаталог, содержащий экспортированные снимки (snapshots)

Элемент	Описание
`pg_stat`	Подкаталог, содержащий постоянные файлы для подсистемы статистики.
`pg_stat_tmp`	Подкаталог, содержащий временные файлы для подсистемы статистики
`pg_subtrans`	Подкаталог, содержащий данные о состоянии подтранзакций
`pg_tblspc`	Подкаталог, содержащий символические ссылки на табличные пространства
`pg_twophase`	Подкаталог, содержащий файлы состояний для подготовленных транзакций
`pg_wal`	Подкаталог, содержащий файлы WAL (журнал предзаписи)
`pg_xact`	Подкаталог, содержащий данные о состоянии транзакции
`postgresql.auto.conf`	Файл, используемый для хранения параметров конфигурации, которые устанавливаются при помощи `ALTER SYSTEM`
`postmaster.opts`	Файл, содержащий параметры командной строки, с которыми сервер был запущен в последний раз
`postmaster.pid`	Файл блокировки, содержащий идентификатор (ID) текущего управляющего процесса (PID), путь к каталогу данных кластера, время запуска управляющего процесса, номер порта, путь к каталогу Unix-сокета (может быть пустым), первый корректный адрес прослушивания (listen_address) (IP-адрес или `*`, либо пустое значение в случае отсутствия прослушивания по TCP), и ID сегмента разделяемой памяти (этот файл отсутствует после остановки сервера).

Для каждой базы данных в кластере существует подкаталог внутри `PGDATA/base`, названный по OID базы данных в `pg_database`. Этот подкаталог по умолчанию является местом хранения файлов базы данных; в частности, там хранятся её системные каталоги.

Обратите внимание, в следующих разделах описывается поведение встроенного табличного метода доступа `heap` и встроенных индексных методов доступа. Однако Postgres Pro по своей природе является расширяемым, и поэтому другие методы доступа могут работать иначе.

Каждая таблица и индекс хранятся в отдельном файле. Для обычных отношений, эти файлы получают имя по номеру *файлового узла* таблицы или индекса, который содержится в `pg_class.relfilenode`. Но для временных отношений, имя файла имеет форму `t`*`BBB`*`_`*`FFF`*, где *`BBB`* — номер обслуживающего процесса, который создал данный файл, а *`FFF`* — номер файлового узла. В обоих случаях, помимо главного файла (также называемого основным слоем), у каждой таблицы и индекса есть *карта свободного пространства* (см. Раздел 63.3), в которой хранится информация о свободном пространстве в данном отношении. Имя файла карты свободного пространства образуется из номера файлового узла с суффиксом `_fsm`. Также таблицы имеют *карту видимости*, хранящуюся в слое с суффиксом `_vm`, для отслеживания страниц, не содержащих мёртвых записей. Карта видимости подробнее описана в Разделе 63.4. Нежурналируемые таблицы и индексы имеют третий слой, так называемый слой инициализации, имя которого содержит суффикс `_init` (см. Раздел 63.5).

Внимание

Заметьте, что хотя номер файла таблицы часто совпадает с её OID, так бывает *не* всегда; некоторые операции, например, `TRUNCATE`, `REINDEX`, `CLUSTER` и некоторые формы команды `ALTER TABLE` могут изменить номер файла, но при этом сохранят OID. Не следует рассчитывать, что номер файлового узла и OID таблицы совпадают. Кроме того, для некоторых системных каталогов, включая и `pg_class`, в `pg_class.relfilenode` содержится ноль. Фактический номер файлового узла для них хранится в низкоуровневой структуре данных, и его можно получить при помощи функции `pg_relation_filenode()`.

Когда объём таблицы или индекса превышает 1 GB, они делятся на *сегменты* размером в один гигабайт. Файл первого сегмента называется по номеру файлового узла (filenode); последующие сегменты получают имена filenode.1, filenode.2 и т. д. При такой организации хранения не возникает проблем на платформах, имеющих ограничения по размеру файлов. (На самом деле, 1 ГБ — лишь размер по умолчанию. Размер сегмента можно изменить при сборке Postgres Pro, используя параметр конфигурации `--with-segsize`.) В принципе, карты свободного пространства и карты видимости также могут занимать нескольких сегментов, хотя на практике это маловероятно.

У таблицы, столбцы которой могут содержать данные большого объёма, будет иметься собственная таблица *TOAST*, предназначенная для отдельного хранения значений, которые слишком велики для хранения в строках самой таблицы. Основная таблица связывается с её таблицей TOAST (если таковая имеется) через `pg_class.reltoastrelid`. За подробной информацией обратитесь к Разделу 63.2.

Содержание таблиц и индексов рассматривается ниже (см. Раздел 63.6).

Табличное пространство делает сценарий более сложным. Каждое пользовательское табличное пространство имеет символическую ссылку внутри каталога `PGDATA/pg_tblspc`, указывающую на физический каталог табличного пространства (т. е., положение, указанное в команде табличного пространства `CREATE TABLESPACE`). Эта символическая ссылка получает имя по OID табличного пространства. Внутри физического каталога табличного пространства имеется подкаталог, имя которого зависит от версии сервера Postgres Pro, как например `PG_9.0_201008051`. (Этот подкаталог используется для того, чтобы последующие версии базы данных могли свободно использовать одно и то же местоположение, заданное в `CREATE TABLESPACE`.) Внутри каталога конкретной версии находится подкаталог для каждой базы данных, которая имеет элементы в табличном пространстве, названный по OID базы данных. Таблицы и индексы хранятся внутри этого каталога, используя схему именования файловых узлов. Табличное пространство `pg_default` недоступно через `pg_tblspc`, но соответствует `PGDATA/base`. Подобным же образом, табличное пространство `pg_global` недоступно через `pg_tblspc`, но соответствует `PGDATA/global`.

Функция `pg_relation_filepath()` показывает полный путь (относительно `PGDATA`) для любого отношения. Часто это избавляет от необходимости запоминать многие из приведённых выше правил. Но следует помнить, что эта функция выдаёт лишь имя первого сегмента основного слоя отношения, т. е. возможно, понадобится добавить номер сегмента и/или `_fsm`, `_vm` или `_init`, чтобы найти все файлы, связанные с отношением.

Временные файлы (для таких операций, как сортировка объёма данных большего, чем может уместиться в памяти) создаются внутри `PGDATA/base/pgsql_tmp` или внутри подкаталога `pgsql_tmp` каталога табличного пространства, если для них определено табличное пространство, отличное от `pg_default`. Имя временного файла имеет форму `pgsql_tmpPPP.NNN`, где *PPP* — PID серверного процесса, а *NNN* служит для разделения различных временных файлов этого серверного процесса.

63.2. TOAST

В данном разделе рассматривается TOAST (The Oversized-Attribute Storage Technique, Методика хранения сверхбольших атрибутов).

Postgres Pro использует фиксированный размер страницы (обычно 8 КБ), и не позволяет кортежам занимать несколько страниц. Поэтому непосредственно хранить очень большие значения полей невозможно. Для преодоления этого ограничения большие значения полей сжимаются и/или разбиваются на несколько физических строк. Это происходит незаметно для пользователя и на большую часть кода сервера влияет незначительно. Этот метод известен как TOAST (тост, или «лучшее после изобретения нарезанного хлеба»). Инфраструктура TOAST также применяется для оптимизации обработки больших значений данных в памяти.

Лишь определённые типы данных поддерживают TOAST — нет смысла производить дополнительные действия с типами данных, размер которых не может быть большим. Чтобы поддерживать TOAST, тип данных должен представлять значение переменной длины (*varlena*), в котором первое четырёхбайтовое слово любого хранящегося значения содержит общую длину значения в байтах (включая само это слово). Содержание оставшейся части значения TOAST не ограничивает. Специальные представления, в целом называемые *значениями в формате TOAST*, работают, манипулируя этим начальным словом длины и интерпретируя его по-своему. Таким образом, функции уровня C, работающие с типом данных, поддерживающим TOAST, должны аккуратно обращаться со входными значениями, которые могут быть в формате TOAST: входные данные могут и не содержать четырёхбайтовое слово длины и содержимое после него, пока не будут *распакованы*. (Обычно в таких ситуациях нужно использовать макрос `PG_DETOAST_DATUM` прежде чем что-либо делать с входным значением, но в некоторых случаях возможны и более эффективные подходы. За подробностями обратитесь к Подразделу 36.13.1.)

TOAST занимает два бита слова длины varlena (старшие биты на машинах с порядком байт от старшего к младшему, или младшие биты — при другом порядке байт), таким образом, логический размер любого значения в формате TOAST ограничивается 1 Гигабайтом (2^{30} - 1 байт). Когда оба бита равны нулю, значение является обычным, не в формате TOAST, и оставшиеся биты слова длины задают общий размер элемента данных (включая слово длины) в байтах. Когда установлен старший (или младший, в зависимости от архитектуры) бит, значение имеет однобайтовый заголовок вместо обычного четырёхбайтового, а оставшиеся биты этого байта задают общий размер элемента данных (включая байт длины) в байтах. Этот вариант позволяет экономно хранить значения короче 127 байт и при этом допускает расширение значения этого типа данных до 1 Гбайта при необходимости. Значения с однобайтовыми заголовками не выравниваются по какой-либо определённой границе, тогда как значения с четырёхбайтовыми заголовками выравниваются по границе минимум четырёх байт; это избавление от выравнивания даёт дополнительный выигрыш в объёме, очень ощутимый для коротких значений. В качестве особого случая, если все оставшиеся биты однобайтового заголовка равны нулю (что в принципе невозможно с учётом включения размера длины), значением является указатель на отдельно размещённые данные, с несколькими возможными вариантами, описанными ниже. Тип и размер такого *указателя TOAST* определяется кодом, хранящимся во втором байте значения. Наконец, когда старший (или младший, в зависимости от архитектуры) бит очищен, а соседний бит установлен, содержимое данных хранится в упакованном виде и должно быть распаковано перед использованием. В этом случае оставшиеся биты четырёхбайтового слова длины задают общий размер сжатых, а не исходных данных. Заметьте, что сжатие также возможно и для отделённых данных, но заголовок varlena не говорит, имеет ли оно место — это определяется содержимым, на которое указывает указатель TOAST.

Метод, который будет применяться для сжатия данных при внутреннем и внешнем хранении, можно выбрать для каждого отдельного столбца, задав параметр `COMPRESSION` в команде `CREATE TABLE` или `ALTER TABLE`. Если метод сжатия для столбца не задан явным образом, по умолчанию при вставке данных будет использоваться метод из параметра default_toast_compression.

Как уже было сказано, существуют разные варианты использования указателя TOAST. Самый старый и наиболее популярный вариант — когда он указывает на отделённые данные, размещённые в *TOAST-таблице*, которая отделена, но связана с таблицей, содержащей собственно указатель данных TOAST. Такой указатель на данные *на диске* создаётся кодом обработки TOAST, когда кортеж, сохраняемый на диск, оказывается слишком большим. Дополнительные подробности описаны в Подразделе 63.2.1. Кроме того, указатель TOAST может указывать на отделённые данные, размещённые где-то в памяти. Такие данные обязательно недолговременные и никогда не оказываются на диске, но этот механизм очень полезен для исключения копирования и избыточной обработки данных большого размера. Дополнительные подробности описаны в Подразделе 63.2.2.

63.2.1. Отдельное размещение TOAST на диске

Если какие-либо столбцы таблицы хранятся в формате TOAST, у таблицы будет связанная с ней таблица TOAST, OID которой хранится в значении `pg_class.reltoastrelid` для данной таблицы. Размещаемые на диске TOAST-значения содержатся в таблице TOAST, что подробнее описано ниже.

Отделённые значения делятся на порции (после сжатия, если оно применяется) размером не более `TOAST_MAX_CHUNK_SIZE` байт (по умолчанию это значение выбирается таким образом, чтобы на странице помещались четыре строки порций, то есть размер одной составляет порядка 2000 байт). Каждая порция хранится как отдельная строка в таблице TOAST, принадлежащей исходной таблице-владельцу. Каждая таблица TOAST имеет столбцы `chunk_id` (OID, идентифицирующий конкретное TOAST-значение), `chunk_seq` (последовательный номер для порции внутри значения) и `chunk_data` (фактические данные порции). Уникальный индекс по `chunk_id` и `chunk_seq` обеспечивает быструю выдачу значений. Таким образом, в указателе, представляющем отдельно размещаемое на диске значение TOAST, должно храниться OID таблицы TOAST, к которой нужно обращаться, и OID определённого значения (его `chunk_id`). Для удобства в данных указателя также хранится логический размер элемента данных (исходных данных без сжатия), фактический размер хранимых данных (отличающийся, если было применено сжатие) и используемый метод сжатия, если он задан. Учитывая байты заголовка varlena, общий размер указателя на хранимое на диске значение TOAST составляет 18 байт, независимо от фактического размера собственно значения.

Код обработки TOAST срабатывает, только когда значение строки, которое должно храниться в таблице, по размеру больше, чем `TOAST_TUPLE_THRESHOLD` байт (обычно это 2 Кб). Код TOAST будет сжимать и/или выносить значения поля за пределы таблицы до тех пор, пока значение строки не станет меньше `TOAST_TUPLE_TARGET` байт (переменная величина, так же обычно 2 Кб) или уменьшить объём станет невозможно. Во время операции UPDATE значения неизменённых полей обычно сохраняются как есть, поэтому модификация строки с отдельно хранимыми значениями не несёт издержек, связанных с TOAST, если все такие значения остаются без изменений.

Код обработки TOAST распознаёт четыре различные стратегии хранения столбцов, совместимых с TOAST, на диске:

- `PLAIN` не допускает ни сжатия, ни отдельного хранения. Это единственно возможная стратегия для столбцов типов данных, которые несовместимы с TOAST.
- `EXTENDED` допускает как сжатие, так и отдельное хранение. Это стандартный вариант для большинства типов данных, совместимых с TOAST. Сначала происходит попытка выполнить сжатие, затем — сохранение вне таблицы, если строка всё ещё слишком велика.
- `EXTERNAL` допускает отдельное хранение, но не сжатие. Использование `EXTERNAL` ускорит операции над частями строк в больших столбцах `text` и `bytea` (ценой увеличения объёма памяти для хранения), так как эти операции оптимизированы для извлечения только требуемых частей отделённого значения, когда оно не сжато.
- `MAIN` допускает сжатие, но не отдельное хранение. (Фактически для таких столбцов будет тем не менее применяться отдельное хранение, но лишь как крайняя мера, когда нет другого способа уменьшить строку так, чтобы она помещалась на странице.)

Каждый тип данных, совместимый с TOAST, определяет стандартную стратегию для столбцов этого типа данных, но стратегия для заданного столбца таблицы может быть изменена с помощью `ALTER TABLE ... SET STORAGE`.

`TOAST_TUPLE_TARGET` можно задавать на уровне таблиц с помощью команды `ALTER TABLE ... SET (toast_tuple_target = N)`

Эта схема имеет ряд преимуществ по сравнению с более простым подходом, когда значения строк могут занимать несколько страниц. Если предположить, что обычно запросы характеризуются выполнением сравнения с относительно маленькими значениями ключа, большая часть работы будет выполняться с использованием главной записи строки. Большие значения атрибутов в формате TOAST будут просто передаваться (если будут выбраны) в тот момент, когда результирующий набор отправляется клиенту. Таким образом, главная таблица получается гораздо меньше, и в общий кеш буферов помещается больше её строк, чем их было бы без использования отдельного хранения. Наборы данных для сортировок также уменьшаются, а сортировки чаще будут выполняться исключительно в памяти. Небольшой тест показал, что таблица, содержащая типичные HTML-страницы и их URL после сжатия занимала примерно половину объёма исходных данных, включая таблицу TOAST, и что главная таблица содержала лишь около 10% всех данных (URL и некоторые маленькие HTML-страницы). Время обработки не отличалось от времени, необходимого для обработки таблицы без использования TOAST, в которой размер всех HTML-страниц был уменьшен до 7 Кб, чтобы они уместились в строках.

63.2.2. Отдельное размещение TOAST в памяти

Указатели TOAST могут указывать на данные, размещённые не на диске, а где-либо в памяти текущего серверного процесса. Очевидно, что такие указатели не могут быть долговременными, но они тем не менее полезны. В настоящее время поддерживаются два подварианта: *косвенные* указатели на данные и указатели на *развёрнутые* данные.

Косвенный указатель TOAST просто указывает на значение varlena, хранящееся где-то в памяти. Этот вариант изначально был реализован просто как подтверждение концепции, но в настоящее время он применяется при логическом декодировании, чтобы не приходилось создавать физические кортежи больше одного 1 ГБ (что может потребоваться при консолидации всех отделённых значений полей в одном кортеже). Данный вариант имеет ограниченное применение, так как создатель такого указателя должен полностью понимать, что целевые данные будут существовать, только пока существует указатель, и никакой инфраструктуры для сохранения их нет.

Указатели на развёрнутые данные TOAST полезны для сложных типов, представление которых на диске плохо приспособлено для вычислительных целей. Например, стандартное представление в виде varlena массива Postgres Pro включает информацию о размерности, битовую карту элементов NULL (если они в нём содержатся), а затем значения всех элементов по порядку. Когда элемент сам по себе имеет переменную длину, единственный способ найти *N*-ый элемент — просканировать все предыдущие элементы. Это представление компактно и поэтому подходит для хранения на диске, но для вычислительной обработки массива гораздо удобнее иметь «развёрнутое» или «деконструированное» представление, в котором можно определить начальные адреса всех элементов. Механизм указателей TOAST способствует решению этой задачи, допуская передачу по ссылке элемента Datum как указателя на стандартное значение varlena (представление на диске) или указателя TOAST на развёрнутое представление где-то в памяти. Детали развёрнутого представление определяются самим типом данных, хотя оно может иметь стандартный заголовок и удовлетворять другим требованиям API, описанным в *`src/include/utils/expandeddatum.h`*. Функции уровня C, работающие с этим типом, могут реализовать поддержку любого из этих представлений. Функции, не знающие о развёрнутом представлении, а просто применяющие `PG_DETOAST_DATUM` к своим входным данным, будут автоматически получать традиционное представление varlena; так что поддержка развёрнутого представления может вводиться постепенно, по одной функции.

Указатели TOAST на развёрнутые значения далее подразделяются на указатели *для чтения/записи* и указатели *только для чтения*. Представление, на которое они указывают, в любом случае одинаковое, но функции, получающей указатель для чтения/записи, разрешается модифицировать целевые данные прямо на месте, тогда как функция, получающая указатель только для чтения, не должна этого делать; если ей нужно получить изменённую версию значения, она должна сначала сделать копию. Это отличие и связанные с ним соглашения позволяют избежать излишнего копирования развёрнутых значений при выполнении запросов.

Для всех типов указателей TOAST на данные в памяти, код обработки TOAST гарантирует, что такие данные не окажутся случайно сохранены на диске. Указатели TOAST в памяти автоматически сворачиваются в обычные значения varlena перед сохранением — а затем могут преобразоваться в указатели TOAST на диске, если без этого не смогут уместиться в содержащем их кортеже.

63.2.3. Подключаемый механизм TOAST

Механизм TOAST является частью ядра Postgres Pro. Он использует одну стратегию для всех типов данных, не является расширяемым и неэффективен в работе со структурированными данными (JSON) или данными, требующими специальной обработки (`bytea`).

Экспериментальный подключаемый механизм TOAST предоставляет открытый API, позволяющий разрабатывать и использовать собственные реализации TOAST для столбцов таблиц и типов данных в дополнение к стандартной реализации. Реализации TOAST могут обрабатывать данные TOAST, используя знания о внутренней структуре данных и рабочем процессе.

Реализации TOAST могут использовать внутри любой тип хранилища, сложные алгоритмы сжатия, защитное преобразование и прочую функциональность, вне зависимости от метода доступа к таблице, который вызывает эту реализацию.

Подключаемый механизм TOAST состоит из ядра, то есть API TOAST, и настраиваемых реализаций TOAST, подключаемых к Postgres Pro через этот API.

Этот API никак не связан с логикой функциональности TOAST, а является просто обёрткой, позволяющей регистрировать, вызывать и удалять подключённые реализации TOAST.

Чтобы разрешить применение пользовательских реализаций TOAST, API TOAST определяет новый тип `Custom` указателя TOAST в дополнение к `External` и `Extended` (соответствует стратегиям `EXTERNAL` и `EXTENDED` хранения столбцов, совместимых с TOAST).

В настоящее время подключаемый механизм TOAST имеет несколько ограничений:

- Пользовательские реализации TOAST должны назначаться только для столбцов со стратегией хранения `EXTERNAL`.
- В настоящее время подключаемый механизм TOAST не поддерживает логическую репликацию. Пользовательский указатель TOAST пропускается механизмом логической репликации и не реплицируется. Это связано с тем, что поддержка пользовательских указателей требует серьёзных изменений в ядре Postgres Pro (механизме логической репликации) и даже изменения архитектуры, поскольку теперь пользовательские указатели позволяют изменять значения в формате TOAST, а указатели по умолчанию — нет.
- Пользовательские реализации TOAST не восстанавливаются из копий и требуют восстановления вручную. Расширения пользовательских реализаций TOAST не восстанавливаются из копий, поэтому обработанные ими данные недоступны. Ещё одна особенность заключается в том, что одно и то же расширение пользовательской реализации TOAST имеет разные идентификаторы в разных системах, поэтому для восстановления файла TOAST из копии потребуется определить пользовательскую реализацию TOAST, используемую для обработки данных.
- Когда столбцу таблицы назначается пользовательская реализация TOAST, во всех транзакциях после этого назначения данные, хранящиеся в TOAST-таблицах, обрабатываются реализацией, назначенной этому столбцу.
- Когда столбцу таблицы назначается пользовательская реализация TOAST, старые значения остаются неизменными, но при изменении старого значения оно сохраняется в формате TOAST, используя назначенную реализацию, из-за особенной работы механизма TOAST по умолчанию.
- Если реализация TOAST удалена, невозможно обратиться стандартными методами к данным, преобразованным с её помощью в формат TOAST, и они будут считаться потерянными. Чтобы защитить значения в формате TOAST, API создаёт в системе некоторые зависимости (таблица `PG_DEPEND`):
  - При добавлении реализации TOAST в систему создаётся зависимость реализации TOAST от расширения.
  - При назначении реализации TOAST столбцу создаётся зависимость реализации TOAST от отношения.
  - При удалении реализации TOAST, назначенной столбцу, удаляется зависимость реализации TOAST от отношения.
  - При удалении реализации TOAST из системы удаляется зависимость реализации TOAST от расширения.

Пользовательские реализации TOAST добавляются как расширения, выполняющие некоторую обязательную функциональность (за дополнительными сведениями обратитесь к файлу `README.toastapi` в каталоге `/contrib/toastapi`). Подключаемый механизм TOAST содержит набор функций SQL для управления пользовательскими реализациями TOAST. Эти функции вызываются командой SQL `SELECT`:

`add_toaster(`*`toaster_name`*` text, `*`toaster_handler_func`*` text)` → `integer`

Подключить новую реализацию TOAST (добавленную ранее как расширение). Эта функция также создаёт зависимость в `PG_DEPEND` для OID реализации TOAST от OID её расширения. Эта зависимость используется, чтобы реализация TOAST не удалялась при удалении расширения этой реализации.

Аргументы:

- *`toaster_name`* — имя реализации TOAST, которое будет храниться в `PG_TOASTER`. Должно быть уникальным.
- *`toaster_handler_func`* — имя функции-обработчика реализации TOAST (автоматически добавляется как `regproc` при установке расширения реализации TOAST).

Возвращает OID, присвоенный новой реализации TOAST.

`set_toaster(`*`toaster_name`*` text, `*`tab_name`*` text, `*`col_name`*` text)` → `integer`

Назначить столбцу таблицы реализацию TOAST. Эта функция также создаёт зависимость в `PG_DEPEND` для OID реализации TOAST от OID отношения. Эта зависимость используется для защиты данных в формате TOAST путём ограничения на удаление реализации TOAST, назначенной столбцу.

Аргументы:

- *`toaster_name`* — имя реализации TOAST, назначенной столбцу таблицы.
- *`tab_name`* — имя таблицы.
- *`col_name`* — имя столбца.

Возвращает OID реализации TOAST, назначенной столбцу.

`reset_toaster(`*`tab_name`*` text, `*`col_name`*` text)` → `integer`

Сбросить назначенную столбцу таблицы реализацию TOAST и назначить механизм TOAST по умолчанию. Эта функция удаляет зависимость от отношения реализации TOAST, что позволяет затем удалить реализацию TOAST при удалении всех зависимостей.

Аргументы:

- *`tab_name`* — имя таблицы.
- *`col_name`* — имя столбца таблицы.

Всегда возвращает 0.

`drop_toaster(`*`toaster_name`*` text)`

Удалить реализацию TOAST из системы. Можно удалять только неиспользуемые реализации TOAST, не имеющие данных в формате TOAST. Удаление защищено зависимостями, создаваемыми функциями `add_toaster` и `set_toaster`. Если зависимость всё ещё существует, реализация TOAST не будет удалена. Если реализация успешно удалена, расширение с этой реализацией можно удалить командой `DROP EXTENSION`.

Аргументы:

- *`toaster_name`* — имя удаляемой реализации TOAST.

Возвращает OID реализации TOAST в случае успеха или 0, если реализация TOAST используется.

`get_toaster(tab_name text, col_name text) → integer`

Получить OID реализации TOAST, назначенной столбцу таблицы.

Аргументы:

- *`tab_name`* — имя таблицы.
- *`col_name`* — имя столбца таблицы.

Возвращает OID реализации TOAST, назначенной столбцу, или 0, если она не назначена.

В следующем примере показано использование подключаемого API TOAST:

```
CREATE EXTENSION toastapi;

CREATE EXTENSION bytea_toaster;

SELECT add_toaster('bytea_toaster');

CREATE TABLE test_bytea_append (id int, a bytea);

ALTER TABLE test_bytea_append ALTER a SET STORAGE external;

SELECT set_toaster('bytea_toaster', 'test_bytea_append', 'a');
 test_set_toaster
------------------

(1 row)

...

SELECT get_toaster('test_bytea_append', 'a') AS bytea_toaster_oid;
 get_toaster
-------------
        16348
(1 row)


SELECT pgpro_toast.reset_toaster('test_bytea_append','a');
 reset_toaster
---------------
             0
(1 row)

DROP TABLE test_bytea_append;
SELECT pgpro_toast.drop_toaster('bytea_toaster');
 drop_toaster
--------------
         16348
(1 row)

DROP EXTENSION bytea_toaster;
```

63.3. Карта свободного пространства

Каждое табличное и индексное отношение, за исключением хеш-индексов, имеет карту свободного пространства (Free Space Map, FSM) для отслеживания доступного места. Она хранится рядом с данными главного отношения в отдельном слое, имя которого образуется номером файлового узла отношения с суффиксом `_fsm`. Например, если файловый узел отношения — 12345, FSM хранится в файле с именем `12345_fsm` в том же каталоге, что и основной файл отношения.

Карта свободного пространства представляет собой дерево страниц FSM. Страницы FSM нижнего уровня хранят информацию о свободном пространстве, доступном на каждой странице таблицы (или индекса), используя один байт для представления каждой такой страницы. Верхние уровни агрегируют информацию нижних уровней.

Внутри каждой страницы FSM имеется двоичное дерево, хранящееся в массиве, где один байт выделяется на каждый узел дерева. Каждый листовой узел представляет страницу таблицы или страницу FSM нижнего уровня. В каждом узле выше листовых хранится наибольшее из значений его узлов-потомков. Поэтому максимальное из значений листовых узлов хранится в корневом узле.

Для просмотра информации, хранящейся в картах свободного пространства, можно воспользоваться модулем pg_freespacemap.

63.4. Карта видимости

Каждое отношение таблицы имеет карту видимости (Visibility Map, VM) для отслеживания страниц, содержащих только кортежи, которые видны всем активным транзакциям; в ней также отслеживается, какие страницы содержат только замороженные кортежи. Она хранится вместе с данными главного отношения в отдельном файле, имя которого образуется номером файлового узла отношения с суффиксом `_vm`. Например, если файловый узел отношения — 12345, VM хранится в файле `12345_vm`, в том же самом каталоге, что и основной файл отношения. Заметьте, что индексы не имеют VM.

Карта видимости хранит по два бита на страницу таблицы. Первый бит, если он установлен, показывает, что вся страница видна или, другими словами, не содержит кортежей, которые необходимо очистить. Эта информация может также использоваться при *сканировании только индекса* для поиска ответов только в данных индекса. Установленный второй бит показывает, что все кортежи на этой странице заморожены. Это означает, что процесс очистки для предотвращения зацикливания не должен больше посещать эту страницу.

Карта может отражать реальные данные с запаздыванием в том смысле, что мы уверены, что в случаях, когда установлен бит, известно, что условие верно, но если бит не установлен, оно может быть верным или неверным. Биты карты видимости устанавливаются только при очистке, а сбрасываются при любых операциях, изменяющих данные на странице.

Для изучения информации, хранящейся в карте видимости, можно воспользоваться модулем pg_visibility.

63.5. Слой инициализации

Каждая нежурналируемая таблица, и каждый индекс такой таблицы имеет файл инициализации. Файл инициализации представляет собой пустую таблицу или индекс соответствующего типа. Когда нежурналируемая таблица должна быть заново очищена по причине сбоя, файл инициализации копируется поверх главного файла, а все прочие файлы удаляются (при необходимости они будут автоматически созданы заново).

63.6. Компоновка страницы базы данных

В данном разделе рассматривается формат страницы, используемый в таблицах и индексах Postgres Pro.[1] Последовательности и таблицы TOAST форматируются как обычные таблицы.

В дальнейшем подразумевается, что *байт* содержит 8 бит. В дополнение, термин *элемент* относится к индивидуальному значению данных, которое хранится на странице. В таблице элемент — это строка; в индексе — элемент индекса.

Каждая таблица и индекс хранятся как массив *страниц* фиксированного размера (обычно 8 kB, хотя можно выбрать другой размер страницы при компиляции сервера). В таблице все страницы

---

[1] На самом деле этот формат страниц не является обязательным ни для табличных, ни для индексных методов доступа. Его всегда использует табличный метод `heap` и все существующие индексные методы, но в метастраницах индексов данные обычно компонуются по другим правилам.

логически эквивалентны, поэтому конкретный элемент (строка) может храниться на любой странице. В индексах первая страница обычно резервируется как *метастраница*, хранящая контрольную информацию, а внутри индекса могут быть разные типы страниц, в зависимости от метода доступа индекса.

Таблица 63.2 показывает общую компоновку страницы. Каждая страница имеет пять частей.

Таблица 63.2. Общая компоновка страницы

Элемент	Описание
Данные заголовка страницы	Длина — 24 байта. Содержит общую информацию о странице, включая указатели свободного пространства.
Данные идентификаторов элементов	Массив идентификаторов, указывающих на фактические элементы. Каждый идентификатор представляет собой пару «смещение, длина» и занимает 4 байта.
Свободное пространство	Незанятое пространство. Новые идентификаторы элементов размещаются с начала этой области, сами новые элементы — с конца.
Элементы	Сами элементы данных как таковые.
Специальное пространство	Специфические данные метода доступа. Для различных методов хранятся различные данные. Для обычных таблиц таких данных нет.

Первые 24 байта каждой страницы образуют заголовок страницы (`PageHeaderData`). Его формат подробно описан в Таблице 63.3. В первом поле отслеживается самая последняя запись в WAL, связанная с этой страницей. Второе поле содержит контрольную сумму страницы, если включён режим Контрольные суммы данных. Затем идёт двухбайтовое поле, содержащее биты флагов. За ним следуют три двухбайтовых целочисленных поля (`pd_lower`, `pd_upper` и `pd_special`). Они содержат смещения в байтах от начала страницы до начала незанятого пространства, до конца незанятого пространства и до начала специального пространства. В следующих 2 байтах заголовка страницы, в поле `pd_pagesize_version`, хранится размер страницы и индикатор версии. Начиная с PostgreSQL 8.3, используется версия 4; в PostgreSQL 8.1 и 8.2 использовалась версия 3; в PostgreSQL 8.0 — версия 2; в PostgreSQL 7.3 и 7.4 — версия 1; в предыдущих выпусках — версия 0. (Основная структура страницы и формат заголовка почти во всех этих версиях одни и те же, но структура заголовка строк в куче изменялась.) Размер страницы присутствует в основном только для перекрёстной проверки; возможность использовать в одной инсталляции разные размеры страниц не поддерживается. Последнее поле подсказывает, насколько вероятна возможность получить выигрыш, произведя очистку страницы: оно отслеживает самый старый XMAX на странице, не подвергавшийся очистке.

Таблица 63.3. Данные заголовка страницы (PageHeaderData)

Поле	Тип	Длина	Описание
pd_lsn	PageXLogRecPtr	8 байт	LSN: Следующий байт после последнего байта записи WAL для последнего изменения на этой странице
pd_checksum	uint16	2 байта	Контрольная сумма страницы
pd_flags	uint16	2 байта	Биты признаков
pd_lower	LocationIndex	2 байта	Смещение до начала свободного пространства

Поле	**Тип**	**Длина**	**Описание**
pd_upper	LocationIndex	2 байта	Смещение до конца свободного пространства
pd_special	LocationIndex	2 байта	Смещение до начала специального пространства
pd_pagesize_version	uint16	2 байта	Информация о размере страницы и номере версии компоновки
pd_prune_xid	TransactionId	4 байта	Самый старый неочищенный идентификатор XMAX на странице или ноль при отсутствии такового

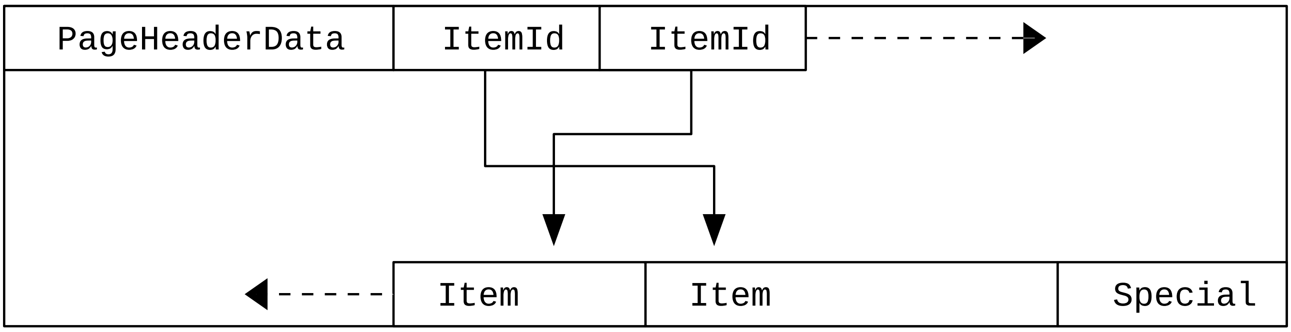
За заголовком страницы следуют идентификаторы элемента (`ItemIdData`), каждому из которых требуется 4 байта. Идентификатор элемента содержит байтовое смещение до начала элемента, его длину в байтах и несколько битов атрибутов, которые влияют на его интерпретацию. Новые идентификаторы элементов размещаются по мере необходимости от начала свободного пространства. Количество имеющихся идентификаторов элементов можно определить через значение `pd_lower`, которое увеличивается при добавлении нового идентификатора. Поскольку идентификатор элемента никогда не перемещается до тех пор, пока он не освобождается, его индекс можно использовать в течение длительного периода времени, чтобы ссылаться на элемент, даже когда сам элемент перемещается по странице для уплотнения свободного пространства. Фактически каждый указатель на элемент (`ItemPointer`, также известный как `CTID`), созданный Postgres Pro, состоит из номера страницы и индекса идентификатора элемента.

Сами элементы хранятся в пространстве, выделяемом в направлении от конца к началу незанятого пространства. Точная структура меняется в зависимости от того, каким будет содержимое таблицы. Как таблицы, так и последовательности используют структуру под названием `HeapTupleHeaderData`, которая описывается ниже.

Последний раздел является «особым разделом», который может содержать всё, что необходимо методу доступа для хранения. Например, индексы-B-деревья хранят ссылки на страницы слева и справа, равно как и некоторые другие данные, соответствующие структуре индекса. Обычные таблицы не используют особый раздел вовсе (что указывается установкой значения `pd_special` равным размеру страницы).

Рисунок 63.1 показывает, как эти компоненты размещаются в странице.

Рисунок 63.1. Компоновка страницы



63.6.1. Компоновка строки таблицы

Все строки таблицы имеют одинаковую структуру. Они включают заголовок фиксированного размера (занимающий 23 байта на большинстве машин), за которым следует необязательная битовая карта пустых значений, необязательное поле идентификатора объекта и данные пользователя. Подробное описание заголовка представлено в Таблице 63.4. Актуальные пользовательские данные (столбцы строки) начинаются после смещения, заданного в `t_hoff`, которое должно всегда быть кратным величине MAXALIGN для платформы. Битовая карта пустых значений имеется тогда, когда бит *HEAP_HASNULL* установлен в значении `t_infomask`. В случае наличия она начинается сразу после фиксированного заголовка и занимает столько байтов, сколько требуется для размещения битов по количеству столбцов (т. е. число битов равно количеству атрибутов, определяемому полем `t_infomask2`). В этом списке битов установленный бит означает непустое значение, а сброшенный соответствует пустому значению. Когда битовая карта отсутствует, все столбцы считаются непустыми. Идентификатор объекта присутствует, если только в значении `t_infomask` установлен бит *HEAP_HASOID_OLD*. Если он есть, он расположен сразу перед началом `t_hoff`. Любое заполнение, необходимое для того, чтобы сделать `t_hoff` кратным MAXALIGN, будет добавлено между битовой картой пустых значений и идентификатором объекта. (Это в свою очередь гарантирует, что идентификатор объекта будет правильно выровнен.)

Таблица 63.4. Данные заголовка строки таблицы (HeapTupleHeaderData)

Поле	**Тип**	**Длина**	**Описание**
t_xmin	TransactionId	4 байта	значение XID вставки
t_xmax	TransactionId	4 байта	значение XID удаления
t_cid	CommandId	4 байта	значение CID для вставки и/или удаления (пересекается с t_xvac)
t_xvac	TransactionId	4 байта	XID для операции VACUUM, которая перемещает версию строки
t_ctid	ItemPointerData	6 байт	текущее значение TID этой или более новой версии строки
t_infomask2	uint16	2 байта	количество атрибутов плюс различные биты флагов
t_infomask	uint16	2 байта	различные биты флагов
t_hoff	uint8	1 байт	отступ до пользовательских данных

Интерпретировать текущие данные можно только с использованием информации, полученной из других таблиц, в основном из `pg_attribute`. Ключевыми значениями, необходимыми для определения расположения полей, являются `attlen` и `attalign`. Не существует способа непосредственного получения заданного атрибута, кроме случая, когда имеются только поля фиксированной длины и при этом нет значений NULL. Все эти особенности учитываются в функциях *heap_getattr*, *fastgetattr* и *heap_getsysattr*.

Чтобы прочитать данные, необходимо просмотреть каждый атрибут по очереди. В первую очередь нужно проверить, является ли значение поля пустым согласно битовой карте пустых значений. Если это так, можно переходить к следующему полю. Затем следует убедиться, что выравнивание является верным. Если это поле фиксированной ширины, берутся просто все его байты. Если это поле переменной длины (attlen = -1), всё несколько сложнее. Все типы данных с переменной длиной имеют общую структуру заголовка `struct varlena`, которая включает общую длину сохранённого значения и некоторые биты флагов. В зависимости от установленных флагов, данные могут храниться либо локально, либо в таблице TOAST. Также, возможно сжатие данных (см. Раздел 63.2).

63.7. Кортежи только в куче (Heap-Only Tuples, HOT)

Для обеспечения высокой степени параллельности Postgres Pro сохраняет строки, реализуя многоверсионное управление конкурентным доступом (MVCC). Однако механизм MVCC работает не очень эффективно, когда происходит изменение данных. В частности, при изменении строк в таблицы должны добавляться их новые версии. Кроме того, для каждой изменённой строки в индексы должны добавляться новые элементы, а удаление старых версий строк и элементов индекса может повлечь дополнительные издержки.

Для снижения издержек, связанных с изменениями данных, в Postgres Pro реализована оптимизация, называемая «кортежи только в куче» (HOT). Её применение возможно в следующих условиях:

- При изменении не меняются столбцы, на которые ссылаются индексы таблицы, за исключением индексов сводных данных. Единственный метод, определяющий поведение сводных значений в базовом дистрибутиве Postgres Pro, — BRIN.
- На странице, содержащей старую версию строки, достаточно свободного места для новой.

При выполнении этих условий оптимизация HOT даёт два преимущества:

- Для представления изменённых строк не нужны новые элементы индекса, однако они могут потребоваться для индексов сводных данных.
- При многократном изменении строки её версии, кроме первой и последней, могут быть полностью удалены в ходе обычной работы, даже во время `SELECT`, для этого не нужно периодически выполнять операции очистки. (Индексы всегда ссылаются на идентификатор элемента страницы исходной версии строки. Данные кортежей, связанных с этой версией строки, удаляются, а её идентификатор элемента преобразуется в ссылку, указывающую на первую версию строки, которая может быть по-прежнему видна параллельной транзакции. Промежуточные версии строки, которые больше никому не видны, полностью удаляются, а связанные идентификаторы элементов страниц становятся доступны для повторного использования.)

Чтобы на страницах было достаточно места для HOT-кортежей, можно уменьшить `fillfactor` (фактор заполнения) таблицы. Даже если это не делать, механизм HOT всё равно будет работать, поскольку новые строки будут естественным образом переноситься на новые страницы и существующие страницы, где достаточно свободного места для новых версий строк. За изменениями в режиме HOT и в обычном режиме позволяет наблюдать системное представление pg_stat_all_tables.

Глава 64. Обработка транзакций

В этой главе представлен обзор внутреннего устройства системы управления транзакциями Postgres Pro. В английском языке слово «transaction» зачастую сокращается до *xact*.

64.1. Транзакции и идентификаторы

Транзакции могут создаваться явным образом командой `BEGIN` или `START TRANSACTION` и завершаться командой `COMMIT` или `ROLLBACK`. SQL-операторы вне явного блока транзакций автоматически обрабатываются в виде транзакций, состоящих из одного оператора.

Каждой транзакции присваивается уникальный идентификатор `VirtualTransactionId` (также именуемый `virtualXID` или `vxid`), который состоит из идентификатора обслуживающего процесса (или `procNumber`) и последовательно назначаемого номера — внутреннего для такого обслуживающего процесса (или `localXID`). Например, виртуальный идентификатор `4/12532` состоит из следующих компонентов: `procNumber` со значением `4` и `localXID` со значением `12532`.

Невиртуальные идентификаторы `TransactionId` (или `xid`), например `278394`, последовательно выбираются для транзакций из глобального счётчика, который используется всеми базами данных в рамках кластера Postgres Pro. Значение присваивается при первой операции записи транзакции в базу данных. Это означает, что транзакции с меньшими xid начинают запись раньше транзакций с большими xid. Обратите внимание, что порядок, в котором транзакции выполняют запись в базу данных впервые, может отличаться от порядка, в котором они запускаются, особенно если транзакции начинаются с операторов, выполняющих только операции чтения.

Внутренний тип идентификаторов транзакций `xid` имеет размер 32 бита, и значения в нём повторяются через каждые 4 миллиарда транзакций. После каждого цикла 32-битная эпоха увеличивается на 1. Существует также и 64-битный тип `xid8`, который включает эту эпоху и поэтому не повторяется на протяжении жизни сервера. Его можно преобразовать в `xid` посредством операции приведения типа. Функции в Таблица 9.82 возвращают значения `xid8`. Идентификаторы используются в качестве основы механизма многоверсионного управления конкурентным доступом MVCC Postgres Pro.

Когда транзакция верхнего уровня с (невиртуальным) xid фиксируется, она помечается как зафиксированная в каталоге `pg_xact`. Дополнительная информация записывается в каталог `pg_commit_ts` при включённом параметре track_commit_timestamp.

Помимо того, что с подготовленными транзакциями сопоставляются определённые значения `vxid` и `xid`, им также назначаются глобальные идентификаторы транзакций (GID). GID представляет собой строковый литерал длиной до 200 байт, который должен иметь уникальное значение среди других существующих подготовленных транзакций. Сопоставление GID с идентификаторами транзакций представлено в `pg_prepared_xacts`.

64.2. Транзакции и блокировки

Идентификаторы выполняемых транзакций выводятся в представлении `pg_locks`, в столбцах `virtualxid` и `transactionid`. Для транзакций в режиме только чтения выводится только `virtualxid`, а в поле `transactionid` будет NULL, тогда как для транзакций чтения-записи выводятся оба идентификатора.

Некоторые типы блокировок ожидают `virtualxid`, другие типы ожидают `transactionid`. Информация о блокировках чтения и записи на уровне строк регистрируется непосредственно в заблокированных строках, её можно прочитать при помощи расширения pgrowlocks. Блокировки чтения на уровне строк также могут потребовать присвоения идентификаторов мультитранзакций (`mxid`; см. Подраздел 23.1.5.1).

64.3. Подтранзакции

Подтранзакции запускаются внутри транзакций и позволяют разбивать более крупные транзакции на составляющие. Подтранзакции могут фиксироваться или прерываться, не влияя на родительские транзакции, которые, соответственно, могут продолжать выполняться. Такое решение позволяет обрабатывать ошибки проще и часто используется при разработке приложений. В английском языке слово подтранзакция зачастую сокращается до *subxact*.

Подтранзакции могут явным образом запускаться как при помощи команды `SAVEPOINT`, так и другими способами, например посредством предложения `EXCEPTION` языка PL/pgSQL. Языки PL/Python и PL/Tcl также поддерживают явные подтранзакции. Подтранзакции можно также запускать в других подтранзакциях. Транзакция верхнего уровня и её дочерние подтранзакции формируют иерархию или дерево, поэтому по отношению к основной транзакции используется термин «транзакция верхнего уровня».

Если подтранзакции присваивается невиртуальный идентификатор, его называют «subxid». Подтранзакциям в режиме только чтения не присваиваются subxid, однако при первой попытке записи такой идентификатор будет присвоен. При этом всем родительским транзакциям подтранзакции, вплоть до транзакции верхнего уровня включительно, присваивается невиртуальный идентификатор. Значение xid родительской транзакции всегда будет меньше значений subxid подтранзакций.

Идентификатор непосредственного родителя каждой подтранзакции записывается в каталог `pg_subtrans`. Идентификаторы транзакций верхнего уровня не записываются, поскольку у них нет родителя. Также не записываются и идентификаторы подтранзакций в режиме только чтения.

При фиксации подтранзакции все зафиксированные дочерние подтранзакции с subxid также считаются зафиксированными в рамках этой подтранзакции. При прерывании подтранзакции все дочерние подтранзакции также считаются прерванными.

При фиксации транзакции верхнего уровня с xid зафиксированные подтранзакции записываются как зафиксированные в подкаталоге `pg_xact`. При прерывании транзакции верхнего уровня все её подтранзакции также прерываются, даже если они были зафиксированы.

Чем больше подтранзакций остаётся открытыми в каждой транзакции (в отношении которых не выполнен откат или освобождение), тем выше будут издержки, связанные с управлением транзакциями. До 64 открытых subxid кешируется в общей памяти для каждого сервера. После этого этапа издержки хранения для операций ввода-вывода существенно возрастают из-за дополнительных операций поиска записей о subxid в `pg_subtrans`.

64.4. Двухфазные транзакции

Postgres Pro поддерживает протокол двухфазной фиксации (2PC), позволяющий нескольким распределённым системам работать в транзакционной манере. В рамках этого протокола используются следующие команды: `PREPARE TRANSACTION`, `COMMIT PREPARED` и `ROLLBACK PREPARED`. Двухфазные транзакции предназначены для использования внешними системами управления транзакциями. Postgres Pro придерживается особенностей и модели, предложенных в рамках стандарта X/Open XA, однако не применяет некоторые редко используемые аспекты.

Когда пользователь выполняет команду `PREPARE TRANSACTION` для подготовки транзакции, следующим шагом он может выполнить только команду `COMMIT PREPARED` или `ROLLBACK PREPARED`. Как правило, транзакции могут находиться в таком подготовленном состоянии очень непродолжительное время, однако ввиду проблем внешней доступности они могут сохранять такое состояние несколько дольше. Подготовленные транзакции с коротким временем жизни хранятся только в общей памяти и журнале WAL. Транзакции, которые проходят контрольные точки, записываются в каталог `pg_twophase`. Подготовленные транзакции можно просмотреть в представлении `pg_prepared_xacts`.

Глава 65. Как планировщик использует статистику

Данная глава основана на материалах, рассмотренных ранее (см. Раздел 14.1 и Раздел 14.2), и подробнее рассказывает о том, как планировщик использует статистику для определения количества строк, которое может вернуть каждая часть запроса. Это важная составляющая процесса создания плана запроса, предоставляющая большую часть исходного материала для расчёта стоимости.

Целью данной главы является не подробное документирование кода, а общее описание его работы. Возможно, это поможет тем, кто пожелает в дальнейшем ознакомиться с кодом.

65.1. Примеры оценки количества строк

В приведённых ниже примерах используются таблицы базы данных регрессионного тестирования Postgres Pro. Заметьте также, что поскольку команда `ANALYZE` использует случайную выборку при формировании статистики, после любого нового выполнения команды `ANALYZE` результаты незначительно изменятся.

Давайте начнём с очень простого запроса:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Как планировщик определяет мощность `tenk1`, рассматривается выше (см. Раздел 14.2), но для полноты здесь говорится об этом ещё раз. Количество страниц и строк берётся в `pg_class`:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 |     10000
```

Это текущие цифры, полученные при последнем выполнении команд `VACUUM` или `ANALYZE`, применённых к этой таблице. Затем планировщик выполняет выборку фактического текущего числа страниц в таблице (это недорогая операция, для которой не требуется сканирование таблицы). Если оно отличается от `relpages`, то `reltuples` изменяется для того, чтобы привести это значение к текущей оценке количества строк. В показанном выше примере значение `relpages` является актуальным, поэтому количество строк берётся равным `reltuples`.

Давайте обратимся к примеру с диапазонным условием в предложении `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Планировщик рассматривает условие предложения `WHERE` и находит в справочнике функцию избирательности для оператора `<` в `pg_operator`. Это значение содержится в столбце `oprrest`, и в данном случае значением является `scalarltsel`. Функция `scalarltsel` извлекает гистограмму для `unique1` из `pg_statistic`. Для вводимых вручную запросов удобнее просматривать более простое представление `pg_stats`:

```
SELECT histogram_bounds FROM pg_stats
```

```
WHERE tablename='tenk1' AND attname='unique1';

                   histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

Затем обрабатывается часть гистограммы, которая соответствует условию «< 1000». Таким образом и определяется избирательность. Гистограмма делит диапазон на равные частотные группы, поэтому нужно лишь определить группу, содержащую наше значение, и подсчитать её *долю* и *долю групп*, предшествующих данной. Очевидно, что значение 1000 находится во второй группе (993–1997). Если предположить, что внутри каждой группы распределение значений линейное, мы можем вычислить избирательность следующим образом:

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

т. е. сумма элементов одной целой группы и пропорциональной части элементов второй, делённая на число групп. Теперь примерное число строк может быть рассчитано как произведение избирательности и мощности `tenk1`:

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (округлённо)
```

Далее, давайте рассмотрим пример с условием на равенство в предложении `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

Планировщик вновь проверяет условие в предложении `WHERE` и определяет функцию избирательности для `=`, и этой функцией является `eqsel`. Для оценки равенства гистограмма бесполезна, вместо неё для оценки избирательности используется список *самых частых значений* (Most Common Values, MCV). Давайте рассмотрим MCV и соответствующие дополнительные столбцы, которые пригодятся позже:

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,
JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

Так как значение `CRAAAA` оказалось в списке MCV, избирательность будет определяться просто соответствующим элементом в списке частот самых частых значений (Most Common Frequencies, MCF):

```
selectivity = mcf[3]
            = 0.003
```

Как и в предыдущем примере, оценка числа строк берётся как произведение мощности и избирательности `tenk1`:

```
rows = 10000 * 0.003
     = 30
```

Теперь рассмотрим тот же самый запрос, но с константой, которой нет в списке MCV:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

Это совершенно другая задача — как оценить избирательность значения, которого *нет* в списке MCV. При её решении используется факт отсутствия данного значения в списке в сочетании с частотой для каждого значения из списка MCV.

```
selectivity = (1 - sum(mcv_freqs))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

Т. е. нужно сложить частоты значений из списка MCV, отнять полученное число от единицы, и полученное значение разделить на количество *остальных* уникальных значений. Эти вычисления основаны на предположении, что значения, которые не входят в список MCV, имеют равномерное распределение. Заметьте, что в данном примере нет неопределённых значений, поэтому о них беспокоиться не нужно (иначе их долю также пришлось бы вычитать из числителя). Оценка числа строк затем производится как обычно:

```
rows = 10000 * 0.0014559
     = 15  (округлённо)
```

Предыдущий пример с `unique1 < 1000` был большим упрощением того, что в действительности делает `scalarltsel`. Но после того, как мы увидели пример использования списка MCV, мы можем внести некоторые дополнения. Что касается самого примера, в нём все было правильно, поскольку `unique1` — это уникальный столбец, у него нет значений в списке MCV (очевидно, в данном случае нет значения, которое встречается чаще, чем какое-либо другое). Для неуникального столбца обычно создаётся как гистограмма, так и список MCV, при этом *гистограмма не включает значения, представленные в списке MCV*. Данный способ позволяет выполнить более точный подсчёт. В этой ситуации `scalarltsel` напрямую применяет условие «< 1000» к каждому значению списка MCV и суммирует частоты значений MCV, для которых условие является верным. Это даёт точную оценку избирательности для той части таблицы, которая содержит значения из списка MCV. Подобным же образом используется гистограмма для оценки избирательности для той части таблицы, которая не содержит значения из списка MCV, а затем эти две цифры складываются для оценки общей избирательности. Например, рассмотрим

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

Мы уже видели данные списка MCV для `stringu1`, а это его гистограмма:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
--------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

Проверяя список MCV, находим, что условие `stringu1 < 'IAAAAA'` соответствует первым шести записям, но не соответствует последним четырём, поэтому избирательность для значений, соответствующих значениям в списке MCV, такова:

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

Сумма всех частот из списка MCF также сообщает нам, что общая часть представленной списком MCV совокупности записей равняется 0.03033333, и поэтому представленная гистограммой часть равняется 0.96966667 (в этом случае тоже нет неопределённых значений, иначе их пришлось бы также исключить). Видно, что значение `IAAAAA` попадает почти в конец третьего столбца гистограммы. Основываясь на простых предположениях относительно частоты различных символов, планировщик получает число 0.298387 для части значений, представленных в гистограмме, которые меньше чем `IAAAAA`. Затем объединяем оценки части значений из списка MCV и значений, не содержащихся в нём:

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (округлённо)
```

В этом конкретном примере, корректировка со стороны списка MCV достаточно мала, потому что распределение значений столбца довольно плоское (статистика, показывающая конкретные значения как более распространённые, чаще всего получается вследствие статистической погрешности). В более типичном случае, когда некоторые значения являются значительно более распространёнными по сравнению с другими, этот более сложный метод повышает точность вследствие точного определения избирательности наиболее распространённых значений.

Теперь давайте рассмотрим случай с более чем одним условием в предложении `WHERE`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                   QUERY PLAN
--------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Планировщик исходит из того, что два условия независимы, таким образом, отдельные значения избирательности можно перемножить:

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (округлённо)
```

Заметьте, что число строк, которые предполагается вернуть через сканирование битового индекса, соответствует условию, используемому при работе индекса; это важно, так как влияет на оценку стоимости для последующих выборок из таблицы.

В заключение исследуем запрос, выполняющий соединение:

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1 width=244)
```

```
         Index Cond: (unique2 = t1.unique2)
```

Ограничение, накладываемое на `tenk1`, `unique1 < 50`, производится до соединения вложенным циклом. Это обрабатывается аналогично предыдущему примеру с диапазонным условием. На этот раз значение 50 попадает в первый столбец гистограммы `unique1`:

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (округлённо)
```

Ограничение для соединения следующее `t2.unique2 = t1.unique2`. Здесь используется уже известный нам оператор `=`, однако функцию избирательности получаем из столбца `oprjoin` представления `pg_operator`, и эта функция — `eqjoinsel`. Функция `eqjoinsel` находит статистические данные как для `tenk2`, так и для `tenk1`:

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

В этом случае нет данных MCV для `unique2`, и все значения будут уникальными (n_distinct = -1). Таким образом, используется алгоритм, зависящий от оценки числа строк для обеих таблиц (показывается не `num_rows`, а `tenk`) и от данных с неопределёнными значениями столбца (ноль для обеих таблиц):

```
selectivity = (1 - null_frac1) * (1 - null_frac2) / max(num_rows1, num_rows2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

То есть вычитаем долю неопределённых значений из единицы для каждой таблицы и делим на число строк более крупной таблицы (это значение увеличивается в неуникальном случае). Количество строк, которое соединение, вероятно, сгенерирует, вычисляется как мощность декартова произведения двух входных значений, умноженная на избирательность:

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

Если бы имелись списки MCV для двух столбцов, функцией `eqjoinsel` использовалось бы прямое сравнение со списками MCV для определения общей избирательности той части данных, которая содержит значения списка MCV. Оценка остальной части данных при этом выполнялась бы представленным выше способом.

Заметьте, что здесь выводится для `inner_cardinality` значение 10000, то есть исходный размер `tenk2`. Если изучить вывод `EXPLAIN`, может показаться, что оценка количества строк вычисляется как 50 * 1, то есть число внешних строк умножается на ориентировочное число строк, получаемых при каждом внутреннем сканировании индекса в `tenk2`. Но это не так, ведь размер результата соединения оценивается до того, как выбирается конкретный план соединения. Если всё работает корректно, оба варианта вычисления этого размера должны давать один и тот же ответ, но из-за ошибок округления и других факторов иногда они значительно различаются.

65.2. Примеры многовариантной статистики

65.2.1. Функциональные зависимости

Многовариантную корреляцию можно продемонстрировать на очень простом наборе данных — таблице с двумя столбцами, содержащими одинаковые значения:

```
CREATE TABLE t (a INT, b INT);
INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1, 10000) s(i);
ANALYZE t;
```

Как рассказывается в Разделе 14.2, планировщик может определить мощность `t`, исходя из числа страниц и строк, полученного из `pg_class`:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 't';

 relpages | reltuples
----------+-----------
       45 |     10000
```

Распределение данных очень простое: в каждом столбце содержится всего 100 различных значений, равномерно распределённых.

Следующий пример показывает результат оценивания условия `WHERE` по столбцу `a`:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1;
                                 QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: (a = 1)
   Rows Removed by Filter: 9900
```

Планировщик рассматривает условие и определяет, что его избирательность равна 1%. Сравнивая эту оценку и фактическое число строк, мы видим, что оценка очень точна (на самом деле абсолютна точна, так как таблица очень маленькая). Если изменить условие `WHERE`, чтобы использовался столбец `b`, будет получен такой же план. Но посмотрите, что получится, если мы применим одинаковое условие к двум столбцам, объединив их оператором `AND`:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

Планировщик оценивает избирательность каждого условия индивидуально, и получает ту же оценку в 1%, что и выше. Затем он предполагает, что условия независимы, так что он перемножает избирательности и выдаёт окончательную оценку избирательности, равную всего 0.01%. Это значительная недооценка, так как фактическое число строк, соответствующих условию, (100) на два порядка больше.

Эту проблему можно решить, создав объект статистики, который укажет команде `ANALYZE` вычислить многовариантную статистику функциональной зависимости по двум столбцам:

```
CREATE STATISTICS stts (dependencies) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

65.2.2. Многовариантное число различных значений

Подобная проблема возникает с оценкой мощности наборов с несколькими столбцами, например, с оценкой числа групп, которые могут быть выданы предложением `GROUP BY`. Когда в `GROUP BY` указан один столбец, оценка числа различных значений (которую можно увидеть как ожидаемое число строк, выдаваемое узлом HashAggregate) очень точная:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a;
                                       QUERY PLAN
--------------------------------------------------------------------
----------------------
 HashAggregate  (cost=195.00..196.00 rows=100 width=12) (actual rows=100 loops=1)
   Group Key: a
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=4) (actual rows=10000
 loops=1)
```

Но оценка числа групп в запросе с двумя столбцами в `GROUP BY` без многовариантной статистики, как и в предыдущем примере, отличается от правильной на порядок:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                       QUERY PLAN
--------------------------------------------------------------------
-------------------------
 HashAggregate  (cost=220.00..230.00 rows=1000 width=16) (actual rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000
 loops=1)
```

Если переопределить объект статистики, чтобы он включал подсчёт числа различных значений для двух столбцов, оценка станет гораздо лучше:

```
DROP STATISTICS stts;
CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                       QUERY PLAN
--------------------------------------------------------------------
-------------------------
 HashAggregate  (cost=220.00..221.00 rows=100 width=16) (actual rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8) (actual rows=10000
 loops=1)
```

65.2.3. Списки MCV

Как рассказывалось в Подразделе 65.2.1, статистика функциональных зависимостей очень недорога и эффективна, но имеет серьёзное ограничение, связанное с её природой (отслеживаются только зависимости на уровне столбцов, но не между отдельными значениями столбцов).

В этом разделе описываются многовариантные списки MCV (Most-Common Values, Самые частые значения), естественное расширение статистики по столбцам, описанной в Разделе 65.1. Списки MCV устраняют упомянутое ограничение, храня отдельные значения, что, разумеется, усложняет построение статистики в ходе `ANALYZE`, а также снижает скорость планирования и эффективность хранения.

Взгляните ещё раз на запрос, фигурировавший в Подразделе 65.2.1, но на этот раз со списком MCV, созданным по тем же столбцам (функциональные зависимости нужно удалить, чтобы планировщик мог использовать только новую статистику).

```
DROP STATISTICS stts;
CREATE STATISTICS stts2 (mcv) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 1;
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

Оценка оказывается такой же точной, как и с функциональными зависимостями, во многом благодаря тому, что таблица достаточно мала и характеризуется простым распределением с небольшим количеством уникальных значений. Прежде чем перейти ко второму запросу, в котором функциональные зависимости показали себя не очень хорошо, давайте просмотрим список MCV.

Просмотреть список MCV можно с помощью возвращающей множество функции `pg_mcv_list_items`.

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname = 'stts2';
 index |  values  | nulls | frequency | base_frequency
-------+----------+-------+-----------+----------------
     0 | {0, 0}   | {f,f} |      0.01 |         0.0001
     1 | {1, 1}   | {f,f} |      0.01 |         0.0001
   ...
    49 | {49, 49} | {f,f} |      0.01 |         0.0001
    50 | {50, 50} | {f,f} |      0.01 |         0.0001
   ...
    97 | {97, 97} | {f,f} |      0.01 |         0.0001
    98 | {98, 98} | {f,f} |      0.01 |         0.0001
    99 | {99, 99} | {f,f} |      0.01 |         0.0001
(100 rows)
```

Полученная информация подтверждает, что два столбца содержат 100 уникальных комбинаций значений и что эти столбцы скорее всего равны (частота каждой комбинации — 1%). Базовая частота вычисляется, исходя из частот значений отдельных столбцов, без учёта наличия статистики по нескольким столбцам. Если бы в одном из столбцов оказались значения NULL, это было бы отражено в столбце `nulls`.

Оценивая избирательность, планировщик применяет все условия к элементам списка MCV и суммирует частоты тех, которые этим условиям удовлетворяют.

В сравнении с функциональными зависимостями списки MCV имеют два важных преимущества. Во-первых, в этих списках хранятся фактические значения, что позволяет определить, какие комбинации являются совместимыми.

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b = 10;
                                 QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=0 loops=1)
   Filter: ((a = 1) AND (b = 10))
   Rows Removed by Filter: 10000
```

Во-вторых, списки MCV подходят для более широкого ассортимента условий, а не только для сравнений, как функциональные зависимости. Например, взгляните на запрос с диапазоном значений, выполняемый с той же таблицей:

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a <= 49 AND b > 49;
                                QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=0 loops=1)
   Filter: ((a <= 49) AND (b > 49))
   Rows Removed by Filter: 10000
```

65.3. Статистика планировщика и безопасность

Доступ к таблице `pg_statistic` разрешён только суперпользователям, так что обычные пользователи не могут получить из неё сведения о содержимом таблиц других пользователей. Но некоторые функции оценки избирательности будут использовать пользовательский оператор (оператор, фигурирующий в запросе, или связанный) для анализа сохранённой статистики. Например, чтобы определить применимость сохранённого самого частого значения, функция оценки избирательности должна задействовать соответствующий оператор `=` для сравнения константы в запросе с этим сохранённым значением. Таким образом, данные `pg_statistic` в принципе могут передаваться пользовательским операторам. А особым образом сконструированный оператор может выводить наружу передаваемые ему операнды преднамеренно (например, записывая их в журнал или помещая в другую таблицу) либо непреднамеренно (показывая их значения в сообщениях об ошибках). В любом случае это даёт возможность пользователю, не имеющему доступа к таблице `pg_statistic`, увидеть содержащиеся в ней данные.

Для предотвращения этого все встроенные функции оценки избирательности действуют по следующим правилам. Чтобы сохранённая статистика могла использоваться при планировании запроса, текущий пользователь должен иметь либо право `SELECT` для таблицы или задействованных столбцов, либо у оператора должна быть характеристика `LEAKPROOF` (точнее, она должна быть у функции, реализующей этот оператор). В противном случае оценка избирательности будет осуществляться так, как если бы статистики не было вовсе, и планировщик продолжит работу с общими или вторичными предположениями.

Если пользователь не имеет требуемого права доступа к таблице или столбцам, то во многих случаях при выполнении запроса в конце концов возникнет ошибка «нет доступа», так что этот механизм будет незаметен на практике. Но если пользователь читает данные из представления с барьером безопасности, планировщик может захотеть проверить статистику нижележащей таблицы, которая недоступна пользователю непосредственно. В этом случае оператор должен быть герметичным; иначе статистика не будет использоваться. Это не будет иметь внешних проявлений кроме того, что план запроса может быть неоптимальным. В случае подозрений, что вы столкнулись с этим, попробуйте запустить запрос от имени пользователя с расширенными правами и проверьте, не выбирается ли другой план запроса.

Это ограничение применяется только тогда, когда планировщику может потребоваться выполнить пользовательский оператор с одним или несколькими значениями из `pg_statistic`. При этом планировщику разрешено использовать общую статистическую информацию, например, процент значений NULL или количество различных значений в столбце, вне зависимости от прав доступа.

Реализуемые в дополнительных расширениях функции оценки избирательности, которые могут обращаться к статистике, вызывая пользовательские операторы, должны следовать тем же правилам безопасности.

Глава 66. Формат манифеста копии

Манифест копии формируется программой pg_basebackup и предназначен в первую очередь для того, чтобы копию можно было проверить с помощью pg_verifybackup. Однако и другие программные средства могут прочитать файл манифеста и использовать содержащуюся в нём информацию для своих целей. В данной главе описывается формат файла манифеста.

Манифест копии представляет собой документ JSON в кодировке в UTF-8. (Хотя сам стандарт JSON требует, чтобы документы были представлены в Unicode, Postgres Pro позволяет представлять значения `json` и `jsonb` в любой поддерживаемой сервером кодировке. Для манифестов копий такое исключение не делается.) Этот документ JSON всегда содержит один объект; ключи этого объекта описаны в следующем разделе.

66.1. Объект верхнего уровня в манифесте

Документ JSON, представляющий манифест копии, содержит следующие ключи.

`PostgreSQL-Backup-Manifest-Version`

Связанное с этим ключом значение является целым числом. Начиная с Postgres Pro `17`, это `2`, в предыдущих версиях — `1`.

`System-Identifier`

Идентификатор базы данных экземпляра Postgres Pro, из которого была создана резервная копия. Это поле присутствует, только если `PostgreSQL-Backup-Manifest-Version` имеет значение `2`.

`Files`

С этим ключом всегда связан список объектов, каждый из которых описывает файл, имеющийся в копии. В этом списке отсутствуют записи о файлах WAL, требующихся для использования копии, а также о самом манифесте. Структура каждого объекта в этом списке описана в Разделе 66.2.

`WAL-Ranges`

С этим ключом всегда связан список объектов, каждый из которых описывает диапазон записей WAL, которые необходимо прочитать для определённой линии времени, чтобы можно было восстановить эту копию. Структура этих объектов подробнее описана в Разделе 66.3.

`Manifest-Checksum`

Этот ключ всегда находится в последней строке файла манифеста. Связанное с ним значение содержит контрольную сумму всех предыдущих строк, вычисленную по алгоритму SHA256. Здесь используется фиксированный алгоритм, чтобы клиенты могли разобрать манифест последовательно. Алгоритм SHA256 имеет большую вычислительную сложность, чем CRC32C, но манифест обычно имеет достаточно скромный размер, так что эти дополнительные вычисления не должны играть большой роли.

66.2. Объект файла в манифесте

Этот объект описывает один файл и обычно содержит ключ `Path` либо может содержать ключ `Encoded-Path`. Связанное с этим ключом значение задаёт путь к файлу относительно корневого каталога копии. Для файлов, расположенных в пользовательских табличных пространствах, первыми компонентами пути будут `pg_tblspc` и OID табличного пространства. Если путь задаётся строкой не в кодировке UTF-8 или пользователь выбрал вариант кодирования всех путей файлов, вместо ключа `Path` присутствует `Encoded-Path`. Он имеет то же содержимое, но закодированное в виде строки шестнадцатеричных цифр, каждая пара которых представляет один байт.

Всегда присутствуют следующие ключи:

`Size`

Ожидаемый размер файла, в виде целого числа.

`Last-Modified`

Дата последнего изменения файла, полученная на сервере во время создания копии. В отличие от других полей, хранящихся в манифесте, это поле не используется программой pg_verifybackup. Оно добавлено исключительно для информации.

Если копия была сделана с контрольными суммами, будут присутствовать следующие ключи:

`Checksum-Algorithm`

Алгоритм, который применялся для подсчёта контрольной суммы этого файла. В настоящее время алгоритм будет одинаковым для всех файлов в манифесте копии, но в будущих выпусках это может измениться. На данный момент поддерживаются алгоритмы `CRC32C`, `SHA224`, `SHA256`, `SHA384` и `SHA512`.

`Checksum`

Контрольная сумма этого файла, представленная в виде последовательности шестнадцатеричных цифр, по две на каждый байт значения.

66.3. Объект диапазона WAL в манифесте копии

Объект, описывающий диапазон WAL, всегда содержит три ключа:

`Timeline`

Линия времени, к которой относится этот диапазон записей WAL, в виде целого числа.

`Start-LSN`

Позиция LSN, с которой должно начаться воспроизведение на указанной линии времени, чтобы эту копию можно было восстановить. LSN хранится в формате, принятом в Postgres Pro; то есть в виде двух строк, разделённых косой чертой и содержащих от 1 до 6 шестнадцатеричных символов.

`End-LSN`

Самый первый LSN, на котором может закончиться воспроизведение на указанной линии времени при восстановлении этой копии. Он хранится в том же формате, что и `Start-LSN`.

Обычно в манифесте описывается всего один диапазон WAL. Но если копия была сделана с ведомого сервера, переключившего линии времени во время повышения, в манифесте могут присутствовать несколько диапазонов, относящихся к разным линиям времени. К одной линии времени может относиться только один диапазон WAL.

Часть VIII. Приложения

Приложение A. Коды ошибок Postgres Pro

Всем сообщениям, которые выдаёт сервер Postgres Pro, назначены пятисимвольные коды ошибок, соответствующие кодам «SQLSTATE», описанным в стандарте SQL. Приложения, которые должны знать, какое условие ошибки имело место, обычно проверяют код ошибки и только потом обращаются к текстовому сообщению об ошибке. Коды ошибок, скорее всего, не изменятся от выпуска к выпуску Postgres Pro, и они не меняются при локализации как сообщения об ошибках. Заметьте, что отдельные, но не все коды ошибок, которые выдаёт Postgres Pro, определены стандартом SQL; некоторые дополнительные коды ошибок для условий, не описанных стандартом, были добавлены независимо или позаимствованы из других баз данных.

Согласно стандарту, первые два символа кода ошибки обозначают класс ошибок, а последние три символа обозначают определённое условие в этом классе. Таким образом, приложение, не знающее значение определённого кода ошибки, всё же может понять, что делать, по классу ошибки.

В Таблице A.1 перечислены все коды ошибок, определённые в Postgres Pro Standard 17.6.1. (Некоторые коды в настоящее время не используются, хотя они определены в стандарте SQL.) Также показаны классы ошибок. Для каждого класса ошибок имеется «стандартный» код ошибки с последними тремя символами `000`. Этот код выдаётся только для таких условий ошибок, которые относятся к некоторому классу, но не имеют более определённого кода.

Символ, указанный в столбце «Имя условия», определяет условие в PL/pgSQL. Имена условий могут записываться в верхнем или нижнем регистре. (Заметьте, что PL/pgSQL, в отличие от ошибок, не распознаёт предупреждения; то есть классы 00, 01 и 02.)

Для некоторых типов ошибок сервер сообщает имя объекта базы данных (таблица, столбец таблицы, тип данных или ограничение), связанного с ошибкой; например, имя уникального ограничения, вызвавшего ошибку `unique_violation`. Такие имена передаются в отдельных полях сообщения об ошибке, чтобы приложениям не пришлось извлекать его из возможно локализованного текста ошибки для человека. На момент выхода PostgreSQL 9.3 полностью охватывались только ошибки класса SQLSTATE 23 (нарушения ограничений целостности), но в будущем должны быть охвачены и другие классы.

Таблица A.1. Коды ошибок Postgres Pro

Код ошибки	Имя условия
Класс 00 — Успешное завершение	
`00000`	`successful_completion`
Класс 01 — Предупреждение	
`01000`	`warning`
`0100C`	`dynamic_result_sets_returned`
`01008`	`implicit_zero_bit_padding`
`01003`	`null_value_eliminated_in_set_function`
`01007`	`privilege_not_granted`
`01006`	`privilege_not_revoked`
`01004`	`string_data_right_truncation`
`01P01`	`deprecated_feature`
Класс 02 — Нет данных (это также класс предупреждений согласно стандарту SQL)	
`02000`	`no_data`
`02001`	`no_additional_dynamic_result_sets_returned`

Код ошибки	**Имя условия**
Класс 03 — SQL-оператор ещё не завершён	
`03000`	`sql_statement_not_yet_complete`
Класс 08 — Исключение, связанное с подключением	
`08000`	`connection_exception`
`08003`	`connection_does_not_exist`
`08006`	`connection_failure`
`08001`	`sqlclient_unable_to_establish_sqlconnection`
`08004`	`sqlserver_rejected_establishment_of_sqlconnection`
`08007`	`transaction_resolution_unknown`
`08P01`	`protocol_violation`
Класс 09 — Исключение с действием триггера	
`09000`	`triggered_action_exception`
Класс 0A — Неподдерживаемая функциональность	
`0A000`	`feature_not_supported`
Класс 0B — Неверное начало транзакции	
`0B000`	`invalid_transaction_initiation`
Класс 0F — Исключение с указателем на данные	
`0F000`	`locator_exception`
`0F001`	`invalid_locator_specification`
Класс 0L — Неверный праводатель	
`0L000`	`invalid_grantor`
`0LP01`	`invalid_grant_operation`
Класс 0P — Неверное указание роли	
`0P000`	`invalid_role_specification`
Класс 0Z — Исключение диагностики	
`0Z000`	`diagnostics_exception`
`0Z002`	`stacked_diagnostics_accessed_without_active_handler`
Класс 20 — Case не найден	
`20000`	`case_not_found`
Класс 21 — Нарушение количества	
`21000`	`cardinality_violation`
Класс 22 — Исключение в данных	
`22000`	`data_exception`
`2202E`	`array_subscript_error`
`22021`	`character_not_in_repertoire`
`22008`	`datetime_field_overflow`
`22012`	`division_by_zero`
`22005`	`error_in_assignment`
`2200B`	`escape_character_conflict`

Код ошибки	Имя условия
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
22013	invalid_preceding_or_following_size
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character

Код ошибки	**Имя условия**
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
22030	duplicate_json_object_key_value
22031	invalid_argument_for_sql_json_datetime_function
22032	invalid_json_text
22033	invalid_sql_json_subscript
22034	more_than_one_sql_json_item
22035	no_sql_json_item
22036	non_numeric_sql_json_item
22037	non_unique_keys_in_a_json_object
22038	singleton_sql_json_item_required
22039	sql_json_array_not_found
2203A	sql_json_member_not_found
2203B	sql_json_number_not_found
2203C	sql_json_object_not_found
2203D	too_many_json_array_elements
2203E	too_many_json_object_members
2203F	sql_json_scalar_required
2203G	sql_json_item_cannot_be_cast_to_target_type
Класс 23 — Нарушение ограничения целостности	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Класс 24 — Неверное состояние курсора	
24000	invalid_cursor_state
Класс 25 — Неверное состояние транзакции	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction

Код ошибки	Имя условия
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
25P04	transaction_timeout
Класс 26 — Неверное имя SQL-оператора	
26000	invalid_sql_statement_name
Класс 27 — Нарушение при изменении данных в триггере	
27000	triggered_data_change_violation
Класс 28 — Неверное указание авторизации	
28000	invalid_authorization_specification
28P01	invalid_password
Класс 2B — Зависимые описания привилегий всё ещё существуют	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Класс 2D — Неверное завершение транзакции	
2D000	invalid_transaction_termination
Класс 2F — Исключение в подпрограмме SQL	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Класс 34 — Неверное имя курсора	
34000	invalid_cursor_name
Класс 38 — Исключение во внешней подпрограмме	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Класс 39 — Исключение при вызове внешней подпрограммы	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated

Код ошибки	**Имя условия**
39P03	event_trigger_protocol_violated
Класс 3B — Исключение точки сохранения	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Класс 3D — Неверное имя каталога	
3D000	invalid_catalog_name
Класс 3F — Неверное имя схемы	
3F000	invalid_schema_name
Класс 40 — Откат транзакции	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Класс 42 — Ошибка синтаксиса или нарушение правила доступа	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor

Код ошибки	**Имя условия**
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Класс 44 — Нарушение WITH CHECK OPTION	
44000	with_check_option_violation
Класс 53 — Нехватка ресурсов	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
53500	cardinality_estimation_accuracy
Класс 54 — Превышение ограничения программы	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Класс 55 — Объект не в требуемом состоянии	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available

Код ошибки	**Имя условия**
55P04	unsafe_new_enum_value_usage
Класс 57 — Вмешательство оператора	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
57P05	idle_session_timeout
Класс 58 — Ошибка системы (ошибка, внешняя по отношению к PostgreSQL)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Класс F0 — Ошибка файла конфигурации	
F0000	config_file_error
F0001	lock_file_exists
Класс HV — Ошибка обёртки сторонних данных (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found

Код ошибки	**Имя условия**
`HV00K`	`fdw_reply_handle`
`HV00Q`	`fdw_schema_not_found`
`HV00R`	`fdw_table_not_found`
`HV00L`	`fdw_unable_to_create_execution`
`HV00M`	`fdw_unable_to_create_reply`
`HV00N`	`fdw_unable_to_establish_connection`
Класс P0 — Ошибка PL/pgSQL	
`P0000`	`plpgsql_error`
`P0001`	`raise_exception`
`P0002`	`no_data_found`
`P0003`	`too_many_rows`
`P0004`	`assert_failure`
Класс XX — Внутренняя ошибка	
`XX000`	`internal_error`
`XX001`	`data_corrupted`
`XX002`	`index_corrupted`

Приложение B. Поддержка даты и времени

Postgres Pro использует внутренний эвристический анализатор для поддержки всех значений даты и времени. Дата и время вводятся как строки и разделяются на различные поля, при этом предварительно определяется, какого рода информация содержится в конкретном поле. Каждое поле интерпретируется и получает числовое значение, игнорируется или отклоняется. Анализатор содержит внутренние справочные таблицы для текстовых полей, включая месяцы, дни недели и часовые пояса.

Данное приложение включает информацию о содержании справочных таблиц и описывает этапы, необходимые анализатору для распознавания даты и времени.

B.1. Интерпретация данных даты и времени

Строки с датой/временем разбираются при вводе по следующему алгоритму.

1. Разделить входную строку на фрагменты и определить каждый фрагмент как строку, время, часовой пояс или цифру.
    a. Если числовой фрагмент содержит двоеточие (`:`), значит эта строка представляет время. Включаются все последующие цифры и двоеточия.
    b. Если числовой фрагмент содержит тире (`-`), косую черту (`/`) или две и более точек (`.`), то это строка даты, которая, возможно, включает название месяца. Если фрагмент даты уже встречался, он интерпретируется как название часового пояса (например, `America/New_York`).
    c. Если этот фрагмент является лишь числом, он представляет собой отдельное поле или составную дату ISO 8601 (например, `19990113` для 13 января 1999 года) или время (например, `141516` для 14:15:16).
    d. Если фрагмент начинается с плюса (`+`) или минуса (`-`), то это или числовой часовой пояс или специальное поле.
2. Если фрагмент содержит только буквы, сопоставить его с возможными строками:
    a. Проверить, не совпадает ли фрагмент с известной аббревиатурой часового пояса. Эти аббревиатуры считываются из файла конфигурации, описанного в Разделе B.4.
    b. Если фрагмент не найден, проверить во внутренней таблице, не совпадает ли он со специальной строкой (например, `today`), днём недели (например, `Thursday`), месяцем (например, `January`) или игнорируемым словом (например, `at`, `on`).
    c. Если фрагмент всё же не найден, выдать ошибку.
3. Когда фрагмент является числом или числовым полем:
    a. Если получено восемь или шесть цифр и никакое другое поле даты ранее не было прочитано, интерпретировать их как «составленную дату» (например, `19990118` или `990118`). Такая дата интерпретируется как `ГГГГММДД` или `ГГММДД`.
    b. Если фрагмент представляет собой трёхзначное число, и год уже был прочитан, интерпретировать как день года.
    c. Если это четыре или шесть цифр и год уже был прочитан, интерпретировать как время (`ЧЧММ` или `ЧЧММСС`).
    d. Если найдены три или более цифр, а поля даты ещё не были найдены, интерпретировать как год (это ведёт к установке порядка гг-мм-дд для оставшихся полей даты).

   e. В противном случае подразумевается, что порядок сортировки полей даты определяется значением `DateStyle`: мм-дд-гг, дд-мм-гг или гг-мм-дд. Выдать ошибку, если оказалось, что поле месяца или дня вышло за пределы диапазона.

4. Если указан год до н. э., отнять год и добавить единицу для внутреннего хранения. (В григорианском календаре отсутствует нулевой год, поэтому 1 год до н. э. становится нулевым.)
5. Если год до н. э. не был указан, и если поле года имело два разряда, установить для записи года четыре разряда. Если поле меньше 70, добавить 2000, в противном случае добавить 1900.

Подсказка

Годы 1–99 н. э. по григорианскому календарю могут вводиться в виде четырёхзначного числа с начальными нулями (например, `0099` — 99 год н. э.).

B.2. Обработка недопустимых или неоднозначных значений даты/времени

Обычно, если строка даты/времени синтаксически корректна, но содержит поля со значениями вне допустимого диапазона, выдаётся ошибка. Например, не будет принята строка с числом 31 февраля.

Однако с учётом перевода часов на летнее время визуально корректная строка с датой/временем может указывать не несуществующий или неоднозначный момент времени. Такие значения тем не менее принимаются; для разрешения неоднозначности учитывается смещение от UTC. Например, в предположении, что значение TimeZone — `America/New_York`, рассмотрите следующий пример:

```
=> SELECT '2018-03-11 02:30'::timestamptz;
      timestamptz
------------------------
 2018-03-11 03:30:00-04
(1 row)
```

Так как в этот день в данном часовом поясе стрелки часов переводились вперёд, по гражданским часам не существовал момент времени 2:30; они перескочили с 2:00 (EST) на 3:00 (EDT). Postgres Pro воспринимает заданное время, как если бы оно было стандартным временем (в часовом поясе UTC-5), в результате чего оно преобразуется в 3:30AM (в часовом поясе EDT или UTC-4).

И наоборот, рассмотрите поведение в случае перевода времени назад:

```
=> SELECT '2018-11-04 01:30'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-05
(1 row)
```

В этот день возможны две интерпретации времени 1:30; сначала было 1:30 в часовом поясе EDT, а час спустя, после перевода часов назад с 2:00 на 1:00 и смены часового пояса с EDT на EST, наступило 1:30 в часовом поясе EST. Так же и в этом случае Postgres Pro интерпретирует заданное время, как если бы оно было стандартным (UTC-5). Мы можем выбрать другой момент, указав часовой пояс летнего времени:

```
=> SELECT '2018-11-04 01:30 EDT'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-04
(1 row)
```

Точное правило, применяемое в таких случаях, звучит так: несуществующий момент времени, попадающий в интервал перевода часов вперёд, привязывается к смещению от UTC, действовавшему в данном часовом поясе до перевода, а неоднозначный момент времени, попадающий в оба интервала при переводе часов назад, привязывается к смещению от UTC, действующему после перевода. Для большинства часовых поясов это равносильно правилу «в случае неясности предпочитать интерпретацию со стандартным временем».

В любом случае смещение от UTC, связанное с меткой времени, можно задать явно, добавив либо числовое смещение, либо аббревиатуру часового пояса, которой соответствует некоторое фиксированное смещение. Вышеприведённое правило применяется, только когда необходимо вычислить смещение от UTC для часового пояса, в котором оно меняется.

B.3. Ключевые слова для обозначения даты и времени

Таблица B.1 показывает фрагменты, которые распознаются как названия месяцев.

Таблица B.1. Названия месяцев

Месяц	**Аббревиатуры**
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep, Sept
October	Oct
November	Nov
December	Dec

Таблица B.2 показывает фрагменты, которые распознаются как названия дней недели.

Таблица B.2. Названия дней недели

День	**Аббревиатуры**
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Таблица B.3 показывает фрагменты, которые выполняют различные функции модификаторов.

Таблица B.3. Модификаторы поля даты/времени

Идентификатор	**Описание**
`AM`	Время до 12:00
`AT`	Игнорируется
`JULIAN`, `JD`, `J`	Следующее поле является юлианской датой

Идентификатор	**Описание**
`ON`	Игнорируется
`PM`	Время 12:00 и более позднее
`T`	Следующее поле указывает на время

B.4. Файлы конфигурации даты/времени

Поскольку аббревиатуры часовых поясов недостаточно стандартизированы, Postgres Pro предлагает средства для определения набора аббревиатур, принимаемых сервером. Параметром выполнения timezone_abbreviations определяется активный набор аббревиатур. Хотя данный параметр может быть изменён любым пользователем базы данных, возможные значения для него контролируются администратором базы данных и являются именами конфигурационных файлов, хранящихся в `.../share/timezonesets/` каталога установки. Добавляя или изменяя файлы в этом каталоге, администратор может определить местную специфику выбора аббревиатур часовых поясов.

Значение `timezone_abbreviations` может быть установлено в любое имя файла, находящегося в `.../share/timezonesets/`, если имя файла состоит только из букв. (Запрет на использование небуквенных символов в `timezone_abbreviations` делает невозможным чтение файлов, находящихся вне заданного каталога, а также резервных файлов редактора и прочих внешних файлов.)

Файл аббревиатур часовых поясов может содержать пустые строки и комментарии, начинающиеся с `#`. Строки, не имеющие комментариев, должны иметь один из следующих форматов:

```
аббревиатура_пояса смещение
аббревиатура_пояса смещение D
аббревиатура_пояса имя_часового_пояса
@INCLUDE имя_файла
@OVERRIDE
```

Поле *`аббревиатура_пояса`* лишь задаёт определяемую аббревиатуру. *`Смещение`* — это целое число, задающее эквивалентное смещение от UTC в секундах, положительное — к востоку от Гринвичского меридиана, а отрицательное — к западу. Например, -18000 означало бы пять часов к западу от Гринвича, или Североамериканское восточное время. `D` указывает, что название пояса представляет местное летнее, а не поясное время.

В качестве альтернативы может быть задано *`имя_часового_пояса`*, определённое в базе данных часовых поясов IANA. В этом случае система обращается к определению пояса, чтобы выяснить, используется или использовалась ли аббревиатура для этого пояса, и если да, действовать будет соответствующее значение — то есть, значение, действовавшее в указанный момент времени, или действовавшее непосредственно перед ним, если текущее на тот момент неизвестно, либо самое старое значение, если первое определение появилось после этого момента. Это поведение важно для тех аббревиатур, значение которых менялось в ходе истории. Также допускается определение аббревиатуры через имя часового пояса, с которым эта аббревиатура не связана; в таком случае использование аббревиатуры равнозначно написанию просто имени пояса.

Подсказка

Простое целочисленное *`смещение`* предпочтительнее использовать, когда определяется аббревиатура, смещение которой от UTC никогда не менялось, так как обрабатывать подобные аббревиатуры гораздо легче, чем те, что требуют обращения к определению часового пояса.

Использование `@INCLUDE` позволяет включить другой файл в каталоге `.../share/timezonesets/`. Включение может быть вложенным до ограниченной глубины.

Использование `@OVERRIDE` указывает, что последующие записи в файле могут переопределять предыдущие (как правило, это записи, полученные из включённых файлов). Без этого указания конфликтующие определения аббревиатуры одного и того же часового пояса считаются ошибкой.

При установке без внесения изменений, файл `Default` содержит все неконфликтующие аббревиатуры часовых поясов для большей части земного шара. Дополнительные файлы `Australia` и `India` предоставляются для данных регионов. Эти файлы сначала включают файл `Default`, а затем добавляют и изменяют аббревиатуры по мере необходимости.

В качестве справочной информации стандартная установка также содержит файлы `Africa.txt`, `America.txt` и т. д., содержащие информацию о каждой используемой аббревиатуре часового пояса, включённой в базу данных часовых поясов IANA. Определения названий часовых поясов, находящиеся в этих файлах, можно копировать и помещать в файл с нестандартной конфигурацией по мере необходимости. Заметьте, что данные файлы нельзя указывать непосредственно в значении `timezone_abbreviations`, так как их имена включают точку.

Примечание

Если при чтении набора аббревиатур часовых поясов возникает ошибка, новое значение не применяется и сохраняется старый набор. Если ошибка возникает при запуске базы данных, происходит сбой.

Внимание

Аббревиатуры часового пояса, определённые в файле конфигурации, переопределяют не относящиеся к часовому поясу значения, встроенные в Postgres Pro. Например, файл конфигурации `Australia` определяет `SAT` (для Южноавстралийского стандартного времени, South Australian Standard Time). Когда этот файл активен, `SAT` не будет распознаваться как сокращение слова «суббота».

Внимание

Если вы модифицируете файлы в `.../share/timezonesets/`, вы можете сами выполнить резервное копирование, так как обычный дамп базы данных не содержит этот каталог.

B.5. Указание часовых поясов в стиле POSIX

Postgres Pro может воспринимать указание часового пояса, записанное по правилам стандарта POSIX, в соответствии с которыми, в частности, задаётся переменная окружения `TZ`. Указания часовых поясов в стиле POSIX не удовлетворяют всем условиям, продиктованным сложной историей часовых поясов в реальном мире, но иногда использование этих указаний бывает оправданным.

Указание часового пояса в стиле POSIX выглядит так:

```
STD смещение [ DST [ смещение_летнего_времени ] [ , правило ] ]
```

(Для наглядности между полями добавлены пробелы, но в реальном указании их не должно быть.) В нём содержатся следующие поля:

- *`STD`* — аббревиатура, используемая для стандартного часового пояса.
- *`смещение`* — стандартное смещение пояса от UTC.
- *`DST`* — аббревиатура часового пояса при переходе на летнее время. Если это поле и следующие за ним опущены, для часового пояса будет использоваться фиксированное смещение от UTC без учёта перехода на летнее время.
- *`смещение_летнего_времени`* — смещение часового пояса от UTC при переходе на летнее время. Это поле обычно опускается, так как по умолчанию это смещение равно стандартному *`смещению`* минус один час, что практически всегда имеет место на практике.

- *`правило`* — определяет, как должен учитываться переход на летнее время, в соответствии с представленным ниже описанием.

В этой записи аббревиатура часового пояса может задаваться набором букв, например `EST`, или произвольной строкой, заключённой в угловые скобки, например `<UTC-05>`. Заметьте, что показанные здесь аббревиатуры используются только при выводе и только в некоторых выходных форматах. Распознаваемые во входных значениях даты/времени аббревиатуры часовых поясов описаны в Разделе B.4.

В полях смещений задаётся сдвиг от UTC в часах и, возможно, минутах и секундах. Они имеют формат *`чч`*[`:`*`мм`*[`:`*`сс`*]] и могут содержать спереди знак (`+` или `-`). Положительный знак имеют часовые пояса *к западу* от Гринвича. (Заметьте, что это противоположно соглашению ISO-8601, используемому в других областях Postgres Pro.) Компонент *`чч`* может содержать одну или две цифры, а *`мм`* и *`сс`* (если они задаются) должны состоять ровно из двух цифр.

Поле, задающее *`правило`* перехода на летнее время, имеет следующий формат:

`дата_перехода` `[ /` *`время_перехода`* `] ,` *`дата_возврата`* `[ /` *`время_возврата`* `]`

(Как и ранее, реальное указание не должно содержать пробелы.) Поля *`дата_перехода`* и *`время_перехода`* определяют момент перехода на летнее время, а поля *`дата_возврата`* и *`время_возврата`* определяют дату возврата к поясному времени. (В некоторых регионах, в частности, в Южном полушарии, вторая дата в календаре может предшествовать первой.) Даты могут задаваться в одном из следующих форматов:

`n`

Простое целое, обозначающее день года, с нумерацией от нуля до 364, или 365 в високосном году.

`J`*`n`*

В этой записи *`n`* обозначает номер дня от 1 до 365, а 29 февраля пропускается, даже когда фактически присутствует. (Таким образом, смены часового пояса, происходящие 29 февраля, в этой записи выразить нельзя. Однако после февраля дни имеют одинаковые номера и в обычные, и в високосные годы, так что эта запись обычно полезнее тем, что позволяет выразить одним числом фиксированную дату перевода часов.)

`M`*`m`*`.`*`n`*`.`*`d`*

Эта форма задаёт дату перехода, которая всегда назначается в определённый месяц и день недели. Здесь *`m`* обозначает номер месяца от 1 до 12, а *`n`* — номер повторения в этом месяце дня недели, заданного полем *`d`*. В качестве *`n`* может задаваться число от 1 до 4 либо число 5, выбирающее последнюю дату с заданным днём недели в этом месяце (она может быть фактически четвёртой или пятой). В качестве *`d`* задаётся число от 0 до 6, обозначающее день недели (0 — воскресенье). Например, запись `M3.2.0` расшифровывается как «второе воскресенье марта».

Примечание

Для описания многих существующих правил перехода на летнее время обычно достаточно формата `M`. Но заметьте, что ни одна из этих вариаций не позволяет описать изменения правил перевода часов. Поэтому на практике для правильной интерпретации значений времени, относящихся к прошлому, необходимы исторические данные, хранящиеся в привязке к именованным часовым поясам (в базе данных IANA с информацией о часовых поясах).

Поля времени в описании правила перехода имеют тот же формат, что и поля смещений, описанные ранее, за исключением того, что в них не может быть знака. Они определяют текущее местное время, в которое производится перевод часов. По умолчанию временем перевода часов считается `02:00:00`.

Если задаётся аббревиатура для летнего времени, но *`правило`* перехода не задано, в качестве правила по умолчанию используется `M3.2.0,M11.1.0`, что соответствует существующему в США в 2020 г. порядку. То есть часы переводятся на летнее время во второе воскресенье марта, а на зимнее — в первое воскресенье ноября, в два часа по действующему времени. Заметьте, что это правило даёт правильные даты для США только с 2007 г.

Например, запись `CET-1CEST,M3.5.0,M10.5.0/3` отражает действующую в 2020 г. практику перевода часов в Париже. Это указание говорит, что стандартное поясное время имеет аббревиатуру `CET` и оно смещено на час вперёд (к востоку) от UTC; летнее время имеет аббревиатуру `CEST` и смещено вперед от UTC на два часа (это определяется неявно). Часы переводятся на летнее время в последнее воскресенье марта в 2:00 CET, а на зимнее — в последнее воскресенье октября в 3:00 CEST.

Названия четырёх часовых поясов `EST5EDT`, `CST6CDT`, `MST7MDT` и `PST8PDT` выглядят как определяющие часовые пояса в стиле POSIX, но в реальности по историческим причинам они воспринимаются как имена часовых поясов, наравне с другими, так как в базе данных IANA есть файлы с такими именами. Практическое следствие этого состоит в том, что для данных имён может быть получена историческая информация о действовавших правилах перехода на летнее время в США, которую не может дать обычное указание в стиле POSIX.

Имейте в виду, что ошибиться в указании часового пояса в стиле POSIX очень легко, так как адекватность аббревиатуры никак не проверяется. Например, команда `SET TIMEZONE TO FOOBAR0` выполнится без ошибки, и система в итоге будет использовать весьма своеобразную аббревиатуру для UTC.

B.6. История единиц измерения времени

Стандарт SQL устанавливает, что «В определении „литерала типа дата-время“, „значения типа дата-время“ ограничены естественными правилами, касающимися дат и времени согласно григорианскому календарю». Следуя стандарту SQL, Postgres Pro подсчитывает даты исключительно в григорианском календаре, включая годы, когда этот календарь ещё не использовался. Это правило известно как *пролептический григорианский календарь*.

Юлианский календарь был введён Юлием Цезарем в 45 г. до н. э. Он широко использовался западной цивилизацией до 1582 года, когда страны начали переходить на григорианский календарь. В юлианском календаре тропический год длится приблизительно 365 1/4 дня = 365,25 дня. Каждые 128 лет накапливается примерно 1 день.

Накапливающаяся погрешность побудила папу Григория XIII реформировать календарь в соответствии с постановлениями Тридентского собора. В григорианском календаре тропический год длится приблизительно 365 + 97 / 400 дней = 365,2425 дней. Таком образом, погрешность в один день тропического года накапливается примерно за 3300 лет.

Приблизительное число 365+97/400 получается из-за того, что из каждых 400 лет 97 високосные. При этом действуют следующие правила:

Каждый год кратный 4 является високосным.
Однако каждый год кратный 100 не является високосным.
Тем не менее каждый год кратный 400 всё же является високосным.

Таким образом, 1700, 1800, 1900, 2100 и 2200 не являются високосными годами. Но 1600, 2000 и 2400 — високосные. А в юлианском календаре високосными считаются все годы, кратные 4.

Папская булла, изданная в феврале 1582 года, предписывала сделать октябрь 1582 года на 10 дней короче, чтобы 15 октября следовало сразу за 4 октября. Это правило соблюдалось в Италии, Польше, Португалии и Испании. Вскоре к ним присоединились и прочие католические страны, но протестантские страны вводили эти изменения неохотно, а страны греческой православной церкви не переходили на новый календарь до начала 20 века. В 1752 г. реформа была проведена в Великобритании и её доминионах (включая территорию сегодняшних Соединённых Штатов Америки).

Таким образом, за 2 сентября 1752 года следовало 14 сентября 1752 года. Поэтому в системах Unix программа `cal` выводит следующее:

```
$ cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Но этот календарь действует только в Великобритании и доминионах. В других местах он является недействительным. Чтобы избежать сложностей и возможной путаницы при отслеживании календарей, которыми фактически пользовались в различных местах в разное время, Postgres Pro применяет правила григорианского календаря ко всем датам, даже если это нарушает историческую достоверность.

Разные календари были составлены в различных частях земного шара, многие из них до григорианской системы. Например, появление китайского календаря относится к 14 веку до н. э. Легенда гласит, что император Хуан-ди изобрёл этот календарь в 2637 г. до н. э. В Китайской Народной Республике григорианский календарь используется для официальных и коммерческих нужд. Китайский календарь используется для определения дат традиционных праздников.

B.7. Юлианские даты

Система *юлианских дат* определяет способ исчисления времени. Она не имеет отношения к юлианскому календарю, несмотря на схожее название. Система юлианских дат была предложена французским ученым Жозефом Жюстом Скалигером (1540–1609) и получила своё название, вероятно, в честь его отца, итальянского ученого Юлия Цезаря Скалигера (1484–1558).

В системе юлианских дат каждый день имеет порядковый номер, начиная с 0 (иногда его называют *номером юлианского дня*). Первый день имеет номер 0 и соответствует 1 января 4713 г. до н. э. по юлианскому календарю или 24 ноября 4714 г. до н. э. по григорианскому календарю. Юлианская дата чаще всего используется в астрономических расчётах для записи ночных наблюдений, и поэтому день длится с полудня до полудня UTC, а не с полуночи до полуночи: Первый юлианский день (JD 0) обозначает 24 часа от полудня UTC 24 ноября 4714 г. до н. э. до полудня UTC 25 ноября 4714 г. до н. э.

Хотя Postgres Pro поддерживает юлианскую дату для записи входных и выходных дат (а также использует юлианские даты для некоторых внутренних вычислений в формате дата-время), полдень не считается началом суток. Postgres Pro считает юлианский день длящимся от полуночи до полуночи, как и обычный день.

Однако если вам нужно получить астрономическое определение, это можно сделать, произведя вычисления в часовом поясе `UTC+12`. Например,

```
=> SELECT extract(julian from '2021-06-23 7:00:00-04'::timestamptz at time zone 'UTC
+12');
           extract
------------------------------
 2459388.95833333333333333333
(1 row)
=> SELECT extract(julian from '2021-06-23 8:00:00-04'::timestamptz at time zone 'UTC
+12');
               extract
--------------------------------------
 2459389.0000000000000000000000000000
(1 row)
=> SELECT extract(julian from date '2021-06-23');
 extract
---------
```

```
 2459389
(1 row)
```

Приложение C. Ключевые слова SQL

В Таблице C.1 перечислены все слова, которые являются ключевыми в стандарте SQL и в Postgres Pro Standard 17.6.1. Общее описание ключевых слов можно найти в Подразделе 4.1.1. (Для экономии места в таблицу включены только две последние версии стандарта SQL и SQL-92 для исторического сравнения. Отличия между ними и другими промежуточными версиями стандарта невелики.)

В SQL есть различие между *зарезервированными* и *незарезервированными* ключевыми словами. Согласно стандарту, действительно ключевыми словами являются только зарезервированные слова; они не могут быть идентификаторами. Незарезервированные ключевые слова имеют особое значение только в определённых контекстах и могут быть идентификаторами в других. Большинство незарезервированных ключевых слов на самом деле представляют имена встроенных таблиц и функций, определённых в SQL. Концепция незарезервированных ключевых слов собственно введена только для того, чтобы показать, что эти слова имеют некоторое предопределённое значение в отдельных контекстах.

В Postgres Pro анализатор SQL сталкивается с дополнительными сложностями. Ему приходится иметь дело с несколькими различными классами элементов языка, начиная с тех, что никогда не могут использоваться как идентификаторы, и заканчивая теми, которые не имеют никакого специального значения для анализатора, а считаются обычными идентификаторами. (Последнее обычно относится к функциям, описанным в SQL.) Даже зарезервированные ключевые слова не полностью зарезервированы в Postgres Pro, а могут использоваться в качестве меток столбцов (например, можно написать `SELECT 55 AS CHECK`, хотя `CHECK` и является зарезервированным ключевым словом).

В Таблице C.1, в столбце Postgres Pro мы даём пометку «не зарезервировано» тем ключевым словам, которые явно известны анализатору запросов, но их можно использовать в качестве имени столбца или таблицы. Некоторые ключевые слова, которые недопустимы в качестве имени функции или типа данных, но в остальном не отличаются от незарезервированных слов, помечены соответственно. (Большинство из этих слов представляют встроенные функции или типы данных со специальным синтаксисом. Функции или типы с таким именем существуют, но пользователь не может их переопределить.) Метка «зарезервировано» даётся тем словам, которые не могут быть именами столбцов или таблиц. Некоторые зарезервированные ключевые слова могут быть именами функций или типов данных; это также отмечается в таблице. Если такой пометки нет, зарезервированное слово допускается только в качестве метки столбца. Пустое содержимое ячейки означает, что Postgres Pro воспринимает это слово как обычный идентификатор.

Кроме того, хотя большинство ключевых слов можно использовать как «открытые» метки столбцов, не добавляя перед ними `AS` (как описано в Подразделе 7.3.2), есть ряд исключений — перед такими ключевыми словами нужно добавлять `AS` во избежание двусмысленности. В этой таблице они имеют пометку «requires `AS`».

Вообще, если вы сталкиваетесь с разнообразными ошибками разбора команд, использующих в качестве идентификаторов какие-либо из перечисленных ключевых слов, попробуйте для решения проблемы заключить идентификатор в кавычки.

Изучая Таблицу C.1, важно понимать, что отсутствие какого-либо ключевого слова в списке зарезервированных в Postgres Pro не означает, что функциональность, связанная с этим словом, не реализована. И наоборот, присутствие ключевого слова не обязательно говорит о наличии соответствующей функциональности.

Таблица C.1. SQL Key Words

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`A`		non-reserved	non-reserved	
`ABORT`	non-reserved			
`ABS`		reserved	reserved	

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`ABSENT`	non-reserved	reserved	reserved	
`ABSOLUTE`	non-reserved	non-reserved	non-reserved	reserved
`ACCESS`	non-reserved			
`ACCORDING`		non-reserved	non-reserved	
`ACCOUNT`	non-reserved			
`ACOS`		reserved	reserved	
`ACTION`	non-reserved	non-reserved	non-reserved	reserved
`ADA`		non-reserved	non-reserved	non-reserved
`ADD`	non-reserved	non-reserved	non-reserved	reserved
`ADMIN`	non-reserved	non-reserved	non-reserved	
`AFTER`	non-reserved	non-reserved	non-reserved	
`AGGREGATE`	non-reserved			
`ALL`	reserved	reserved	reserved	reserved
`ALLOCATE`		reserved	reserved	reserved
`ALSO`	non-reserved			
`ALTER`	non-reserved	reserved	reserved	reserved
`ALWAYS`	non-reserved	non-reserved	non-reserved	
`ANALYSE`	reserved			
`ANALYZE`	reserved			
`AND`	reserved	reserved	reserved	reserved
`ANY`	reserved	reserved	reserved	reserved
`ANY_VALUE`		reserved		
`APPLICATION`	non-reserved			
`ARE`		reserved	reserved	reserved
`ARRAY`	reserved, requires `AS`	reserved	reserved	
`ARRAY_AGG`		reserved	reserved	
`ARRAY_MAX_CARDINALITY`		reserved	reserved	
`AS`	reserved, requires `AS`	reserved	reserved	reserved
`ASC`	reserved	non-reserved	non-reserved	reserved
`ASENSITIVE`	non-reserved	reserved	reserved	
`ASIN`		reserved	reserved	
`ASSERTION`	non-reserved	non-reserved	non-reserved	reserved
`ASSIGNMENT`	non-reserved	non-reserved	non-reserved	
`ASYMMETRIC`	reserved	reserved	reserved	
`AT`	non-reserved	reserved	reserved	reserved
`ATAN`		reserved	reserved	
`ATOMIC`	non-reserved	reserved	reserved	
`ATTACH`	non-reserved			

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`ATTRIBUTE`	non-reserved	non-reserved	non-reserved	
`ATTRIBUTES`		non-reserved	non-reserved	
`AUTHORIZATION`	reserved (can be function or type)	reserved	reserved	reserved
`AVG`		reserved	reserved	reserved
`BACKWARD`	non-reserved			
`BASE64`		non-reserved	non-reserved	
`BEFORE`	non-reserved	non-reserved	non-reserved	
`BEGIN`	non-reserved	reserved	reserved	reserved
`BEGIN_FRAME`		reserved	reserved	
`BEGIN_PARTITION`		reserved	reserved	
`BERNOULLI`		non-reserved	non-reserved	
`BETWEEN`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`BIGINT`	non-reserved (cannot be function or type)	reserved	reserved	
`BINARY`	reserved (can be function or type)	reserved	reserved	
`BIT`	non-reserved (cannot be function or type)			reserved
`BIT_LENGTH`				reserved
`BLOB`		reserved	reserved	
`BLOCKED`		non-reserved	non-reserved	
`BOM`		non-reserved	non-reserved	
`BOOLEAN`	non-reserved (cannot be function or type)	reserved	reserved	
`BOTH`	reserved	reserved	reserved	reserved
`BREADTH`	non-reserved	non-reserved	non-reserved	
`BTRIM`		reserved		
`BY`	non-reserved	reserved	reserved	reserved
`C`		non-reserved	non-reserved	non-reserved
`CACHE`	non-reserved			
`CALL`	non-reserved	reserved	reserved	
`CALLED`	non-reserved	reserved	reserved	
`CARDINALITY`		reserved	reserved	
`CASCADE`	non-reserved	non-reserved	non-reserved	reserved
`CASCADED`	non-reserved	reserved	reserved	reserved
`CASE`	reserved	reserved	reserved	reserved
`CAST`	reserved	reserved	reserved	reserved

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`CATALOG`	non-reserved	non-reserved	non-reserved	reserved
`CATALOG_NAME`		non-reserved	non-reserved	non-reserved
`CEIL`		reserved	reserved	
`CEILING`		reserved	reserved	
`CHAIN`	non-reserved	non-reserved	non-reserved	
`CHAINING`		non-reserved	non-reserved	
`CHAR`	non-reserved (cannot be function or type), requires `AS`	reserved	reserved	reserved
`CHARACTER`	non-reserved (cannot be function or type), requires `AS`	reserved	reserved	reserved
`CHARACTERISTICS`	non-reserved	non-reserved	non-reserved	
`CHARACTERS`		non-reserved	non-reserved	
`CHARACTER_LENGTH`		reserved	reserved	reserved
`CHARACTER_SET_CATALOG`		non-reserved	non-reserved	non-reserved
`CHARACTER_SET_NAME`		non-reserved	non-reserved	non-reserved
`CHARACTER_SET_SCHEMA`		non-reserved	non-reserved	non-reserved
`CHAR_LENGTH`		reserved	reserved	reserved
`CHECK`	reserved	reserved	reserved	reserved
`CHECKPOINT`	non-reserved			
`CLASS`	non-reserved			
`CLASSIFIER`		reserved	reserved	
`CLASS_ORIGIN`		non-reserved	non-reserved	non-reserved
`CLOB`		reserved	reserved	
`CLOSE`	non-reserved	reserved	reserved	reserved
`CLUSTER`	non-reserved			
`COALESCE`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`COBOL`		non-reserved	non-reserved	non-reserved
`COLLATE`	reserved	reserved	reserved	reserved
`COLLATION`	reserved (can be function or type)	non-reserved	non-reserved	reserved
`COLLATION_CATALOG`		non-reserved	non-reserved	non-reserved
`COLLATION_NAME`		non-reserved	non-reserved	non-reserved
`COLLATION_SCHEMA`		non-reserved	non-reserved	non-reserved
`COLLECT`		reserved	reserved	
`COLUMN`	reserved	reserved	reserved	reserved
`COLUMNS`	non-reserved	non-reserved	non-reserved	
`COLUMN_NAME`		non-reserved	non-reserved	non-reserved
`COMMAND_FUNCTION`		non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`COMMAND_FUNCTION_CODE`		non-reserved	non-reserved	
`COMMENT`	non-reserved			
`COMMENTS`	non-reserved			
`COMMIT`	non-reserved	reserved	reserved	reserved
`COMMITTED`	non-reserved	non-reserved	non-reserved	non-reserved
`COMPRESSION`	non-reserved			
`CONCURRENTLY`	reserved (can be function or type)			
`CONDITION`		reserved	reserved	
`CONDITIONAL`	non-reserved	non-reserved	non-reserved	
`CONDITION_NUMBER`		non-reserved	non-reserved	non-reserved
`CONFIGURATION`	non-reserved			
`CONFLICT`	non-reserved			
`CONNECT`		reserved	reserved	reserved
`CONNECTION`	non-reserved	non-reserved	non-reserved	reserved
`CONNECTION_NAME`		non-reserved	non-reserved	non-reserved
`CONSTRAINT`	reserved	reserved	reserved	reserved
`CONSTRAINTS`	non-reserved	non-reserved	non-reserved	reserved
`CONSTRAINT_CATALOG`		non-reserved	non-reserved	non-reserved
`CONSTRAINT_NAME`		non-reserved	non-reserved	non-reserved
`CONSTRAINT_SCHEMA`		non-reserved	non-reserved	non-reserved
`CONSTRUCTOR`		non-reserved	non-reserved	
`CONTAINS`		reserved	reserved	
`CONTENT`	non-reserved	non-reserved	non-reserved	
`CONTINUE`	non-reserved	non-reserved	non-reserved	reserved
`CONTROL`		non-reserved	non-reserved	
`CONVERSION`	non-reserved			
`CONVERT`		reserved	reserved	reserved
`COPARTITION`		non-reserved		
`COPY`	non-reserved	reserved	reserved	
`CORR`		reserved	reserved	
`CORRESPONDING`		reserved	reserved	reserved
`COS`		reserved	reserved	
`COSH`		reserved	reserved	
`COST`	non-reserved			
`COUNT`		reserved	reserved	reserved
`COVAR_POP`		reserved	reserved	
`COVAR_SAMP`		reserved	reserved	
`CREATE`	reserved, requires `AS`	reserved	reserved	reserved

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`CROSS`	reserved (can be function or type)	reserved	reserved	reserved
`CSV`	non-reserved			
`CUBE`	non-reserved	reserved	reserved	
`CUME_DIST`		reserved	reserved	
`CURRENT`	non-reserved	reserved	reserved	reserved
`CURRENT_CATALOG`	reserved	reserved	reserved	
`CURRENT_DATE`	reserved	reserved	reserved	reserved
`CURRENT_DEFAULT_TRANSFORM_GROUP`		reserved	reserved	
`CURRENT_PATH`		reserved	reserved	
`CURRENT_ROLE`	reserved	reserved	reserved	
`CURRENT_ROW`		reserved	reserved	
`CURRENT_SCHEMA`	reserved (can be function or type)	reserved	reserved	
`CURRENT_TIME`	reserved	reserved	reserved	reserved
`CURRENT_TIMESTAMP`	reserved	reserved	reserved	reserved
`CURRENT_TRANSFORM_GROUP_FOR_TYPE`		reserved	reserved	
`CURRENT_USER`	reserved	reserved	reserved	reserved
`CURSOR`	non-reserved	reserved	reserved	reserved
`CURSOR_NAME`		non-reserved	non-reserved	non-reserved
`CYCLE`	non-reserved	reserved	reserved	
`DATA`	non-reserved	non-reserved	non-reserved	non-reserved
`DATABASE`	non-reserved			
`DATALINK`		reserved	reserved	
`DATE`		reserved	reserved	reserved
`DATETIME_INTERVAL_CODE`		non-reserved	non-reserved	non-reserved
`DATETIME_INTERVAL_PRECISION`		non-reserved	non-reserved	non-reserved
`DAY`	non-reserved, requires `AS`	reserved	reserved	reserved
`DB`		non-reserved	non-reserved	
`DEALLOCATE`	non-reserved	reserved	reserved	reserved
`DEC`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`DECFLOAT`		reserved	reserved	
`DECIMAL`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`DECLARE`	non-reserved	reserved	reserved	reserved
`DEFAULT`	reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`DEFAULTS`	non-reserved	non-reserved	non-reserved	
`DEFERRABLE`	reserved	non-reserved	non-reserved	reserved
`DEFERRED`	non-reserved	non-reserved	non-reserved	reserved
`DEFINE`		reserved	reserved	
`DEFINED`		non-reserved	non-reserved	
`DEFINER`	non-reserved	non-reserved	non-reserved	
`DEGREE`		non-reserved	non-reserved	
`DELETE`	non-reserved	reserved	reserved	reserved
`DELIMITER`	non-reserved			
`DELIMITERS`	non-reserved			
`DENSE_RANK`		reserved	reserved	
`DEPENDS`	non-reserved			
`DEPTH`	non-reserved	non-reserved	non-reserved	
`DEREF`		reserved	reserved	
`DERIVED`		non-reserved	non-reserved	
`DESC`	reserved	non-reserved	non-reserved	reserved
`DESCRIBE`		reserved	reserved	reserved
`DESCRIPTOR`		non-reserved	non-reserved	reserved
`DETACH`	non-reserved			
`DETERMINISTIC`		reserved	reserved	
`DIAGNOSTICS`		non-reserved	non-reserved	reserved
`DICTIONARY`	non-reserved			
`DISABLE`	non-reserved			
`DISCARD`	non-reserved			
`DISCONNECT`		reserved	reserved	reserved
`DISPATCH`		non-reserved	non-reserved	
`DISTINCT`	reserved	reserved	reserved	reserved
`DLNEWCOPY`		reserved	reserved	
`DLPREVIOUSCOPY`		reserved	reserved	
`DLURLCOMPLETE`		reserved	reserved	
`DLURLCOMPLETEONLY`		reserved	reserved	
`DLURLCOMPLETEWRITE`		reserved	reserved	
`DLURLPATH`		reserved	reserved	
`DLURLPATHONLY`		reserved	reserved	
`DLURLPATHWRITE`		reserved	reserved	
`DLURLSCHEME`		reserved	reserved	
`DLURLSERVER`		reserved	reserved	
`DLVALUE`		reserved	reserved	
`DO`	reserved			
`DOCUMENT`	non-reserved	non-reserved	non-reserved	

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`DOMAIN`	non-reserved	non-reserved	non-reserved	reserved
`DOUBLE`	non-reserved	reserved	reserved	reserved
`DROP`	non-reserved	reserved	reserved	reserved
`DYNAMIC`		reserved	reserved	
`DYNAMIC_FUNCTION`		non-reserved	non-reserved	non-reserved
`DYNAMIC_FUNCTION_CODE`		non-reserved	non-reserved	
`EACH`	non-reserved	reserved	reserved	
`ELEMENT`		reserved	reserved	
`ELSE`	reserved	reserved	reserved	reserved
`EMPTY`	non-reserved	reserved	reserved	
`ENABLE`	non-reserved			
`ENCODING`	non-reserved	non-reserved	non-reserved	
`ENCRYPTED`	non-reserved			
`END`	reserved	reserved	reserved	reserved
`END-EXEC`		reserved	reserved	reserved
`END_FRAME`		reserved	reserved	
`END_PARTITION`		reserved	reserved	
`ENFORCED`		non-reserved	non-reserved	
`ENUM`	non-reserved			
`EQUALS`		reserved	reserved	
`ERROR`	non-reserved	non-reserved	non-reserved	
`ESCAPE`	non-reserved	reserved	reserved	reserved
`EVENT`	non-reserved			
`EVERY`		reserved	reserved	
`EXCEPT`	reserved, requires `AS`	reserved	reserved	reserved
`EXCEPTION`				reserved
`EXCLUDE`	non-reserved	non-reserved	non-reserved	
`EXCLUDING`	non-reserved	non-reserved	non-reserved	
`EXCLUSIVE`	non-reserved			
`EXEC`		reserved	reserved	reserved
`EXECUTE`	non-reserved	reserved	reserved	reserved
`EXISTS`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`EXP`		reserved	reserved	
`EXPLAIN`	non-reserved			
`EXPRESSION`	non-reserved	non-reserved	non-reserved	
`EXTENSION`	non-reserved			
`EXTERNAL`	non-reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`EXTRACT`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`FAILED_AUTH_KEEP_TIME`	non-reserved			
`FAILED_LOGIN_ATTEMPTS`	non-reserved			
`FALSE`	reserved	reserved	reserved	reserved
`FAMILY`	non-reserved			
`FETCH`	reserved, requires `AS`	reserved	reserved	reserved
`FILE`		non-reserved	non-reserved	
`FILTER`	non-reserved, requires `AS`	reserved	reserved	
`FINAL`		non-reserved	non-reserved	
`FINALIZE`	non-reserved			
`FINISH`		non-reserved	non-reserved	
`FIRST`	non-reserved	non-reserved	non-reserved	reserved
`FIRST_VALUE`		reserved	reserved	
`FLAG`		non-reserved	non-reserved	
`FLOAT`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`FLOOR`		reserved	reserved	
`FOLLOWING`	non-reserved	non-reserved	non-reserved	
`FOR`	reserved, requires `AS`	reserved	reserved	reserved
`FORCE`	non-reserved			
`FOREIGN`	reserved	reserved	reserved	reserved
`FORMAT`	non-reserved	non-reserved	non-reserved	
`FORTRAN`		non-reserved	non-reserved	non-reserved
`FORWARD`	non-reserved			
`FOUND`		non-reserved	non-reserved	reserved
`FRAME_ROW`		reserved	reserved	
`FREE`		reserved	reserved	
`FREEZE`	reserved (can be function or type)			
`FROM`	reserved, requires `AS`	reserved	reserved	reserved
`FS`		non-reserved	non-reserved	
`FULFILL`		non-reserved	non-reserved	
`FULL`	reserved (can be function or type)	reserved	reserved	reserved
`FUNCTION`	non-reserved	reserved	reserved	
`FUNCTIONS`	non-reserved			

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`FUSION`		reserved	reserved	
`G`		non-reserved	non-reserved	
`GENERAL`		non-reserved	non-reserved	
`GENERATED`	non-reserved	non-reserved	non-reserved	
`GET`		reserved	reserved	reserved
`GLOBAL`	non-reserved	reserved	reserved	reserved
`GO`		non-reserved	non-reserved	reserved
`GOTO`		non-reserved	non-reserved	reserved
`GRANT`	reserved, requires `AS`	reserved	reserved	reserved
`GRANTED`	non-reserved	non-reserved	non-reserved	
`GREATEST`	non-reserved (cannot be function or type)	reserved		
`GROUP`	reserved, requires `AS`	reserved	reserved	reserved
`GROUPING`	non-reserved (cannot be function or type)	reserved	reserved	
`GROUPS`	non-reserved	reserved	reserved	
`HANDLER`	non-reserved			
`HAVING`	reserved, requires `AS`	reserved	reserved	reserved
`HEADER`	non-reserved			
`HEX`		non-reserved	non-reserved	
`HIERARCHY`		non-reserved	non-reserved	
`HOLD`	non-reserved	reserved	reserved	
`HOUR`	non-reserved, requires `AS`	reserved	reserved	reserved
`ID`		non-reserved	non-reserved	
`IDENTITY`	non-reserved	reserved	reserved	reserved
`IF`	non-reserved			
`IGNORE`		non-reserved	non-reserved	
`ILIKE`	reserved (can be function or type)			
`IMMEDIATE`	non-reserved	non-reserved	non-reserved	reserved
`IMMEDIATELY`		non-reserved	non-reserved	
`IMMUTABLE`	non-reserved			
`IMPLEMENTATION`		non-reserved	non-reserved	
`IMPLICIT`	non-reserved			
`IMPORT`	non-reserved	reserved	reserved	
`IN`	reserved	reserved	reserved	reserved

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`INCLUDE`	non-reserved			
`INCLUDING`	non-reserved	non-reserved	non-reserved	
`INCREMENT`	non-reserved	non-reserved	non-reserved	
`INDENT`	non-reserved	non-reserved	non-reserved	
`INDEX`	non-reserved			
`INDEXES`	non-reserved			
`INDICATOR`		reserved	reserved	reserved
`INHERIT`	non-reserved			
`INHERITS`	non-reserved			
`INITIAL`		reserved	reserved	
`INITIALLY`	reserved	non-reserved	non-reserved	reserved
`INLINE`	non-reserved			
`INNER`	reserved (can be function or type)	reserved	reserved	reserved
`INOUT`	non-reserved (cannot be function or type)	reserved	reserved	
`INPUT`	non-reserved	non-reserved	non-reserved	reserved
`INSENSITIVE`	non-reserved	reserved	reserved	reserved
`INSERT`	non-reserved	reserved	reserved	reserved
`INSTANCE`		non-reserved	non-reserved	
`INSTANTIABLE`		non-reserved	non-reserved	
`INSTEAD`	non-reserved	non-reserved	non-reserved	
`INT`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`INTEGER`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`INTEGRITY`		non-reserved	non-reserved	
`INTERSECT`	reserved, requires `AS`	reserved	reserved	reserved
`INTERSECTION`		reserved	reserved	
`INTERVAL`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`INTO`	reserved, requires `AS`	reserved	reserved	reserved
`INVOKER`	non-reserved	non-reserved	non-reserved	
`IS`	reserved (can be function or type)	reserved	reserved	reserved
`ISNULL`	reserved (can be function or type), requires `AS`			

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`ISOLATION`	non-reserved	non-reserved	non-reserved	reserved
`JOIN`	reserved (can be function or type)	reserved	reserved	reserved
`JSON`	non-reserved (cannot be function or type)	reserved		
`JSON_ARRAY`	non-reserved (cannot be function or type)	reserved	reserved	
`JSON_ARRAYAGG`	non-reserved (cannot be function or type)	reserved	reserved	
`JSON_EXISTS`	non-reserved (cannot be function or type)	reserved	reserved	
`JSON_OBJECT`	non-reserved (cannot be function or type)	reserved	reserved	
`JSON_OBJECTAGG`	non-reserved (cannot be function or type)	reserved	reserved	
`JSON_QUERY`	non-reserved (cannot be function or type)	reserved	reserved	
`JSON_SCALAR`	non-reserved (cannot be function or type)	reserved		
`JSON_SERIALIZE`	non-reserved (cannot be function or type)	reserved		
`JSON_TABLE`	non-reserved (cannot be function or type)	reserved	reserved	
`JSON_TABLE_PRIMITIVE`		reserved	reserved	
`JSON_VALUE`	non-reserved (cannot be function or type)	reserved	reserved	
`K`		non-reserved	non-reserved	
`KEEP`	non-reserved	non-reserved	non-reserved	
`KEY`	non-reserved	non-reserved	non-reserved	reserved
`KEYS`	non-reserved	non-reserved	non-reserved	
`KEY_MEMBER`		non-reserved	non-reserved	
`KEY_TYPE`		non-reserved	non-reserved	
`LABEL`	non-reserved			
`LAG`		reserved	reserved	
`LANGUAGE`	non-reserved	reserved	reserved	reserved
`LARGE`	non-reserved	reserved	reserved	

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`LAST`	non-reserved	non-reserved	non-reserved	reserved
`LAST_VALUE`		reserved	reserved	
`LATERAL`	reserved	reserved	reserved	
`LEAD`		reserved	reserved	
`LEADING`	reserved	reserved	reserved	reserved
`LEAKPROOF`	non-reserved			
`LEAST`	non-reserved (cannot be function or type)	reserved		
`LEFT`	reserved (can be function or type)	reserved	reserved	reserved
`LENGTH`		non-reserved	non-reserved	non-reserved
`LEVEL`	non-reserved	non-reserved	non-reserved	reserved
`LIBRARY`		non-reserved	non-reserved	
`LIKE`	reserved (can be function or type)	reserved	reserved	reserved
`LIKE_REGEX`		reserved	reserved	
`LIMIT`	reserved, requires `AS`	non-reserved	non-reserved	
`LINK`		non-reserved	non-reserved	
`LISTAGG`		reserved	reserved	
`LISTEN`	non-reserved			
`LN`		reserved	reserved	
`LOAD`	non-reserved			
`LOCAL`	non-reserved	reserved	reserved	reserved
`LOCALTIME`	reserved	reserved	reserved	
`LOCALTIMESTAMP`	reserved	reserved	reserved	
`LOCATION`	non-reserved	non-reserved	non-reserved	
`LOCATOR`		non-reserved	non-reserved	
`LOCK`	non-reserved			
`LOCKED`	non-reserved			
`LOG`		reserved	reserved	
`LOG10`		reserved	reserved	
`LOGGED`	non-reserved			
`LOWER`		reserved	reserved	reserved
`LPAD`		reserved		
`LTRIM`		reserved		
`M`		non-reserved	non-reserved	
`MAP`		non-reserved	non-reserved	
`MAPPING`	non-reserved	non-reserved	non-reserved	
`MATCH`	non-reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`MATCHED`	non-reserved	non-reserved	non-reserved	
`MATCHES`		reserved	reserved	
`MATCH_NUMBER`		reserved	reserved	
`MATCH_RECOGNIZE`		reserved	reserved	
`MATERIALIZED`	non-reserved			
`MAX`		reserved	reserved	reserved
`MAXVALUE`	non-reserved	non-reserved	non-reserved	
`MEASURES`		non-reserved	non-reserved	
`MEMBER`		reserved	reserved	
`MERGE`	non-reserved	reserved	reserved	
`MERGE_ACTION`	non-reserved (cannot be function or type)			
`MESSAGE_LENGTH`		non-reserved	non-reserved	non-reserved
`MESSAGE_OCTET_LENGTH`		non-reserved	non-reserved	non-reserved
`MESSAGE_TEXT`		non-reserved	non-reserved	non-reserved
`METHOD`	non-reserved	reserved	reserved	
`MIN`		reserved	reserved	reserved
`MINUTE`	non-reserved, requires `AS`	reserved	reserved	reserved
`MINVALUE`	non-reserved	non-reserved	non-reserved	
`MOD`		reserved	reserved	
`MODE`	non-reserved			
`MODIFIES`		reserved	reserved	
`MODULE`		reserved	reserved	reserved
`MONTH`	non-reserved, requires `AS`	reserved	reserved	reserved
`MORE`		non-reserved	non-reserved	non-reserved
`MOVE`	non-reserved			
`MULTISET`		reserved	reserved	
`MUMPS`		non-reserved	non-reserved	non-reserved
`NAME`	non-reserved	non-reserved	non-reserved	non-reserved
`NAMES`	non-reserved	non-reserved	non-reserved	reserved
`NAMESPACE`		non-reserved	non-reserved	
`NATIONAL`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`NATURAL`	reserved (can be function or type)	reserved	reserved	reserved
`NCHAR`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`NCLOB`		reserved	reserved	

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`NESTED`	non-reserved	non-reserved	non-reserved	
`NESTING`		non-reserved	non-reserved	
`NEW`	non-reserved	reserved	reserved	
`NEXT`	non-reserved	non-reserved	non-reserved	reserved
`NFC`	non-reserved	non-reserved	non-reserved	
`NFD`	non-reserved	non-reserved	non-reserved	
`NFKC`	non-reserved	non-reserved	non-reserved	
`NFKD`	non-reserved	non-reserved	non-reserved	
`NIL`		non-reserved	non-reserved	
`NO`	non-reserved	reserved	reserved	reserved
`NONE`	non-reserved (cannot be function or type)	reserved	reserved	
`NORMALIZE`	non-reserved (cannot be function or type)	reserved	reserved	
`NORMALIZED`	non-reserved	non-reserved	non-reserved	
`NOT`	reserved	reserved	reserved	reserved
`NOTHING`	non-reserved			
`NOTIFY`	non-reserved			
`NOTNULL`	reserved (can be function or type), requires `AS`			
`NOWAIT`	non-reserved			
`NTH_VALUE`		reserved	reserved	
`NTILE`		reserved	reserved	
`NULL`	reserved	reserved	reserved	reserved
`NULLABLE`		non-reserved	non-reserved	non-reserved
`NULLIF`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`NULLS`	non-reserved	non-reserved	non-reserved	
`NULL_ORDERING`		non-reserved	non-reserved	
`NUMBER`		non-reserved	non-reserved	non-reserved
`NUMERIC`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`OBJECT`	non-reserved	non-reserved	non-reserved	
`OCCURRENCE`		non-reserved	non-reserved	
`OCCURRENCES_REGEX`		reserved	reserved	
`OCTETS`		non-reserved	non-reserved	
`OCTET_LENGTH`		reserved	reserved	reserved
`OF`	non-reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`OFF`	non-reserved	non-reserved	non-reserved	
`OFFSET`	reserved, requires `AS`	reserved	reserved	
`OIDS`	non-reserved			
`OLD`	non-reserved	reserved	reserved	
`OMIT`	non-reserved	reserved	reserved	
`ON`	reserved, requires `AS`	reserved	reserved	reserved
`ONE`		reserved	reserved	
`ONLY`	reserved	reserved	reserved	reserved
`OPEN`		reserved	reserved	reserved
`OPERATOR`	non-reserved			
`OPTION`	non-reserved	non-reserved	non-reserved	reserved
`OPTIONS`	non-reserved	non-reserved	non-reserved	
`OR`	reserved	reserved	reserved	reserved
`ORDER`	reserved, requires `AS`	reserved	reserved	reserved
`ORDERING`		non-reserved	non-reserved	
`ORDINALITY`	non-reserved	non-reserved	non-reserved	
`OTHERS`	non-reserved	non-reserved	non-reserved	
`OUT`	non-reserved (cannot be function or type)	reserved	reserved	
`OUTER`	reserved (can be function or type)	reserved	reserved	reserved
`OUTPUT`		non-reserved	non-reserved	reserved
`OVER`	non-reserved, requires `AS`	reserved	reserved	
`OVERFLOW`		non-reserved	non-reserved	
`OVERLAPS`	reserved (can be function or type), requires `AS`	reserved	reserved	reserved
`OVERLAY`	non-reserved (cannot be function or type)	reserved	reserved	
`OVERRIDING`	non-reserved	non-reserved	non-reserved	
`OWNED`	non-reserved			
`OWNER`	non-reserved			
`P`		non-reserved	non-reserved	
`PAD`		non-reserved	non-reserved	reserved
`PARALLEL`	non-reserved			
`PARAMETER`	non-reserved	reserved	reserved	
`PARAMETER_MODE`		non-reserved	non-reserved	

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`PARAMETER_NAME`		non-reserved	non-reserved	
`PARAMETER_ORDINAL_POSITION`		non-reserved	non-reserved	
`PARAMETER_SPECIFIC_CATALOG`		non-reserved	non-reserved	
`PARAMETER_SPECIFIC_NAME`		non-reserved	non-reserved	
`PARAMETER_SPECIFIC_SCHEMA`		non-reserved	non-reserved	
`PARSER`	non-reserved			
`PARTIAL`	non-reserved	non-reserved	non-reserved	reserved
`PARTITION`	non-reserved	reserved	reserved	
`PARTITIONS`	non-reserved			
`PASCAL`		non-reserved	non-reserved	non-reserved
`PASS`		non-reserved	non-reserved	
`PASSING`	non-reserved	non-reserved	non-reserved	
`PASSTHROUGH`		non-reserved	non-reserved	
`PASSWORD`	non-reserved			
`PASSWORD_GRACE_TIME`	non-reserved			
`PASSWORD_LIFE_TIME`	non-reserved			
`PASSWORD_MIN_LEN`	non-reserved			
`PASSWORD_MIN_UNIQUE_CHARS`	non-reserved			
`PASSWORD_REQUIRE_COMPLEX`	non-reserved			
`PASSWORD_REUSE_MAX`	non-reserved			
`PASSWORD_REUSE_TIME`	non-reserved			
`PAST`		non-reserved	non-reserved	
`PATH`	non-reserved	non-reserved	non-reserved	
`PATTERN`		reserved	reserved	
`PER`		reserved	reserved	
`PERCENT`		reserved	reserved	
`PERCENTILE_CONT`		reserved	reserved	
`PERCENTILE_DISC`		reserved	reserved	
`PERCENT_RANK`		reserved	reserved	
`PERIOD`		reserved	reserved	
`PERMISSION`		non-reserved	non-reserved	
`PERMUTE`		non-reserved	non-reserved	
`PIPE`		non-reserved	non-reserved	
`PLACING`	reserved	non-reserved	non-reserved	
`PLAN`	non-reserved	non-reserved	non-reserved	
`PLANS`	non-reserved			
`PLI`		non-reserved	non-reserved	non-reserved
`POLICY`	non-reserved			
`PORTION`		reserved	reserved	

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`POSITION`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`POSITION_REGEX`		reserved	reserved	
`POWER`		reserved	reserved	
`PRECEDES`		reserved	reserved	
`PRECEDING`	non-reserved	non-reserved	non-reserved	
`PRECISION`	non-reserved (cannot be function or type), requires `AS`	reserved	reserved	reserved
`PREPARE`	non-reserved	reserved	reserved	reserved
`PREPARED`	non-reserved			
`PRESERVE`	non-reserved	non-reserved	non-reserved	reserved
`PREV`		non-reserved	non-reserved	
`PRIMARY`	reserved	reserved	reserved	reserved
`PRIOR`	non-reserved	non-reserved	non-reserved	reserved
`PRIVATE`		non-reserved	non-reserved	
`PRIVILEGES`	non-reserved	non-reserved	non-reserved	reserved
`PROCEDURAL`	non-reserved			
`PROCEDURE`	non-reserved	reserved	reserved	reserved
`PROCEDURES`	non-reserved			
`PROFILE`	non-reserved			
`PROGRAM`	non-reserved			
`PRUNE`		non-reserved	non-reserved	
`PTF`		reserved	reserved	
`PUBLIC`		non-reserved	non-reserved	reserved
`PUBLICATION`	non-reserved			
`QUOTE`	non-reserved			
`QUOTES`	non-reserved	non-reserved	non-reserved	
`RANGE`	non-reserved	reserved	reserved	
`RANK`		reserved	reserved	
`READ`	non-reserved	non-reserved	non-reserved	reserved
`READS`		reserved	reserved	
`REAL`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`REASSIGN`	non-reserved			
`RECHECK`	non-reserved			
`RECOVERY`		non-reserved	non-reserved	
`RECURSIVE`	non-reserved	reserved	reserved	
`REF`	non-reserved	reserved	reserved	
`REFERENCES`	reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`REFERENCING`	non-reserved	reserved	reserved	
`REFRESH`	non-reserved			
`REGR_AVGX`		reserved	reserved	
`REGR_AVGY`		reserved	reserved	
`REGR_COUNT`		reserved	reserved	
`REGR_INTERCEPT`		reserved	reserved	
`REGR_R2`		reserved	reserved	
`REGR_SLOPE`		reserved	reserved	
`REGR_SXX`		reserved	reserved	
`REGR_SXY`		reserved	reserved	
`REGR_SYY`		reserved	reserved	
`REINDEX`	non-reserved			
`RELATIVE`	non-reserved	non-reserved	non-reserved	reserved
`RELEASE`	non-reserved	reserved	reserved	
`RENAME`	non-reserved			
`REPEATABLE`	non-reserved	non-reserved	non-reserved	non-reserved
`REPLACE`	non-reserved			
`REPLICA`	non-reserved			
`REQUIRING`		non-reserved	non-reserved	
`RESET`	non-reserved			
`RESPECT`		non-reserved	non-reserved	
`RESTART`	non-reserved	non-reserved	non-reserved	
`RESTORE`		non-reserved	non-reserved	
`RESTRICT`	non-reserved	non-reserved	non-reserved	reserved
`RESULT`		reserved	reserved	
`RETURN`	non-reserved	reserved	reserved	
`RETURNED_CARDINALITY`		non-reserved	non-reserved	
`RETURNED_LENGTH`		non-reserved	non-reserved	non-reserved
`RETURNED_OCTET_LENGTH`		non-reserved	non-reserved	non-reserved
`RETURNED_SQLSTATE`		non-reserved	non-reserved	non-reserved
`RETURNING`	reserved, requires `AS`	non-reserved	non-reserved	
`RETURNS`	non-reserved	reserved	reserved	
`REVOKE`	non-reserved	reserved	reserved	reserved
`RIGHT`	reserved (can be function or type)	reserved	reserved	reserved
`ROLE`	non-reserved	non-reserved	non-reserved	
`ROLLBACK`	non-reserved	reserved	reserved	reserved
`ROLLUP`	non-reserved	reserved	reserved	
`ROUTINE`	non-reserved	non-reserved	non-reserved	

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`ROUTINES`	non-reserved			
`ROUTINE_CATALOG`		non-reserved	non-reserved	
`ROUTINE_NAME`		non-reserved	non-reserved	
`ROUTINE_SCHEMA`		non-reserved	non-reserved	
`ROW`	non-reserved (cannot be function or type)	reserved	reserved	
`ROWS`	non-reserved	reserved	reserved	reserved
`ROW_COUNT`		non-reserved	non-reserved	non-reserved
`ROW_NUMBER`		reserved	reserved	
`RPAD`		reserved		
`RTRIM`		reserved		
`RULE`	non-reserved			
`RUNNING`		reserved	reserved	
`SAVEPOINT`	non-reserved	reserved	reserved	
`SCALAR`	non-reserved	non-reserved	non-reserved	
`SCALE`		non-reserved	non-reserved	non-reserved
`SCHEMA`	non-reserved	non-reserved	non-reserved	reserved
`SCHEMAS`	non-reserved			
`SCHEMA_NAME`		non-reserved	non-reserved	non-reserved
`SCOPE`		reserved	reserved	
`SCOPE_CATALOG`		non-reserved	non-reserved	
`SCOPE_NAME`		non-reserved	non-reserved	
`SCOPE_SCHEMA`		non-reserved	non-reserved	
`SCROLL`	non-reserved	reserved	reserved	reserved
`SEARCH`	non-reserved	reserved	reserved	
`SECOND`	non-reserved, requires `AS`	reserved	reserved	reserved
`SECTION`		non-reserved	non-reserved	reserved
`SECURITY`	non-reserved	non-reserved	non-reserved	
`SEEK`		reserved	reserved	
`SELECT`	reserved	reserved	reserved	reserved
`SELECTIVE`		non-reserved	non-reserved	
`SELF`		non-reserved	non-reserved	
`SEMANTICS`		non-reserved	non-reserved	
`SENSITIVE`		reserved	reserved	
`SEQUENCE`	non-reserved	non-reserved	non-reserved	
`SEQUENCES`	non-reserved			
`SERIALIZABLE`	non-reserved	non-reserved	non-reserved	non-reserved
`SERVER`	non-reserved	non-reserved	non-reserved	
`SERVER_NAME`		non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`SESSION`	non-reserved	non-reserved	non-reserved	reserved
`SESSION_USER`	reserved	reserved	reserved	reserved
`SET`	non-reserved	reserved	reserved	reserved
`SETOF`	non-reserved (cannot be function or type)			
`SETS`	non-reserved	non-reserved	non-reserved	
`SHARE`	non-reserved			
`SHOW`	non-reserved	reserved	reserved	
`SIMILAR`	reserved (can be function or type)	reserved	reserved	
`SIMPLE`	non-reserved	non-reserved	non-reserved	
`SIN`		reserved	reserved	
`SINH`		reserved	reserved	
`SIZE`		non-reserved	non-reserved	reserved
`SKIP`	non-reserved	reserved	reserved	
`SMALLINT`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`SNAPSHOT`	non-reserved			
`SOME`	reserved	reserved	reserved	reserved
`SORT_DIRECTION`		non-reserved	non-reserved	
`SOURCE`	non-reserved	non-reserved	non-reserved	
`SPACE`		non-reserved	non-reserved	reserved
`SPECIFIC`		reserved	reserved	
`SPECIFICTYPE`		reserved	reserved	
`SPECIFIC_NAME`		non-reserved	non-reserved	
`SPLIT`	non-reserved			
`SQL`	non-reserved	reserved	reserved	reserved
`SQLCODE`				reserved
`SQLERROR`				reserved
`SQLEXCEPTION`		reserved	reserved	
`SQLSTATE`		reserved	reserved	reserved
`SQLWARNING`		reserved	reserved	
`SQRT`		reserved	reserved	
`STABLE`	non-reserved			
`STANDALONE`	non-reserved	non-reserved	non-reserved	
`START`	non-reserved	reserved	reserved	
`STATE`		non-reserved	non-reserved	
`STATEMENT`	non-reserved	non-reserved	non-reserved	
`STATIC`		reserved	reserved	

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
STATISTICS	non-reserved			
STDDEV_POP		reserved	reserved	
STDDEV_SAMP		reserved	reserved	
STDIN	non-reserved			
STDOUT	non-reserved			
STORAGE	non-reserved			
STORED	non-reserved			
STRICT	non-reserved			
STRING	non-reserved	non-reserved	non-reserved	
STRIP	non-reserved	non-reserved	non-reserved	
STRUCTURE		non-reserved	non-reserved	
STYLE		non-reserved	non-reserved	
SUBCLASS_ORIGIN		non-reserved	non-reserved	non-reserved
SUBMULTISET		reserved	reserved	
SUBSCRIPTION	non-reserved			
SUBSET		reserved	reserved	
SUBSTRING	non-reserved (cannot be function or type)	reserved	reserved	reserved
SUBSTRING_REGEX		reserved	reserved	
SUCCEEDS		reserved	reserved	
SUM		reserved	reserved	reserved
SUPPORT	non-reserved			
SYMMETRIC	reserved	reserved	reserved	
SYSID	non-reserved			
SYSTEM	non-reserved	reserved	reserved	
SYSTEM_TIME		reserved	reserved	
SYSTEM_USER	reserved	reserved	reserved	reserved
T		non-reserved	non-reserved	
TABLE	reserved	reserved	reserved	reserved
TABLES	non-reserved			
TABLESAMPLE	reserved (can be function or type)	reserved	reserved	
TABLESPACE	non-reserved			
TABLE_NAME		non-reserved	non-reserved	non-reserved
TAN		reserved	reserved	
TANH		reserved	reserved	
TARGET	non-reserved			
TEMP	non-reserved			
TEMPLATE	non-reserved			
TEMPORARY	non-reserved	non-reserved	non-reserved	reserved

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`TEXT`	non-reserved			
`THEN`	reserved	reserved	reserved	reserved
`THROUGH`		non-reserved	non-reserved	
`TIES`	non-reserved	non-reserved	non-reserved	
`TIME`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`TIMESTAMP`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`TIMEZONE_HOUR`		reserved	reserved	reserved
`TIMEZONE_MINUTE`		reserved	reserved	reserved
`TO`	reserved, requires `AS`	reserved	reserved	reserved
`TOKEN`		non-reserved	non-reserved	
`TOP_LEVEL_COUNT`		non-reserved	non-reserved	
`TRAILING`	reserved	reserved	reserved	reserved
`TRANSACTION`	non-reserved	non-reserved	non-reserved	reserved
`TRANSACTIONS_COMMITTED`		non-reserved	non-reserved	
`TRANSACTIONS_ROLLED_BACK`		non-reserved	non-reserved	
`TRANSACTION_ACTIVE`		non-reserved	non-reserved	
`TRANSFORM`	non-reserved	non-reserved	non-reserved	
`TRANSFORMS`		non-reserved	non-reserved	
`TRANSLATE`		reserved	reserved	reserved
`TRANSLATE_REGEX`		reserved	reserved	
`TRANSLATION`		reserved	reserved	reserved
`TREAT`	non-reserved (cannot be function or type)	reserved	reserved	
`TRIGGER`	non-reserved	reserved	reserved	
`TRIGGER_CATALOG`		non-reserved	non-reserved	
`TRIGGER_NAME`		non-reserved	non-reserved	
`TRIGGER_SCHEMA`		non-reserved	non-reserved	
`TRIM`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`TRIM_ARRAY`		reserved	reserved	
`TRUE`	reserved	reserved	reserved	reserved
`TRUNCATE`	non-reserved	reserved	reserved	
`TRUSTED`	non-reserved			
`TYPE`	non-reserved	non-reserved	non-reserved	non-reserved
`TYPES`	non-reserved			
`UESCAPE`	non-reserved	reserved	reserved	

Key Word	**PostgreSQL**	**SQL:2023**	**SQL:2016**	**SQL-92**
`UNBOUNDED`	non-reserved	non-reserved	non-reserved	
`UNCOMMITTED`	non-reserved	non-reserved	non-reserved	non-reserved
`UNCONDITIONAL`	non-reserved	non-reserved	non-reserved	
`UNDER`		non-reserved	non-reserved	
`UNENCRYPTED`	non-reserved			
`UNION`	reserved, requires `AS`	reserved	reserved	reserved
`UNIQUE`	reserved	reserved	reserved	reserved
`UNKNOWN`	non-reserved	reserved	reserved	reserved
`UNLIMITED`	non-reserved			
`UNLINK`		non-reserved	non-reserved	
`UNLISTEN`	non-reserved			
`UNLOCK`	non-reserved			
`UNLOGGED`	non-reserved			
`UNMATCHED`		non-reserved	non-reserved	
`UNNAMED`		non-reserved	non-reserved	non-reserved
`UNNEST`		reserved	reserved	
`UNTIL`	non-reserved			
`UNTYPED`		non-reserved	non-reserved	
`UPDATE`	non-reserved	reserved	reserved	reserved
`UPPER`		reserved	reserved	reserved
`URI`		non-reserved	non-reserved	
`USAGE`		non-reserved	non-reserved	reserved
`USER`	reserved	reserved	reserved	reserved
`USER_DEFINED_TYPE_CATALOG`		non-reserved	non-reserved	
`USER_DEFINED_TYPE_CODE`		non-reserved	non-reserved	
`USER_DEFINED_TYPE_NAME`		non-reserved	non-reserved	
`USER_DEFINED_TYPE_SCHEMA`		non-reserved	non-reserved	
`USER_INACTIVE_TIME`	non-reserved			
`USING`	reserved	reserved	reserved	reserved
`UTF16`		non-reserved	non-reserved	
`UTF32`		non-reserved	non-reserved	
`UTF8`		non-reserved	non-reserved	
`VACUUM`	non-reserved			
`VALID`	non-reserved	non-reserved	non-reserved	
`VALIDATE`	non-reserved			
`VALIDATOR`	non-reserved			
`VALUE`	non-reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`VALUES`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`VALUE_OF`		reserved	reserved	
`VARBINARY`		reserved	reserved	
`VARCHAR`	non-reserved (cannot be function or type)	reserved	reserved	reserved
`VARIADIC`	reserved			
`VARYING`	non-reserved, requires `AS`	reserved	reserved	reserved
`VAR_POP`		reserved	reserved	
`VAR_SAMP`		reserved	reserved	
`VERBOSE`	reserved (can be function or type)			
`VERSION`	non-reserved	non-reserved	non-reserved	
`VERSIONING`		reserved	reserved	
`VIEW`	non-reserved	non-reserved	non-reserved	reserved
`VIEWS`	non-reserved			
`VOLATILE`	non-reserved			
`WHEN`	reserved	reserved	reserved	reserved
`WHENEVER`		reserved	reserved	reserved
`WHERE`	reserved, requires `AS`	reserved	reserved	reserved
`WHITESPACE`	non-reserved	non-reserved	non-reserved	
`WIDTH_BUCKET`		reserved	reserved	
`WINDOW`	reserved, requires `AS`	reserved	reserved	
`WITH`	reserved, requires `AS`	reserved	reserved	reserved
`WITHIN`	non-reserved, requires `AS`	reserved	reserved	
`WITHOUT`	non-reserved, requires `AS`	reserved	reserved	
`WORK`	non-reserved	non-reserved	non-reserved	reserved
`WRAPPER`	non-reserved	non-reserved	non-reserved	
`WRITE`	non-reserved	non-reserved	non-reserved	reserved
`XML`	non-reserved	reserved	reserved	
`XMLAGG`		reserved	reserved	
`XMLATTRIBUTES`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLBINARY`		reserved	reserved	
`XMLCAST`		reserved	reserved	

Key Word	PostgreSQL	SQL:2023	SQL:2016	SQL-92
`XMLCOMMENT`		reserved	reserved	
`XMLCONCAT`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLDECLARATION`		non-reserved	non-reserved	
`XMLDOCUMENT`		reserved	reserved	
`XMLELEMENT`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLEXISTS`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLFOREST`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLITERATE`		reserved	reserved	
`XMLNAMESPACES`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLPARSE`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLPI`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLQUERY`		reserved	reserved	
`XMLROOT`	non-reserved (cannot be function or type)			
`XMLSCHEMA`		non-reserved	non-reserved	
`XMLSERIALIZE`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLTABLE`	non-reserved (cannot be function or type)	reserved	reserved	
`XMLTEXT`		reserved	reserved	
`XMLVALIDATE`		reserved	reserved	
`YEAR`	non-reserved, requires `AS`	reserved	reserved	reserved
`YES`	non-reserved	non-reserved	non-reserved	
`ZONE`	non-reserved	non-reserved	non-reserved	reserved

Приложение D. Соответствие стандарту SQL

В этом разделе в общих чертах отмечается, в какой степени Postgres Pro соответствует текущему стандарту SQL. Следующая информация не является официальным утверждением о соответствии, а представляет только основные аспекты на уровне детализации, достаточно полезном и целесообразном для пользователей.

Формально стандарт SQL называется ISO/IEC 9075 «Database Language SQL» (Язык баз данных SQL). Время от времени выпускается обновлённая версия стандарта, последняя вышла в 2023 г. Она получила обозначение ISO/IEC 9075:2023 или просто SQL:2023. До этого были выпущены версии SQL:2016, SQL:2011, SQL:2008, SQL:2006, SQL:2003, SQL:1999 и SQL-92. Каждая следующая версия заменяет предыдущую, так что утверждение о совместимости с предыдущими версиями не имеет большой ценности. Разработчики Postgres Pro стремятся обеспечить совместимость с последней официальной версией стандарта, оставаясь при этом в рамках традиционной функциональности и здравого смысла. Postgres Pro реализует большую часть требуемой стандартом функциональности, хотя иногда с немного другими функциями или синтаксисом. Можно ожидать, что со временем степень совместимости будет увеличиваться.

SQL-92 определяет три уровня функциональной совместимости: начальный (Entry), промежуточный (Intermediate) и полный (Full). Большинство СУБД заявляют о совместимости со стандартом SQL только на начальном уровне, так как полный набор возможностей на промежуточном и полном уровнях либо слишком велик, либо конфликтует с ранее принятым поведением.

Начиная с SQL:1999, вместо трёх чрезмерно пространных уровней SQL-92 в стандарте SQL определено множество отдельных функциональных возможностей. Большое его подмножество представляет «Основную» функциональность, которую должны обеспечивать все совместимые с SQL реализации. Поддержка остальных возможностей не является обязательной.

Описание стандарта разделяется на несколько частей, каждая из которых также имеет короткое имя и номер:

- ISO/IEC 9075-1 Структура (SQL/Framework)
- ISO/IEC 9075-2 Основа (SQL/Foundation)
- ISO/IEC 9075-3 Интерфейс уровня вызовов (SQL/CLI)
- ISO/IEC 9075-4 Модули постоянного хранения (SQL/PSM)
- ISO/IEC 9075-9 Управление внешними данными (SQL/MED)
- ISO/IEC 9075-10 Привязки объектных языков (SQL/OLB)
- ISO/IEC 9075-11 Схемы информации и определений (SQL/Schemata)
- ISO/IEC 9075-13 Программы и типы, использующие язык Java (SQL/JRT)
- ISO/IEC 9075-14 Спецификации, связанные с XML (SQL/XML)
- ISO/IEC 9075-15 Многомерные массивы (SQL/MDA)
- ISO/IEC 9075-16 Запросы графов свойств (SQL/PGQ)

Обратите внимание, что некоторые номера этих частей не используются (или больше не используются).

Ядро Postgres Pro реализует части 1, 2, 9, 11 и 14. Часть 3 реализуется драйвером ODBC, а часть 13 — подключаемым расширением PL/Java, но точное соответствие этих компонентов стандарту на данный момент не проверено. Части 4, 10, 15 и 16 в Postgres Pro в настоящее время не реализованы.

Postgres Pro поддерживает почти все основные возможности стандарта SQL:2023. Из 177 обязательных возможностей, которые требуются для полного соответствия «Основной» функциональности, Postgres Pro обеспечивает совместимость как минимум для 170. Кроме того, он реализует длинный список необязательных возможностей. Следует отметить, что на время написания этой документации ни одна существующая СУБД не заявила о полном соответствии «Основной» функциональности SQL:2023.

В следующих двух разделах мы представляем список возможностей, которые поддерживает Postgres Pro, и список возможностей, определённых в SQL:2023, которые ещё не поддерживаются в Postgres Pro. Оба эти списка носят приблизительный характер: какая-то возможность, отмеченная как поддерживаемая, может отличаться от стандарта в деталях, и напротив, для какой-то неподдерживаемой возможности могут быть реализованы ключевые компоненты. Наиболее точная информация о том, что работает, а что нет, содержится в основной документации.

Примечание

Коды возможностей, содержащие знак минус, обозначают подчинённые возможности. При этом, если какая-либо одна подчинённая возможность не поддерживается, основная возможность так же не будет поддерживаться, даже если реализованы все остальные на подуровне.

D.1. Поддерживаемые возможности

Идентификатор	**Основа?**	**Описание**	**Комментарий**
B012		Встроенный C	
B021		Непосредственный SQL	
B128		Язык подпрограмм — SQL	
E011	Основа	Числовые типы данных	
E011-01	Основа	Типы данных INTEGER и SMALLINT	
E011-02	Основа	Типы данных REAL, DOUBLE PRECISION и FLOAT	
E011-03	Основа	Типы данных DECIMAL и NUMERIC	
E011-04	Основа	Арифметические операторы	
E011-05	Основа	Числовые сравнения	
E011-06	Основа	Неявные преобразования между числовыми типами данных	
E021	Основа	Символьные типы данных	
E021-01	Основа	Тип данных CHARACTER	
E021-02	Основа	Тип данных CHARACTER VARYING	

Идентификатор	Основа?	Описание	Комментарий
E021-03	Основа	Символьные строки	
E021-04	Основа	Функция CHARACTER_LENGTH	убирает завершающие пробелы из значений CHARACTER перед подсчётом символов
E021-05	Основа	Функция OCTET_LENGTH	
E021-06	Основа	Функция SUBSTRING	
E021-07	Основа	Конкатенация символьных строк	
E021-08	Основа	Функции UPPER и LOWER	
E021-09	Основа	Функция TRIM	
E021-10	Основа	Неявные преобразования между типами символьных строк	
E021-11	Основа	Функция POSITION	
E021-12	Основа	Сравнения символов	
E031	Основа	Идентификаторы	
E031-01	Основа	Идентификаторы с разделителями	
E031-02	Основа	Идентификаторы в нижнем регистре	
E031-03	Основа	Завершающее подчёркивание	
E051	Основа	Базовое определение запросов	
E051-01	Основа	SELECT DISTINCT	
E051-02	Основа	Предложение GROUP BY	
E051-04	Основа	GROUP BY может содержать столбцы не из <списка выборки>	
E051-05	Основа	Элементы списка выборки могут переименовываться	
E051-06	Основа	Предложение HAVING	
E051-07	Основа	Дополнение * в списке выборки	
E051-08	Основа	Корреляционные имена в предложении FROM	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
E051-09	Основа	Переименование столбцов в предложении FROM	
E061	Основа	Базовые предикаты и условия поиска	
E061-01	Основа	Предикат сравнения	
E061-02	Основа	Предикат BETWEEN	
E061-03	Основа	Предикат IN со списком значений	
E061-04	Основа	Предикат LIKE	
E061-05	Основа	Предложение ESCAPE в предикате LIKE	
E061-06	Основа	Предикат NULL	
E061-07	Основа	Предикаты количественного сравнения	
E061-08	Основа	Предикат EXISTS	
E061-09	Основа	Подзапросы в предикате сравнения	
E061-11	Основа	Подзапросы в предикате IN	
E061-12	Основа	Подзапросы в предикате количественного сравнения	
E061-13	Основа	Коррелирующие подзапросы	
E061-14	Основа	Условие поиска	
E071	Основа	Простые выражения с запросами	
E071-01	Основа	Табличный оператор UNION DISTINCT	
E071-02	Основа	Табличный оператор UNION ALL	
E071-03	Основа	Табличный оператор EXCEPT DISTINCT	
E071-05	Основа	Столбцы, объединяемые табличными операторами, могут иметь разные типы данных	
E071-06	Основа	Табличные операторы в подзапросах	
E081	Основа	Основные права доступа	
E081-01	Основа	Право на SELECT	

Идентификатор	Основа?	Описание	Комментарий
E081-02	Основа	Право на DELETE	
E081-03	Основа	Право на INSERT на уровне таблицы	
E081-04	Основа	Право на UPDATE на уровне таблицы	
E081-05	Основа	Право на UPDATE на уровне столбцов	
E081-06	Основа	Право REFERENCES на уровне таблицы	
E081-07	Основа	Право REFERENCES на уровне столбцов	
E081-08	Основа	Предложение WITH GRANT OPTION	
E081-09	Основа	Право USAGE	
E081-10	Основа	Право на EXECUTE	
E091	Основа	Функции множеств	
E091-01	Основа	AVG	
E091-02	Основа	COUNT	
E091-03	Основа	MAX	
E091-04	Основа	MIN	
E091-05	Основа	SUM	
E091-06	Основа	Дополнение ALL	
E091-07	Основа	Дополнение DISTINCT	
E101	Основа	Базовая обработка данных	
E101-01	Основа	Оператор INSERT	
E101-03	Основа	Оператор UPDATE с критерием отбора	
E101-04	Основа	Оператор DELETE с критерием отбора	
E111	Основа	Оператор SELECT, возвращающий одну строку	
E121	Основа	Базовая поддержка курсоров	

Идентификатор	Основа?	Описание	Комментарий
E121-01	Основа	DECLARE CURSOR	
E121-02	Основа	Столбцы ORDER BY, отсутствующие в списке выборки	
E121-03	Основа	Выражения значений в предложении ORDER BY	
E121-04	Основа	Оператор OPEN	
E121-06	Основа	Оператор UPDATE с позиционированием	
E121-07	Основа	Оператор DELETE с позиционированием	
E121-08	Основа	Оператор CLOSE	
E121-10	Основа	Оператор FETCH с неявным NEXT	
E121-17	Основа	Курсоры WITH HOLD	
E131	Основа	Поддержка NULL (NULL вместо значений)	
E141	Основа	Основные ограничения целостности	
E141-01	Основа	Ограничения NOT NULL	
E141-02	Основа	Ограничения UNIQUE столбцов NOT NULL	
E141-03	Основа	Ограничения PRIMARY KEY	
E141-04	Основа	Базовое ограничение FOREIGN KEY без действия (NO ACTION) по умолчанию и для операций удаления со ссылками, и для операций изменения со ссылками	
E141-06	Основа	Ограничения CHECK	
E141-07	Основа	Значения столбцов по умолчанию	
E141-08	Основа	NOT NULL распространяется на PRIMARY KEY	
E141-10	Основа	Имена во внешнем ключе могут указываться в любом порядке	
E151	Основа	Поддержка транзакций	
E151-01	Основа	Оператор COMMIT	
E151-02	Основа	Оператор ROLLBACK	
E152	Основа	Базовый оператор SET TRANSACTION	

Идентификатор	Основа?	Описание	Комментарий
E152-01	Основа	Оператор SET TRANSACTION: предложение ISOLATION LEVEL SERIALIZABLE	
E152-02	Основа	Оператор SET TRANSACTION: предложения READ ONLY и READ WRITE	
E153	Основа	Запросы, изменяющие данные, с подзапросами	
E161	Основа	Комментарии SQL, начинающиеся с двух минусов	
E171	Основа	Поддержка SQLSTATE	
E182	Основа	Привязки для системных языков	
F021	Основа	Основная информационная схема	
F021-01	Основа	Представление COLUMNS	
F021-02	Основа	Представление TABLES	
F021-03	Основа	Представление VIEWS	
F021-04	Основа	Представление TABLE_CONSTRAINTS	
F021-05	Основа	Представление REFERENTIAL_CONSTRAINTS	
F021-06	Основа	Представление CHECK_CONSTRAINTS	
F031	Основа	Базовые манипуляции со схемой	
F031-01	Основа	Оператор CREATE TABLE создаёт хранимые основные таблицы	
F031-02	Основа	Представление CREATE VIEW	
F031-03	Основа	Оператор GRANT	
F031-04	Основа	Оператор ALTER TABLE: предложение ADD COLUMN	
F031-13	Основа	Оператор DROP TABLE: предложение RESTRICT	
F031-16	Основа	Оператор DROP VIEW: предложение RESTRICT	
F031-19	Основа	Оператор REVOKE: предложение RESTRICT	
F032		Каскадное удаление (CASCADE)	
F033		Оператор ALTER TABLE: предложение DROP COLUMN	
F034		Расширенный оператор REVOKE	

Идентификатор	Основа?	Описание	Комментарий
F035		REVOKE with CASCADE	
F036		Оператор REVOKE, выполняемый не владельцем	
F037		Оператор REVOKE: предложение GRANT OPTION FOR	
F038		REVOKE права WITH GRANT OPTION	
F041	Основа	Базовое соединение таблиц	
F041-01	Основа	Внутреннее соединение (но не обязательно с ключевым словом INNER)	
F041-02	Основа	Ключевое слово INNER	
F041-03	Основа	LEFT OUTER JOIN	
F041-04	Основа	RIGHT OUTER JOIN	
F041-05	Основа	Внешние соединения могут быть вложенными	
F041-07	Основа	Внутренняя таблица с левой или правой стороны внешнего соединения может также участвовать во внутреннем соединении	
F041-08	Основа	Поддерживаются все операторы сравнения (а не только =)	
F051	Основа	Базовая поддержка даты и времени	
F051-01	Основа	Тип данных DATE (включая поддержку строк DATE)	
F051-02	Основа	Тип данных TIME (включая поддержку строк TIME) с точностью до секунд как минимум с 0 знаков после запятой	
F051-03	Основа	Тип данных TIMESTAMP (включая поддержку строк TIMESTAMP) с точностью до секунд как минимум с 0 и 6 знаками после запятой	
F051-04	Основа	Предикаты сравнения с типами данных DATE, TIME и TIMESTAMP	
F051-05	Основа	Явное приведение (CAST) между типами даты/времени и типами символьных строк	
F051-06	Основа	CURRENT_DATE	
F051-07	Основа	LOCALTIME	
F051-08	Основа	LOCALTIMESTAMP	
F052		Арифметика с интервалами и датами/временем	
F053		Предикат OVERLAPS	
F081	Основа	UNION и EXCEPT в представлениях	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
F111		Уровни изоляции, отличные от SERIALIZABLE	
F112		Уровень изоляции READ UNCOMMITTED	
F113		Уровень изоляции READ COMMITTED	
F114		Уровень изоляции REPEATABLE READ	
F131	Основа	Операции группировки	
F131-01	Основа	Предложения WHERE, GROUP BY и HAVING, поддерживаемые в запросах со сгруппированными представлениями	
F131-02	Основа	Поддержка нескольких таблиц в запросах со сгруппированными представлениями	
F131-03	Основа	Поддержка функций множеств в запросах со сгруппированными представлениями	
F131-04	Основа	Подзапросы с предложениями GROUP BY и HAVING и сгруппированные представления	
F131-05	Основа	SELECT, возвращающий одну строку, с предложениями GROUP BY и HAVING и сгруппированными представлениями	
F171		Несколько схем для одного пользователя	
F181	Основа	Поддержка множества модулей	
F191		Действия при удалении со ссылками	
F200		Оператор TRUNCATE TABLE	
F201	Основа	Функция CAST	
F202		TRUNCATE TABLE: возможность перезапуска идентифицирующего столбца	
F221	Основа	Явные значения по умолчанию	
F222		Оператор INSERT: предложение DEFAULT VALUES	
F231		Таблицы прав	
F251		Поддержка доменов	
F261	Основа	Выражение CASE	
F261-01	Основа	Простой оператор CASE	
F261-02	Основа	Оператор CASE с условиями	
F261-03	Основа	NULLIF	
F261-04	Основа	COALESCE	
F262		Расширенные выражения CASE	
F271		Составные строки символов	
F281		Улучшенный оператор LIKE	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
F292		Обработка NULL в ограничениях UNIQUE	
F302		Табличный оператор INTERSECT	
F303		Табличный оператор INTERSECT DISTINCT	
F304		Табличный оператор EXCEPT ALL	
F305		Табличный оператор INTERSECT ALL	
F311	Основа	Оператор определения схемы	
F311-01	Основа	CREATE SCHEMA	
F311-02	Основа	CREATE TABLE для хранимых основных таблиц	
F311-03	Основа	CREATE VIEW	
F311-04	Основа	CREATE VIEW: WITH CHECK OPTION	
F311-05	Основа	Оператор GRANT	
F312		Оператор MERGE	
F313		Расширенный оператор MERGE	
F314		Оператор MERGE с ветвью DELETE	
F321		Авторизация пользователей	
F341		Таблицы использования	
F361		Поддержка подпрограмм	
F381		Расширенные манипуляции со схемой	
F382		Изменение типа данных столбцов	
F383		Предложение, устанавливающее NOT NULL для столбца	
F384		Предложение удаления свойства идентифицирующего столбца	
F385		Предложение удаления выражения, генерирующего значения столбца	
F386		Предложение установления генерирования значений идентифицирующего столбца	
F387		Оператор ALTER TABLE: предложение ALTER COLUMN	
F388		Оператор ALTER TABLE: предложение ADD/DROP CONSTRAINT	
F391		Длинные идентификаторы	
F392		Спецсимволы Unicode в идентификаторах	
F393		Спецсимволы Unicode в текстовых строках	
F394		Необязательное указание нормальной формы	
F401		Расширенное соединение таблиц	
F402		Соединения по именам столбцов для больших объектов, массивов и мультимножеств	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
F404		Табличная переменная с именами общих столбцов	
F405		NATURAL JOIN	
F406		FULL OUTER JOIN	
F407		CROSS JOIN	
F411		Указание часового пояса	отличия в интерпретации строкового представления
F421		Национальные символы	
F431		Прокручиваемые курсоры только для чтения	
F432		FETCH с явным NEXT	
F433		FETCH FIRST	
F434		FETCH LAST	
F435		FETCH PRIOR	
F436		FETCH ABSOLUTE	
F437		FETCH RELATIVE	
F438		Прокручиваемые курсоры	
F441		Расширенная поддержка функций множеств	
F442		Смешанные ссылки на столбцы в функциях множеств	
F471	Основа	Скалярные значения подзапросов	
F481	Основа	Расширенный предикат NULL	
F491		Управление ограничениями	
F501	Основа	Представления возможностей и совместимости	
F501-01	Основа	Представление SQL_FEATURES	
F501-02	Основа	Представление SQL_SIZING	
F502		Таблицы расширенной документации	
F531		Временные таблицы	
F555		Дополнительная точность в секундах	
F561		Полные выражения значений	
F571		Проверки значений истинности	
F591		Производные таблицы	
F611		Типы данных для индикаторов	
F641		Конструкторы строк и таблиц	
F651		Дополнения имён каталогов	
F661		Простые таблицы	
F672		Ограничения CHECK с текущим временем	
F690		Поддержка правил сортировки	
F692		Расширенная поддержка правил сортировки	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
F701		Действия при обновлении со ссылками	
F711		ALTER для домена	
F731		Права на INSERT для столбцов	
F751		Усовершенствования CHECK для представлений	
F761		Управление сеансом	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		Управление соединением	
F781		Самоссылающиеся операции	
F791		Нечувствительные курсоры	
F801		Полные функции множеств	
F850		Предложение ORDER BY на верхнем уровне в выражении запроса	
F851		ORDER BY в подзапросах	
F852		Предложение ORDER BY на верхнем уровне в представлении	
F855		Вложенное предложение ORDER BY в выражении запроса	
F856		Вложенное предложение FETCH FIRST в выражении запроса	
F857		Предложение FETCH FIRST на верхнем уровне в выражении запроса	
F858		FETCH FIRST в подзапросах	
F859		Предложение FETCH FIRST на верхнем уровне в представлениях	
F860		Количество строк в предложении FETCH FIRST	
F861		Предложение OFFSET на верхнем уровне в выражении запроса	
F862		Предложение OFFSET в подзапросах	
F863		Вложенное предложение OFFSET в выражении запроса	
F864		Предложение OFFSET на верхнем уровне в представлениях	
F865		Указание числа строк с предложением OFFSET	
F867		Предложение FETCH FIRST: параметр WITH TIES	
F868		ORDER BY в сгруппированной таблице	
F869		Информация о реализации SQL	
S071		SQL-пути при разрешении имён функций и типов	
S090		Базовая поддержка массивов	
S092		Массивы пользовательских типов	
S095		Конструкторы массива из запроса	
S096		Необязательное указание границ массива	
S098		ARRAY_AGG	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
S099		Выражения с массивами	
S111		ONLY в выражениях запросов	
S201		Вызываемые из SQL подпрограммы, работающие с массивами	
S203		Массивы в параметрах	
S204		Массивы в качестве типа результата функций	
S211		Пользовательские функции приведений	
S301		Расширенный UNNEST	
S404		TRIM_ARRAY	
T031		Тип данных BOOLEAN	
T054		GREATEST и LEAST	разная обработка значений NULL
T055		Функции дополнения строк	
T056		Функции TRIM, усекающие по несколько символов	
T061		Поддержка UCS	
T071		Тип данных BIGINT	
T081		Необязательное указание максимальной длины типов строк	
T121		WITH (без RECURSIVE) в выражении запроса	
T122		WITH (с RECURSIVE) в подзапросе	
T131		Рекурсивный запрос	
T132		Рекурсивный запрос в подзапросе	
T133		Расширенные значения для пометки циклов	
T141		Предикат SIMILAR	
T151		Предикат DISTINCT	
T152		Предикат DISTINCT с отрицанием	
T171		Предложение LIKE в определении таблицы	
T172		Предложение подзапроса AS в определении таблицы	
T173		Расширенное предложение LIKE в определении таблицы	
T174		Идентифицирующие столбцы	
T177		Поддержка генераторов последовательностей: возможность простого перезапуска	
T178		Идентифицирующие столбцы: возможность простого перезапуска	
T191		Действие RESTRICT при нарушении ссылок	
T201		Сравнимые типы данных для ссылочных ограничений	
T212		Расширенные возможности триггеров	
T213		Триггеры INSTEAD OF	
T214		Триггеры BEFORE	
T215		Триггеры AFTER	

Идентификатор	Основа?	Описание	Комментарий
T216		Возможность задавать условие поиска, которое должно быть истинным перед вызовом триггера	
T217		Право TRIGGER	
T241		Оператор START TRANSACTION	
T261		Сцеплённые транзакции	
T271		Точки сохранения	
T281		Право SELECT на уровне столбцов	
T285		Улучшения имён производных столбцов	
T312		Функция OVERLAY	
T321-01	Основа	Пользовательские функции без перегрузки	
T321-02	Основа	Пользовательские хранимые процедуры без перегрузки	
T321-03	Основа	Вызов функций	
T321-04	Основа	Оператор CALL	
T321-05	Основа	Оператор RETURN	
T321-06	Основа	Представление ROUTINES	
T321-07	Основа	Представление PARAMETERS	
T323		Явное управление безопасностью внешних подпрограмм	
T325		Дополненные указания параметров SQL	
T331		Базовые роли	
T332		Расширенные роли	
T341		Перегрузка вызываемых из SQL функций и процедур	
T351		Блочные комментарии	
T431		Расширенные возможности группирования	
T432		Вложения и конкатенация GROUPING SETS	
T433		Функция GROUPING с несколькими аргументами	
T434		GROUP BY DISTINCT	
T441		Функции ABS и MOD	
T461		Симметричный предикат BETWEEN	
T491		Производная таблица LATERAL	
T501		Улучшенный предикат EXISTS	
T521		Именованные аргументы в операторе CALL	
T523		Значения по умолчанию для INOUT-параметров процедур, вызываемых из SQL	
T524		Именованные аргументы при вызове подпрограмм не с применением оператора CALL	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
T525		Значения по умолчанию для параметров функций, вызываемых из SQL	
T551		Необязательные ключевые слова, подразумеваемые синтаксисом по умолчанию	
T581		Функция подстроки по регулярному выражению	
T591		Ограничения UNIQUE для столбцов, принимающих NULL	
T611		Элементарные операции OLAP	
T612		Расширенные операции OLAP	
T613		Получение выборки	
T614		Функция NTILE	
T615		Функции LEAD и LAG	
T617		Функции FIRST_VALUE и LAST_VALUE	
T620		Предложение WINDOW: параметр GROUPS	
T621		Дополнительные численные функции	
T622		Тригонометрические функции	
T623		Общие логарифмические функции	
T624		Общепринятые логарифмические функции	
T626		ANY_VALUE	
T627		COUNT DISTINCT в рамке окна	
T631	Основа	Предикат IN с одним элементом списка	
T651		Операторы модификации схемы SQL в SQL-подпрограммах	
T653		Операторы модификации схемы SQL во внешних подпрограммах	
T655		Циклически зависимые подпрограммы	
T661		Недесятичные целочисленные константы	
T662		Подчёркивания в числовых литералах	
T670		Совместное использование операторов изменения данных и схем	
T803		JSON на основе строк	
T811		Базовые функции-конструкторы SQL/JSON	
T812		SQL/JSON: JSON_OBJECTAGG	
T813		SQL/JSON: JSON_ARRAYAGG с ORDER BY	
T814		Двоеточие в JSON_OBJECT или JSON_OBJECTAGG	
T821		Основные операторы запросов SQL/JSON	
T822		SQL/JSON: предикат IS JSON WITH UNIQUE KEYS	
T823		SQL/JSON: предложение PASSING	
T824		JSON_TABLE: предложение PLAN с указанием плана	
T825		SQL/JSON: предложения ON EMPTY и ON ERROR	

Идентификатор	Основа?	Описание	Комментарий
T826		Произвольные выражения значений в предложениях ON ERROR и ON EMPTY	
T827		JSON_TABLE: одноуровневые предложения NESTED COLUMNS	
T828		JSON_QUERY	
T829		JSON_QUERY: возможности обёртывания массивов	
T830		Требование уникальности ключей в функциях-конструкторах SQL/JSON	
T831		Язык путей SQL/JSON: строгий режим	
T832		Язык путей SQL/JSON: методы элементов	
T833		Язык путей SQL/JSON: множественные индексы	
T834		Язык путей SQL/JSON: обращение к членам по звёздочке	
T835		Язык путей SQL/JSON: выражения фильтров	
T836		Язык путей SQL/JSON: предикат «начинается с»	
T837		Язык путей SQL/JSON: предикат regex_like	
T838		JSON_TABLE: предложение PLAN DEFAULT	
T840		Целые шестнадцатеричные литералы в языке путей SQL/JSON	
T851		SQL/JSON: необязательные ключевые слова, подразумеваемые синтаксисом по умолчанию	
T865		Метод элементов SQL/JSON: bigint()	
T866		Метод элементов SQL/JSON: boolean()	
T867		Метод элементов SQL/JSON: date()	
T868		Метод элементов SQL/JSON: decimal()	
T869		Метод элементов SQL/JSON: decimal() с точностью и масштабом	
T870		Метод элементов SQL/JSON: integer()	
T871		Метод элементов SQL/JSON: number()	
T872		Метод элементов SQL/JSON: string()	
T873		Метод элементов SQL/JSON: time()	
T874		Метод элементов SQL/JSON: time_tz()	
T875		Метод элементов SQL/JSON: точность time	
T876		Метод элементов SQL/JSON: timestamp()	
T877		Метод элементов SQL/JSON: timestamp_tz()	
T878		Метод элементов SQL/JSON: точность timestamp	
T879		JSON в операциях равенства	с jsonb
T880		JSON в операциях группировки	с jsonb
X010		Тип XML	
X011		Массивы типа XML	
X014		Атрибуты типа XML	
X016		Хранимые значения XML	

Идентификатор	Основа?	Описание	Комментарий
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg: параметр ORDER BY	
X036		XMLComment	
X037		XMLPI	
X038		XMLText	поддерживается, за исключением RETURNING
X040		Базовое отображение таблиц	
X041		Базовое отображение таблиц: значение NULL отсутствует	
X042		Базовое отображение таблиц: NULL в виде nil	
X043		Базовое отображение таблиц: таблица в виде леса элементов	
X044		Базовое отображение таблиц: таблица в виде элемента	
X045		Базовое отображение таблиц: с целевым пространством имён	
X046		Базовое отображение таблиц: отображение данных	
X047		Базовое отображение таблиц: отображение метаданных	
X048		Базовое отображение таблиц: кодирование двоичных строк в base64	
X049		Базовое отображение таблиц: кодирование двоичных строк в шестнадцатеричном виде	
X050		Расширенное отображение таблиц	
X051		Расширенное отображение таблиц: значение NULL отсутствует	
X052		Расширенное отображение таблиц: NULL в виде nil	
X053		Расширенное отображение таблиц: таблица в виде леса элементов	
X054		Расширенное отображение таблиц: таблица в виде элемента	
X055		Расширенное отображение таблиц: с целевым пространством имён	
X056		Расширенное отображение таблиц: отображение данных	
X057		Расширенное отображение таблиц: отображение метаданных	
X058		Расширенное отображение таблиц: кодирование двоичных строк в base64	
X059		Расширенное отображение таблиц: кодирование двоичных строк в шестнадцатеричном виде	

Идентификатор	Основа?	Описание	Комментарий
X060		XMLParse: ввод символьных строк и вариант CONTENT	
X061		XMLParse: ввод символьных строк и вариант DOCUMENT	
X069		XMLSerialize: INDENT	
X070		XMLSerialize: сериализация символьных строк и вариант CONTENT	
X071		XMLSerialize: сериализация символьных строк и вариант DOCUMENT	
X072		XMLSerialize: сериализация символьных строк	
X090		Предикат XML-документа	
X120		XML в параметрах SQL-подпрограмм	
X121		XML в параметрах внешних подпрограмм	
X221		Механизм передачи XML BY VALUE	
X301		XMLTable: указание списка производных столбцов	
X302		XMLTable: указание столбца нумерации	
X303		XMLTable: указание значения столбца по умолчанию	
X304		XMLTable: передача контекста	требуется XML DOCUMENT
X400		Сопоставление имён и идентификаторов	
X410		Изменение типа данных столбца: поддержка типа XML	

D.2. Неподдерживаемые возможности

Следующие возможности, описанные в SQL:2023, не реализованы в этом выпуске Postgres Pro. В некоторых случаях они заменяются равнозначной функциональностью.

Идентификатор	Основа?	Описание	Комментарий
B011		Встроенный язык Ada	
B013		Встроенный язык COBOL	
B014		Встроенный язык Fortran	
B015		Встроенный язык MUMPS	
B016		Встроенный язык Pascal	
B017		Встроенный язык PL/I	
B030		Расширенный динамический SQL	
B031		Базовый динамический SQL	
B032		Расширенный динамический SQL	
B033		Нетипизированные аргументы функции, вызываемой из SQL	
B034		Динамическое указание атрибутов курсора	
B035		Нерасширенные имена дескрипторов	
B036		Оператор describe input	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
B041		Расширения встроенных объявлений исключений SQL	
B051		Расширенные права для выполнения	
B111		Язык модулей — Ada	
B112		Язык модулей — C	
B113		Язык модулей — COBOL	
B114		Язык модулей — Fortran	
B115		Язык модулей — MUMPS	
B116		Язык модулей — Pascal	
B117		Язык модулей — PL/I	
B121		Язык подпрограмм — Ada	
B122		Язык подпрограмм — C	
B123		Язык подпрограмм — COBOL	
B124		Язык подпрограмм — Fortran	
B125		Язык подпрограмм — MUMPS	
B126		Язык подпрограмм — Pascal	
B127		Язык подпрограмм — PL/I	
B200		Полиморфные табличные функции (PTF)	
B201		Более одного абстрактного табличного параметра PTF	
B202		Соразделение PTF	
B203		Более одного указания соразделения	
B204		PRUNE WHEN EMPTY	
B205		Сквозная передача столбцов	
B206		Передача в PTF параметров с дескриптором	
B207		Векторное произведение разделов	
B208		Интерфейс компонентных процедур PTF	
B209		Расширенные имена PTF	
B211		Язык модулей — Ada: поддержка VARCHAR и NUMERIC	
B221		Язык подпрограмм — Ada: поддержка VARCHAR и NUMERIC	
F054		TIMESTAMP в списке приоритетов типа DATE	
F120		Оператор GET DIAGNOSTICS	
F121		Базовое управление диагностикой	
F122		Расширенное управление диагностикой	
F123		Вся диагностика	
F124		Оператор SET TRANSACTION: предложение DIAGNOSTICS SIZE	
F263		Разделённые запятыми предикаты в простом выражении CASE	
F291		Предикат UNIQUE	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
F301		CORRESPONDING в выражениях запросов	
F403		Секционированные соединённые таблицы	
F451		Определение набора символов	
F461		Именованные наборы символов	
F492		Необязательное указание соблюдения ограничения таблицы	
F521		Утверждения	
F671		Подзапросы в ограничениях CHECK	намеренно опущено
F673		Вызов подпрограммы, читающей SQL-данные, в условиях CHECK	
F693		Правила сортировки символов для SQL-сеансов и клиентских модулей	
F695		Поддержка перекодировки	
F696		Дополнительная документация по перекодировке	
F721		Откладываемые ограничения	только сторонние и уникальные ключи
F741		Типы ссылочных совпадений MATCH	пока без частичного совпадения
F812		Базовое флагирование	
F813		Расширенное флагирование	
F821		Ссылки на локальные таблицы	
F831		Полное изменение курсора	
F832		Изменяемые прокручиваемые курсоры	
F833		Изменяемые упорядоченные курсоры	
F841		Предикат LIKE_REGEX	возможная альтернатива — regexp_like()
F842		Функция OCCURRENCES_REGEX	возможная альтернатива — regexp_matches()
F843		Функция POSITION_REGEX	возможная альтернатива — regexp_instr()
F844		Функция SUBSTRING_REGEX	возможная альтернатива — regexp_substr()
F845		Функция TRANSLATE_REGEX	возможная альтернатива — regexp_replace()
F846		Поддержка октетов в операторах регулярных выражений	
F847		Неконстантные регулярные выражения	
F866		Предложение FETCH FIRST: параметр PERCENT	
R010		Распознавание шаблона строк: предложение FROM	
R020		Распознавание шаблона строк: предложение WINDOW	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
R030		Распознавание шаблона строк: полная поддержка агрегатов	
S011	Основа	Отдельные типы данных	
S011-01	Основа	Представление USER_DEFINED_TYPES	
S023		Базовые структурированные типы	
S024		Расширенные структурированные типы	
S025		Окончательные структурированные типы	
S026		Самоссылающиеся структурированные типы	
S027		Создание метода по заданному имени метода	
S028		Произвольный порядок параметров UDT	
S041		Базовые ссылочные типы	
S043		Расширенные ссылочные типы	
S051		Создание таблицы из типа	частично поддерживается
S081		Подтаблицы	
S091		Базовая поддержка массивов	частично поддерживается
S093		Массивы отдельных типов	
S094		Массивы ссылочных типов	
S097		Присваивание значения элементу массива	
S151		Предикат типа	см. pg_typeof()
S161		Приведение подтипов	
S162		Приведение подтипов для ссылочных типов	
S202		Вызываемые из SQL подпрограммы, работающие с мультимножествами	
S231		Указатели на структурные типы	
S232		Указатели на массивы	
S233		Указатели на мультимножества	
S241		Функции преобразований	
S242		Оператор изменения преобразования	
S251		Определяемые пользователем упорядочивания	
S261		Метод SPECIFICTYPE	
S271		Базовая поддержка мультимножеств	
S272		Мультимножества пользовательских типов	
S274		Мультимножества ссылочных типов	
S275		Расширенная поддержка мультимножеств	
S281		Типы вложенных коллекций	
S291		Ограничение уникальности для всей строки	
S401		Отдельные типы на базе типов массивов	
S402		Отдельные типы на базе типов мультимножеств	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
S403		ARRAY_MAX_CARDINALITY	
T011		Тип TIMESTAMP в информационной схеме	
T021		Типы данных BINARY и VARBINARY	
T022		Расширенная поддержка типов данных BINARY и VARBINARY	
T023		Составные двоичные строки	
T024		Пробелы в двоичных строках	
T039		Указатель на CLOB: неудерживаемый	
T040		Конкатенация типов данных CLOB	
T041		Базовая поддержка типа данных LOB	
T042		Расширенная поддержка типа данных LOB	
T043		Множитель T	
T044		Множитель P	
T045		Тип данных BLOB	
T046		Тип данных CLOB	
T047		Функции POSITION, OCTET_LENGTH, TRIM и SUBSTRING для типов данных BLOB	
T048		Конкатенация типов данных BLOB	
T049		Указатель на BLOB: неудерживаемый	
T050		Функции POSITION, CHAR_LENGTH, OCTET_LENGTH, LOWER, TRIM, UPPER и SUBSTRING для типов данных CLOB	
T051		Типы кортежей	
T053		Явные псевдонимы ссылки на все поля	
T062		Единицы длины символа	
T076		Тип данных DECFLOAT	
T101		Улучшенное определение возможности NULL	
T111		Изменяемые соединения, объединения и столбцы	
T175		Генерируемые столбцы	в основном поддерживаются
T176		Поддержка генераторов последовательностей	поддерживается за исключением NEXT VALUE FOR
T180		Системное версионирование таблиц	
T181		Таблицы с периодом времени прикладного уровня	
T200		Триггеры DDL	похожи, но не полностью совместимы
T211		Базовые возможности триггеров	
T218		Несколько триггеров для одного события вызываются в порядке создания	намеренно опущено
T231		Чувствительные курсоры	
T251		Оператор SET TRANSACTION: параметр LOCAL	
T262		Мультисерверные транзакции	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
T272		Улучшенное управление точками сохранения	
T301		Функциональные зависимости	частично поддерживается
T321	Основа	Базовые вызываемые из SQL подпрограммы	частично поддерживается
T322		Объявляемые атрибуты типа данных	
T324		Явное управление безопасностью подпрограмм SQL	
T326		Табличные функции	
T471		Наборы результатов в качестве возвращаемого значения	
T472		DESCRIBE CURSOR	
T495		Совместное изменение и извлечение данных	другой синтаксис
T502		Предикаты периодов	
T511		Счётчики транзакций	
T522		Значения по умолчанию для входных параметров процедур, вызываемых из SQL	поддерживаются, за исключением ключевого слова DEFAULT при вызове
T561		Удерживаемые указатели	
T571		Внешние вызываемые из SQL функции, возвращающие массивы	
T572		Внешние вызываемые из SQL функции, возвращающие мультимножества	
T601		Ссылки на локальные курсоры	
T616		Варианты обработки NULL для функций LEAD и LAG	
T618		Функция NTH_VALUE	функция существует, но некоторые возможности отсутствуют
T619		Вложенные оконные функции	
T625		LISTAGG	
T641		Присваивание нескольким столбцам	поддерживаются только некоторые варианты синтаксиса
T652		Операторы динамического SQL в SQL-подпрограммах	
T654		Операторы динамического SQL во внешних подпрограммах	
T801		Тип данных JSON	
T802		Расширенный тип данных JSON	
T839		Преобразование даты/времени из символьных строк с форматированием и обратно	
T860		Упрощённое обращение к SQL/JSON: только ссылки на столбцы	
T861		Упрощённое обращение к SQL/JSON: обращение к элементу JSON с учётом регистра	

Идентификатор	Основа?	Описание	Комментарий
T862		Упрощённое обращение к SQL/JSON: оператор обращения по звёздочке	
T863		Упрощённое обращение к SQL/JSON: строковый литерал в одинарных кавычках в качестве оператора обращения	
T864		Упрощённое обращение к SQL/JSON	
T881		JSON в операциях упорядочивания	с jsonb, частично поддерживается
T882		JSON в операциях группировки элементов мультимножеств	
M001		Связи данных (DATALINK)	
M002		Связи данных через SQL/CLI	
M003		Связи данных через встроенный SQL	
M004		Поддержка сторонних данных	частично поддерживается
M005		Поддержка сторонних схем	
M006		Подпрограмма GetSQLString	
M007		TransmitRequest	
M009		Подпрограммы GetOpts и GetStatistics	
M010		Поддержка обёрток сторонних данных	другой API
M011		Связи данных через Ada	
M012		Связи данных через C	
M013		Связи данных через COBOL	
M014		Связи данных через Fortran	
M015		Связи данных через MUMPS	
M016		Связи данных через Pascal	
M017		Связи данных через PL/I	
M018		Подпрограммы интерфейса обёртки сторонних данных на языке Ada	
M019		Подпрограммы интерфейса обёртки сторонних данных на языке C	другой API
M020		Подпрограммы интерфейса обёртки сторонних данных на языке COBOL	
M021		Подпрограммы интерфейса обёртки сторонних данных на языке Fortran	
M022		Подпрограммы интерфейса обёртки сторонних данных на языке MUMPS	
M023		Подпрограммы интерфейса обёртки сторонних данных на языке Pascal	
M024		Подпрограммы интерфейса обёртки сторонних данных на языке PL/I	
M030		Поддержка сторонних данных SQL-сервера	
M031		Общие подпрограммы обёртки сторонних данных	
X012		Мультимножества типа XML	

Идентификатор	**Основа?**	**Описание**	**Комментарий**
X013		Отдельные типы, производные от XML	
X015		Поля типа XML	
X025		XMLCast	
X030		XMLDocument	
X065		XMLParse: ввод двоичных строк и вариант CONTENT	
X066		XMLParse: ввод двоичных строк и вариант DOCUMENT	
X068		XMLSerialize: BOM	
X073		XMLSerialize: сериализация двоичных строк и вариант CONTENT	
X074		XMLSerialize: сериализация двоичных строк и вариант DOCUMENT	
X075		XMLSerialize: сериализация двоичных строк	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: явное указание ENCODING	
X078		XMLSerialize: явное объявление XML	
X080		Пространства имён при публикации XML	
X081		Объявления пространств имён XML на уровне запроса	
X082		Объявления пространств имён XML в DML	
X083		Объявления пространств имён XML в DDL	
X084		Объявления пространств имён XML в составных операторах	
X085		Предопределённые префиксы пространств имён	
X086		Объявления пространств имён XML в XMLTable	
X091		Предикат содержимого XML	
X096		XMLExists	только XPath 1.0
X100		Поддержка ведущего языка для XML: вариант CONTENT	
X101		Поддержка ведущего языка для XML: вариант DOCUMENT	
X110		Поддержка ведущего языка для XML: отображение VARCHAR	
X111		Поддержка ведущего языка для XML: отображение CLOB	
X112		Поддержка ведущего языка для XML: отображение BLOB	
X113		Поддержка ведущего языка для XML: указание STRIP WHITESPACE	
X114		Поддержка ведущего языка для XML: указание PRESERVE WHITESPACE	
X131		Предложение XMLBINARY на уровне запроса	
X132		Предложение XMLBINARY в DML	

Идентификатор	Основа?	Описание	Комментарий
X133		Предложение XMLBINARY в DDL	
X134		Предложение XMLBINARY в составных операторах	
X135		Предложение XMLBINARY в подзапросах	
X141		Предикат IS VALID: в зависимости от данных	
X142		Предикат IS VALID: предложение ACCORDING TO	
X143		Предикат IS VALID: предложение ELEMENT	
X144		Предикат IS VALID: расположение схемы	
X145		Предикат IS VALID вне ограничений-проверок	
X151		Предикат IS VALID с вариантом DOCUMENT	
X152		Предикат IS VALID с вариантом CONTENT	
X153		Предикат IS VALID с вариантом SEQUENCE	
X155		Предикат IS VALID: NAMESPACE без предложения ELEMENT	
X157		Предикат IS VALID: NO NAMESPACE с предложением ELEMENT	
X160		Базовая информационная схема для зарегистрированных XML-схем	
X161		Расширенная информационная схема для зарегистрированных XML-схем	
X170		Варианты обработки NULL с XML	
X171		Вариант NIL ON NO CONTENT	
X181		Тип XML(DOCUMENT(UNTYPED))	
X182		Тип XML(DOCUMENT(ANY))	
X190		Тип XML(SEQUENCE)	
X191		Тип XML(DOCUMENT(XMLSCHEMA))	
X192		Тип XML(CONTENT(XMLSCHEMA))	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: передача контекста	
X204		XMLQuery: инициализация переменной XQuery	
X205		XMLQuery: указание EMPTY ON EMPTY	
X206		XMLQuery: указание NULL ON EMPTY	
X211		Поддержка XML 1.1	
X222		Механизм передачи XML BY REF	BY REF принимается, но игнорируется; фактически всегда действует BY VALUE
X231		Тип XML(CONTENT(UNTYPED))	
X232		Тип XML(CONTENT(ANY))	
X241		RETURNING CONTENT при публикации XML	
X242		RETURNING SEQUENCE при публикации XML	

Идентификатор	Основа?	Описание	Комментарий
X251		Хранимые значения XML типа XML(DOCUMENT(UNTYPED))	
X252		Хранимые значения XML типа XML(DOCUMENT(ANY))	
X253		Хранимые значения XML типа XML(CONTENT(UNTYPED))	
X254		Хранимые значения XML типа XML(CONTENT(ANY))	
X255		Хранимые значения XML типа XML(SEQUENCE)	
X256		Хранимые значения XML типа XML(DOCUMENT(XMLSCHEMA))	
X257		Хранимые значения XML типа XML(CONTENT(XMLSCHEMA))	
X260		Тип XML: предложение ELEMENT	
X261		Тип XML: NAMESPACE без предложения ELEMENT	
X263		Тип XML: NO NAMESPACE с предложением ELEMENT	
X264		Тип XML: расположение схемы	
X271		XMLValidate: в зависимости от данных	
X272		XMLValidate: предложение ACCORDING TO	
X273		XMLValidate: предложение ELEMENT	
X274		XMLValidate: расположение схемы	
X281		XMLValidate с вариантом DOCUMENT	
X282		XMLValidate с вариантом CONTENT	
X283		XMLValidate с вариантом SEQUENCE	
X284		XMLValidate: NAMESPACE без предложения ELEMENT	
X286		XMLValidate: NO NAMESPACE с предложением ELEMENT	
X300		XMLTable	только XPath 1.0
X305		XMLTable: инициализация переменной XQuery	

D.3. Ограничения XML и совместимость с SQL/XML

В SQL:2006 были внесены значительные изменения в посвящённой XML части ISO/IEC 9075-14 (SQL/XML). Реализация типа данных XML и связанных функций в Postgres Pro в большей степени соответствует более ранней редакции, SQL:2003, с некоторыми заимствованиями из последующих редакций. В частности:

- Тогда как в текущем стандарте существует семейство типов данных XML, содержащих «документы» или «содержимое» в нетипизированном виде или с типами XML Schema, а также тип `XML(SEQUENCE)`, содержащий произвольные части XML-документа, в Postgres Pro есть только один тип `xml`, который может содержать «документ» или «содержимое». Определённый в стандарте тип «последовательность» в Postgres Pro отсутствует.

- Postgres Pro предоставляет две функции, появившиеся в SQL:2006, но вместо языка XML Query, как должно быть согласно стандарту, в них используется язык XPath 1.0.

- Postgres Pro не поддерживает предложения `RETURNING CONTENT` и `RETURNING SEQUENCE`: реализация функций, определяемых с этими предложениями в спецификации, подразумевает возврат содержимого.

В этом разделе описаны некоторые из образовавшихся в итоге различий, с которыми вы можете столкнуться.

D.3.1. Запросы ограничиваются XPath версии 1.0

Специфичные для Postgres Pro функции `xpath()` и `xpath_exists()` выполняют запросы к XML-документам на языке XPath. В Postgres Pro также имеются поддерживающие только XPath стандартные функции `XMLEXISTS` и `XMLTABLE`, хотя согласно стандарту они должны поддерживать XQuery. Все эти функции в Postgres Pro реализованы с использованием библиотеки libxml2, которая поддерживает только XPath 1.0.

Существует тесная связь между языком XQuery и XPath версии 2.0 и новее: любое выражение, синтаксически правильное и выполняющееся успешно, выдаёт в обоих языках одинаковые результаты (за незначительным исключением, связанным с числовым обозначением символов или использованием предопределённых сущностей — XQuery заменяет их соответствующими символами, а XPath оставляет в исходном виде). Но между XPath 1.0 и этими языками подобная связь отсутствует: он появился гораздо раньше и во многом отличается от них.

Заслуживают отдельного рассмотрения две категории ограничений: ограничение языка XQuery до XPath для функций, описанных в стандарте SQL, и ограничение XPath до версии 1.0 как для стандартизированных функций, так и для специфичных функций Postgres Pro.

D.3.1.1. Ограничение языка XQuery до XPath

В число отличий XQuery от XPath входят:

- Выражения XQuery могут выдавать не только всевозможные значения XPath, но и конструировать новые XML-узлы. XPath может создавать и возвращать значения атомарных типов (числа, строки и так далее), но выдаваемые им XML-узлы должны уже присутствовать в документе, поступившем на вход выражения.
- В XQuery есть управляющие конструкции для организации циклов, сортировки и группировки.
- В XQuery поддерживается объявление и использование локальных функций.

В последних версиях XPath начинают появляться возможности, пересекающиеся с имеющимися в XQuery (например, конструкции `for-each`, `sort`, анонимные функции и функция `parse-xml`, создающая узел из строки), но до XPath 3.0 их не было.

D.3.1.2. Ограничения XPath до версии 1.0

Разработчикам, знакомым с XQuery и XPath 2.0 или новее, приходится иметь дело с рядом недостатков XPath версии 1.0:

- Фундаментальный тип результатов XQuery/XPath, тип `sequence`, который может содержать XML-узлы, атомарные значения, и всё это вместе, в XPath 1.0 отсутствует. В 1.0 выражения могут выдавать только набор узлов (состоящих из нуля или нескольких узлов XML) или единственное атомарное значение.
- В отличие от последовательностей XQuery/XPath, которые могут содержать произвольные элементы в любом требующемся порядке, во множестве узлов XPath 1.0 нет гарантированного порядка, и оно, как и любое другое множество, не может содержать несколько вхождений одного элемента.

Примечание

Библиотека libxml2 не всегда возвращает в Postgres Pro наборы узлов с внутренними членами в том порядке, в котором они идут во входном документе. В её документации не гарантируется корректное поведение, а выражение XPath 1.0 не может на это воздействовать.

- Тогда как XQuery/XPath поддерживают все типы, определённые в стандарте XML Schema, а также множество операторов и функций, работающих с этими типами, XPath 1.0 поддерживает только множества узлов и три атомарных типа: `boolean`, `double` и `string`.
- В XPath 1.0 отсутствует условный оператор. Выражение XQuery/XPath вида `if ( hat ) then hat/@size else "no hat"` не имеет эквивалента в XPath 1.0.
- В XPath 1.0 нет оператора сравнения строк с упорядочиванием. Условия `"cat" < "dog"` и `"cat" > "dog"` оба являются ложными, так как они выполняются как числовые сравнения двух значений `NaN`. Условия же `=` и `!=`, напротив, сравнивают строки в виде строк.
- XPath 1.0 размывает разницу между *сравнением значений* и *общими сравнениями*, которая имеется в XQuery/XPath. Сравнения `sale/@hatsize = 7` и `sale/@customer = "alice"` по сути являются количественными сравнениями, и результатом их будет истина, если существует элемент `sale` с заданным значением атрибута, тогда как `sale/@taxable = false()` — сравнение всего набора узлов с *фактическим логическим значением*. Его результат будет истиной, только если у элемента `sale` вовсе не будет атрибута `taxable`.
- В модели данных XQuery/XPath *узел документа* может иметь либо форму документа (то есть содержать в точности один элемент верхнего уровня, снаружи которого допускаются только комментарии и инструкции обработки), либо форму содержимого (с ослабленными ограничениями). В XPath 1.0 ему соответствует *корневой узел*, который может иметь только форму документа. Этим отчасти объясняется то, что значение типа `xml`, передаваемое в качестве элемента контекста любым функциям Postgres Pro на базе XPath, должно быть в форме документа.

Кроме отмеченных выше имеются и другие различия. В языках XQuery и XPath версии 2.0 и новее существует режим совместимости с XPath 1.0, а в документации W3C имеется перечень *изменений функций* и *изменений в языке* применительно к этому режиму. Этот перечень гораздо более полный, но тоже не исчерпывающий. Даже режим совместимости этих языков не обеспечивает их полную идентичность XPath 1.0.

D.3.1.3. Преобразование значений/типов данных между SQL и XML

В SQL:2006 и более поздних ревизиях чётко определены преобразования между стандартными типами SQL и типами стандарта XML Schema в обе стороны. Однако эти правила выражаются в типах и понятиях, определённых в XQuery/XPath, и не могут быть непосредственно применены к другой модели данных, присущей XPath 1.0.

Когда Postgres Pro сопоставляет значения данных SQL с XML (как в функции `xmlelement`), или XML с SQL (как в выходных столбцах `xmltable`), за исключением нескольких отдельно обрабатываемых случаев, Postgres Pro просто полагает, что строка XPath 1.0, содержащая данные типа XML, будет допустимой для ввода в текстовом виде в тип данных SQL, и наоборот. Это правило добродетельно своей простотой, и при этом преобразования для многих типов данных в итоге оказываются такими, какими и должны быть согласно стандарту.

Там же, где это нужно для взаимодействия с другими системами, для некоторых типов данных можно явно использовать функции форматирования типов данных (например, описанные в Разделе 9.8) для получения преобразований, в точности соответствующих стандарту.

D.3.2. Непреднамеренные ограничения реализации

В этом разделе описываются дополнительные ограничения, присущие текущей реализации в Postgres Pro, но не самой библиотеке libxml2.

D.3.2.1. Передача параметров только по значению (`BY VALUE`)

В стандарте SQL определены два *механизма передачи параметров*, осуществляющих передачу XML-аргумента из SQL в XML-функцию или получение результата: `BY REF`, в котором конкретное значение в XML остаётся привязанным к своему узлу, и `BY VALUE`, в котором передаётся содержимое XML, но связь с узлом теряется. Выбрать механизм можно перед списком параметров, в качестве механизма по умолчанию для всех параметров, или после каждого отдельного параметра, переопределив тем самым выбор по умолчанию.

В качестве иллюстрации различия взгляните на следующие два запроса, которые в окружении SQL:2006 выдают true и false, если *x* является XML-значением:

```
SELECT XMLQUERY('$a is $b' PASSING BY REF x AS a, x AS b NULL ON EMPTY);
SELECT XMLQUERY('$a is $b' PASSING BY VALUE x AS a, x AS b NULL ON EMPTY);
```

Postgres Pro принимает указания `BY VALUE` и `BY REF` в конструкции `XMLEXISTS` или `XMLTABLE`, но игнорирует их. Тип `xml` содержит сериализованное представление данных в текстовом виде, поэтому сущность узла, которую нужно сохранять, отсутствует, и передача фактически производится по значению (`BY VALUE`).

D.3.2.2. Отсутствие именованных параметров запросов

Функции на базе XPath могут принимать один параметр, служащий контекстным элементом для выражения XPath, но не поддерживают передачу дополнительных значений, которые могли бы использоваться в выражении как именованные параметры.

D.3.2.3. Отсутствие типа `XML(SEQUENCE)`

Тип данных `xml` в Postgres Pro может содержать значение только в форме документа (`DOCUMENT`) или содержимого (`CONTENT`). Контекстный элемент выражения XQuery/XPath должен быть одиночным XML-узлом или атомарным значением, но в XPath 1.0 это может быть только XML-узел, и при этом нет типа узла, содержащего `CONTENT`. Как следствие, в Postgres Pro в качестве контекстного элемента XPath можно передать данные XML в единственном виде — в виде правильно оформленного документа (`DOCUMENT`).

Приложение E. Замечания к выпускам

В замечаниях к выпускам отмечаются значительные изменения, имевшие место в каждом выпуске Postgres Pro Standard, при этом ключевые изменения и вопросы миграции освещаются в самом начале. В эти замечания не включаются изменения, затрагивающие лишь нескольких пользователей, как и изменения внутренние и поэтому пользователям не заметные. Например, оптимизатор усовершенствуется почти в каждом выпуске, но для пользователей эти улучшения проявляются обычно просто в ускорении запросов.

E.1. Postgres Pro Standard 17.6.1

Дата выпуска: 2025-09-18

E.1.1. Обзор

Этот выпуск основан на PostgreSQL 17.6 и Postgres Pro Standard 17.5.1. Все усовершенствования, унаследованные от PostgreSQL 17.6, перечислены в Замечаниях к выпуску PostgreSQL 17.6. Другие важные изменения и улучшения перечислены ниже:

- Улучшена производительность за счёт замены циклических блокировок, ранее используемых для общей очереди сообщений аннулирования, более эффективным примитивом синхронизации. Это позволило устранить узкое место в производительности. Может быть полезно для 1С.
- Обновлены CREATE DATABASE, createdb и initdb. Теперь если значение lc_collate используется как значение по умолчанию для *`icu_locale`* или `--icu-locale`, выводится информационное сообщение.
- Оптимизировано поведение при работе с функцией `COALESCE`. Теперь функция заменяется одним аргументом, если остальные аргументы оказываются `null`. Это позволяет улучшить оценку избирательности для условий запросов и создавать более эффективные планы выполнения запросов.
- Добавлены следующие изменения в реализацию crash_info:
  - Добавлены параметры конфигурации crash_info_timer, crash_info_timer_interval и crash_info_query_threshold, которые позволяют профилировать длительные запросы в выходных файлах `crash_info`.
  - Расширена диагностическая информация, которая будет записываться в соответствии с настройкой параметра конфигурации crash_info_dump в случае сбоя обслуживающего процесса. Теперь по умолчанию выгружается общая информация о сигнале, а для Linux x64 — ещё и информация о состоянии процессора и байтах около RIP (указатель возврата инструкции). Это обеспечивает лучшую диагностику, особенно с сигналами `SIGILL` (недопустимые инструкции). Для Linux x64 также изменён способ выгрузки файлов стека при сбоях. Кроме того, чтобы избежать ошибок, которые могут возникнуть при записи диагностической информации, из списка источников, выгружаемых по умолчанию, исключён `memory_context`.
- Устранена проблема при обработке очереди `LISTEN`/`NOTIFY`, из-за которой нельзя было узнать статус транзакции. Ранее процедура `VACUUM FREEZE` могла увеличить значение `datfrozenxid` слишком сильно.
- Устранена проблема, из-за которой планировщик мог выбирать неоптимальные планы выполнения запросов с оператором `LIKE` для типа данных `mchar`.
- Добавлена поддержка Debian 13, ОС Red Hat Enterprise Linux 10, Rocky Linux 10, Oracle Linux 10 и AlmaLinux 10.
- Прекращена поддержка системы Ubuntu 20.04.

- Добавлено расширение wal2json для логического декодирования, которое преобразует изменения базы данных из журнала предзаписи (WAL) в формат JSON.
- Обновлён модуль aqo. Включены следующие исправления:
  - В представление aqo_data добавлено поле `tmpoids`, а в функцию `aqo_data_update` — параметр *`tmpoids`* для поддержки запросов, которые используют временные таблицы вместе с постоянными таблицами.
  - Устранено условие гонки при вызове функции `aqo_cleanup` из разных сеансов.
  - Устранена проблема, из-за которой данные обучения aqo не сохранялись для материализуемых запросов.
  - Устранена проблема с обучением aqo на узлах, работа на которых прекращается досрочно. Теперь aqo обучается на таких узлах с минимальным значением `reliability`, только если фактическое число строк превышает ожидаемое. В остальных случаях механизм обучения отключён.
- В выводе auto_explain время планирования запроса вынесено в отдельную строку под деревом планировщика, как это сделано в выводе `EXPLAIN`. До этого pgbadger не мог распознать информацию о запросе и поэтому не включал её в отчёт.
- Обновлено расширение pg_proaudit. Включены следующие усовершенствования:
  - Добавлена поддержка формата CEF для файлов журналов событий.
  - Скорректирован вывод pg_proaudit. Теперь поля `xid` и `vxid` не отображаются, если имеют нулевое значение.
- Приложение pg_probackup обновлено до версии 2.8.10, в которой добавлена новая функциональность, улучшена производительность и исправлены некоторые ошибки:
  - Для команды `add-instance` добавлен параметр `--wal-tree`, который позволяет хранить заархивированные файлы WAL в древовидной структуре. Это повышает производительность операций при работе с журналами большого объёма.
  - В вывод команды `show` добавлено поле `postgres-edition` в формате `JSON` для отображения информации о редакции СУБД.
  - Улучшено действие параметра `--dry-run` при восстановлении табличных пространств. Оценка операции и вывод плана стали более точными.
  - Устранена проблема, связанная с вычислением окна хранения. Значение времени теперь берётся с компьютера, на котором работает pg_probackup. Окно хранения вычисляется корректно, даже если серверы расположены в разных часовых поясах.
  - Устранена проблема, связанная с работой команды `archive-get`. Теперь каталог `pbk_prefetch` создаётся с правильными правами доступа, которые наследуются из `PGDATA`.
  - Устранён ложный сбой команды `checkdb`, возникавший в удалённом режиме при параллельных операциях `TRUNCATE` или удалении таблиц на резервных узлах.
- Устранены следующие проблемы с модулем pgpro_bindump:
  - Устранён риск сбоя *pg_probackup3*, который возникал, если в ходе проверки резервных копий с использованием модуля pgpro_bindump в режиме PRO встречались временные таблицы.
  - Устранена утечка памяти в *pg_probackup3* при использовании модуля pgpro_bindump для операций PTRACK.
  - Устранена проблема, которая могла вызывать зацикливание в *pg_probackup3* при использовании модуля pgpro_bindump.
  - Устранена проблема с резервным копированием в нескольких линиях времени в режиме PTRACK в *pg_probackup3* при использовании модуля pgpro_bindump в режиме PRO.

- Модуль pgpro_pwr обновлён до версии 4.10 с новым дизайном отчётов и поддержкой PostgreSQL 18.
- Обновлено расширение pgpro_stats. Включены следующие усовершенствования и исправления:
  - Реализована возможность отслеживать команды утилиты по `query_id`, как в pg_stat_statements. Ранее команды утилиты отслеживались путём строгого сравнения по текстовым строкам запросов и поэтому не могли быть нормализованы корректно.
  - Исправлена ошибка сегментации, которая могла возникать при определении дополнительных метрик. Ошибка была вызвана проблемой при выделении памяти.
- Устранена проблема, из-за которой при использовании пресета `1c.tune` утилиты pgpro_tune задавались неверные значения параметров конфигурации.
- Модуль pg_query_state обновлён до версии 1.2, в которую добавлены функции pg_progress_bar и pg_progress_bar_visual, позволяющие отслеживать текущий прогресс выполнения запросов.
- Обновлён модуль rum. Включены следующие исправления:
  - Исправлена ошибка сегментации, вызванная некорректным обращением к памяти, когда указатель на следующую страницу индекса использовался после того, как предыдущая страница была изменена другим потоком.
  - Устранён бесконечный цикл сканирования индекса при использовании нескольких предложений `ORDER BY`.

E.1.2. Миграция на версию 17.6.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

> **Важно**
>
> Если вы использовали предыдущую версию aqo и её потребовалось обновить, версия aqo на резервном сервере может отличаться от версии модуля на ведущем.
>
> Чтобы избежать связанных с этим проблем, отключите aqo, задав для параметров `aqo.enable` и `aqo.wal_rw` значение `off` сначала на ведущем сервере, а потом на резервных серверах. После успешного обновления включите aqo, задав для параметров `aqo.wal_rw` и `aqo.enable` значение `on` сначала на резервных серверах, а потом на ведущем сервере.

Версия ABI может измениться в корректирующем выпуске Postgres Pro. В таком случае при попытке запустить расширение возникает ошибка `ABI mismatch` (Несоответствие ABI), поэтому обязательно установите новую версию расширения, поставляемую с новым выпуском Postgres Pro, или перекомпилируйте стороннее расширение, чтобы использовать его с текущей версией Postgres Pro.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 17.0.1.

E.2. Postgres Pro Standard 17.5.1

Дата выпуска: 2025-05-30

E.2.1. Обзор

Этот выпуск основан на PostgreSQL 17.5 и Postgres Pro Standard 17.4.1. Все усовершенствования, унаследованные от PostgreSQL 17.5, перечислены в Замечаниях к выпуску PostgreSQL 17.5. Другие важные изменения и улучшения перечислены ниже:

- Добавлен параметр конфигурации enable_alternative_sorting_cost_model, который позволяет включать или отключать использование планировщиком альтернативной модели подсчёта стоимости сортировки кортежей.
- Добавлен параметр конфигурации enable_any_to_lateral_transformation, который позволяет включать или отключать преобразование подзапросов `ANY` в соединения с ключевым словом `LATERAL`.
- Добавлена экспериментальная функциональность, которая включает механизм параллельного выполнения запросов для запросов, использующих хотя бы одну временную таблицу. Она может быть включена при помощи нового параметра конфигурации enable_parallel_temptables. Эту функциональность пока не следует использовать в производственной среде. Также добавлен параметр конфигурации write_page_cost, который позволяет оценивать приблизительную стоимость сброса страниц временных таблиц на диск и работает только при включённом параметре `enable_parallel_temptables`. Эти параметры полезны для 1С.
- Добавлены следующие изменения в реализацию crash_info:
  - Добавлена обработка сигналов `SIGILL` в процесс обработки `crash_info`.
  - В выходные файлы `crash_info` добавлены новые сведения: время запуска процесса и текст запроса при планировании.
  - Исправлены некорректные имена функций в первых 2-3 строках трассировки стека `crash_info`.
  - Исправлено возможное усечение данных в конце файлов выгрузки SQL-запросов. Ранее переполнение буфера при выгрузке SQL-запроса могло приводить к тому, что записывались не все данные, в результате чего в конце файлов, создаваемых `crash_info`, могли содержаться усечённые данные.
  - Исправлена обработка сигналов о сбоях (отправленных командой kill) обслуживающими процессами. Ранее с первым сигналом информация `crash_info` отправлялась в журнал, но процесс продолжал работу без создания дампа памяти, даже если он был настроен. И только со вторым сигналом обслуживающий процесс завершался и создавался дамп памяти так, как ожидалось.
- Исправлена проблема с недостающей статистикой об очистке при использовании нескольких рабочих процессов очистки индексов.
- Добавлена поддержка МосОС 15.
- Добавлена поддержка ОС Ubuntu 25.04.
- Добавлено расширение daterange_inclusive, которое позволяет добавлять в вывод верхнюю границу временного диапазона.
- Добавлено решение *pg_probackup3* для резервного копирования и восстановления кластеров баз данных Postgres Pro. За подробной информацией обратитесь к *Замечаниям к выпускам* pg_probackup3.
- Добавлен модуль pgpro_bindump для управления операциями резервного копирования и восстановления. В модуле реализованы специализированные команды репликации для расширенного протокола репликации и собственный формат архивирования. Модуль не требует подключения по SSH. Новый модуль разработан специально для использования вместе с утилитой *pg_probackup3*.
- Добавлена утилита pgpro_tune, которая позволяет устанавливать параметры конфигурации Postgres Pro автоматически.
- В initdb добавлена новая переменная окружения `PGPRO_TUNE`, которая указывает, нужно ли использовать утилиту pgpro_tune без изменения параметров в командной строке.
- Удалён параметр `--tune` программы `pg-setup`. Вместо этого используйте новую утилиту pgpro_tune.
- Модуль aqo обновлён до версии 3.1, в которую включены следующие исправления и усовершенствования:

  - Добавлена возможность агрегирования базы знаний для запросов, отличающихся только именами таблиц. Теперь в качестве идентификатора пространства признаков используется так называемое *основное хеш-значение*, которое является общим для таких запросов.
  - Исправлена проблема, из-за которой на резервном сервере мог возникать сбой с сигналом SIGABRT, если версия aqo на таком сервере отличалась от версии модуля на ведущем.
- Обновлён модуль oracle_fdw. Включены следующие исправления и усовершенствования:
  - Увеличена длина строк вывода EXPLAIN для соответствия условиям фильтров некоторых запросов к каталогам Oracle. Новое значение — 3000 символов.
  - Устранён сбой, который мог возникать, если для несуществующих столбцов были определены параметры. Если у таблицы Oracle было меньше столбцов, чем у таблицы Postgres Pro, и для одного из таких дополнительных столбцов был определён параметр, oracle_fdw при записи данных выходил за границы массива, что приводило к повреждению памяти и сбоям. Теперь параметры для таких столбцов игнорируются.
- Обновлено расширение pg_proaudit. Включены следующие исправления и усовершенствования:
  - Добавлены новые типы объектов: `CATALOG RELATION` и `CATALOG FUNCTION`.
  - Добавлены новые поля событий: UUID, XID и VXID. Теперь стало возможно определить событие по его UUID и идентификатору транзакции (если применимо).
  - Исправлено поведение параметра конфигурации `pg_proaudit.log_catalog_access`, теперь оно отражает новую логику регистрации событий для объектов системного каталога.
  - Исправлена логика обработки событий отключения от баз данных. Теперь такие события связаны с соответствующими событиями аутентификации, что позволяет регистрировать события отключения, даже когда соответствующее правило регистрации удаляется после аутентификации, но до непосредственного отключения.
  - Исправлена проблема, из-за которой тип события `DISCONNECT` не регистрировался для пользователя, который являлся членом роли, заданной в правиле регистрации события.
  - Исправлена ошибка, из-за которой запись журнала делалась не в правильном файле, если была настроена ротация файлов журнала.
  - Исправлена проблема, из-за которой pg_proaudit не регистрировал события создания схем.
  - Исправлено поведение процесса `logger` при удалении роли из параллельного сеанса, настроенного в правиле регистрации событий.
- Приложение pg_probackup обновлено до версии 2.8.9, в которой добавлена новая функциональность, улучшена производительность и исправлены некоторые ошибки:
  - Добавлена команда `maintain`, предназначенная для устранения проблем, которые могут возникать при принудительном завершении резервного копирования.
  - Добавлен параметр `--lock-lifetime`, который позволяет задавать тайм-аут для блокировок и может быть полезен для вычислительных сред с низкой скоростью соединения.
  - Стабилизировано сохранение первоначальных прав доступа к каталогам при запуске команды `init`.
  - Стабилизирована работа команды `checkdb` на удалённом сервере.
  - Повышена стабильность восстановления с валидацией на момент времени (PITR).
  - Исправлено некорректное поведение, которое могло наблюдаться при запуске ожидания потока передачи WAL в режиме доставки ARCHIVE.
- Модуль pgpro_pwr обновлён до версии 4.9, в которой в основном улучшена производительность и исправлены некоторые ошибки. Основные изменения перечислены ниже:

  - Добавлена поддержка pgpro_stats 1.9.
  - Добавлена возможность определять режим сбора информации о размере отношений глобально при помощи параметра конфигурации `pgpro_pwr.relsize_collect_mode` расширения или для сервера при помощи функции `set_server_size_sampling`.
  - Добавлена возможность тонкой настройки сбора статистики сервера, которая включается вызовом функции `set_server_setting`. Она позволяет определять статистику для сбора.
  - Добавлена возможность предварительного просмотра параметров хранения таблиц и индексов в разделе отчёта «Schema object statistics» (Статистика по объектам схемы).
- Расширение pgpro_stats обновлено до версии 1.9, в которую включены следующие усовершенствования и исправления:
  - Улучшена трассировка сеансов, которая теперь предоставляет больше информации. В частности, новый атрибут фильтра `time_info` управляет включением дополнительной информации в вывод трассировки сеансов, а новый параметр конфигурации `pgpro_stats.trace_query_text_size` позволяет ограничивать размер запроса в таком выводе.
  - Имена атрибутов фильтра `explain_*` трассировщика сеансов приведены в соответствие с именами параметров конфигурации трассировки сеансов.
  - Изменён формат файла выгрузки статистики и соответствующие процедуры сохранения/загрузки.
  - Реализовано отключение функциональности трассировки сеансов, если не указаны соответствующие фильтры.
  - Запрещено одновременно указывать pgpro_stats и pg_stat_statements в `shared_preload_libraries`. Если они указаны вместе, сервер баз данных не запустится.
- Обновлено расширение pg_wait_sampling. Включены следующие исправления:
  - Исправлена проблема, из-за которой переменные GUC могли перезаписываться при использовании параллельных рабочих процессов.
  - Исправлена проблема с некорректными показаниями, которая могла возникать из-за условий гонки при выключенном параметре конфигурации `pg_wait_sampling.sample_cpu`.

E.2.2. Миграция на версию 17.5.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Версия ABI может измениться в корректирующем выпуске Postgres Pro. В таком случае при попытке запустить расширение возникает ошибка `ABI mismatch` (Несоответствие ABI), поэтому обязательно установите новую версию расширения, поставляемую с новым выпуском Postgres Pro, или перекомпилируйте стороннее расширение, чтобы использовать его с текущей версией Postgres Pro.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 17.0.1.

E.3. Postgres Pro Standard 17.4.1

Дата выпуска: 2025-03-03

E.3.1. Обзор

Этот выпуск основан на PostgreSQL 17.4 и Postgres Pro Standard 17.2.1. Все усовершенствования, унаследованные от PostgreSQL 17.4, перечислены в Замечаниях к выпуску PostgreSQL 17.4. Другие важные изменения и улучшения перечислены ниже:

- В выходные файлы `crash_info` добавлены новые сведения, например текущий рабочий каталог, каталог `PGDATA` и т. д.
- Добавлены повышенные требования безопасности для команды CREATE PROFILE. Теперь при использовании параметра `PASSWORD_REQUIRE_COMPLEX` пароли должны содержать символы из трёх разных групп. Также все буквы или их аналоги в языках без разделения на регистры (хинди, китайский и т.п.) в кодировке UTF-8 считаются строчными буквами.
- Устранена проблема, из-за которой при включённом параметре `crash_info` могли не записываться диагностические файлы во время обработки переполнения стека. Ранее обработчик сигналов запускался на стеке с переполнением, что могло помешать его запуску из-за нехватки памяти. Теперь обработчик использует альтернативный стек.
- Устранена проблема при пользовательском сканировании, которая могла приводить к некорректному состоянию плана запроса и возникновению ошибки при использовании pgpro_stats.
- Устранена проблема, из-за которой роли без явно заданного атрибута `VALID UNTIL` и связанные с профилем, у которого для PASSWORD_GRACE_TIME установлено значение 0, а для `PASSWORD_LIFE_TIME` — `UNLIMITED`, некорректно получали предупреждения об истечении срока действия пароля. Теперь пароли корректно считаются действительными бесконечно.
- Обновлён модуль aqo. Реализован режим `auto`, который работает как режим `learn`, но предотвращает переполнение памяти, реализуя механизм вытеснения из кеша давно неиспользуемых (LRU, least recently used) данных.
- Оптимизирован модуль fasttrun для пропуска ненужных операций усечения пустых временных отношений.
- Приложение mamonsu обновлено до версии 3.5.11, в которую включена поддержка pgpro_stats версии 1.8.
- Приложение pgbadger обновлено до версии 13.0.
- pgbouncer обновлён до версии 1.24.0.
- Расширение pg_portal_modify обновлено до версии 0.3.5.
- Обновлено расширение pg_proaudit. Включены следующие исправления и усовершенствования:
  - Добавлено новое поле событий: имя пользователя сеанса. Теперь события содержат информацию об атрибутах `session_user` и `current_user` SQL-сеанса.
  - Добавлена поддержка новых классов событий: `ALL_DDL_NONTEMP` и `ALL_DML_NONTEMP`. Область действия этих классов ограничена объектами, не находящимися во временных схемах `pg_temp_`*`nnn`*.
  - Исправлено внесение в журнал событий `SELECT FOR UPDATE` и `SELECT FOR KEY SHARE`: теперь они корректно указываются как команды `SELECT`, а не `UPDATE`.
- Приложение pg_probackup обновлено до версии 2.8.7, в которой улучшена производительность и исправлены некоторые ошибки:
  - Устранена проблема с механизмом многопоточного копирования файлов, предотвращающим многократное копирование одного файла. Ранее из-за этой проблемы могла не создаваться резервная копия при высокой нагрузке на сервер.
  - Уровень записи в журнал сообщений вида «checking WAL file name» (проверка имени файла WAL), выдающихся при выполнении команды `show` с параметром `--archive`, понижен с `INFO` до `VERBOSE`.
  - Изменена логика команды `add-instance`: теперь обязательно использовать параметр `-D`, указывающий путь к каталогу. Команда `add-instance` больше не использует переменную

среды `PGDATA` как расположение каталога, а требует явного указания пути к каталогу в параметре команды.

- Модуль pgpro_pwr обновлён до версии 4.8, в которой добавлены новые возможности и исправлены некоторые ошибки. Основные изменения перечислены ниже:
  - Добавлено отслеживание версий расширений, установленных в кластере, в новом разделе отчёта.
  - Добавлена возможность скрыть данные для определённых баз данных в отчёте. Для этого в функции формирования отчёта добавлен новый параметр, принимающий массив исключаемых баз данных.
- Расширение pgpro_stats обновлено до версии 1.8.1, в которую включены следующие усовершенствования и исправления:
  - Устранена проблема, которая могла возникнуть во время трассировки сеанса и вызвать ошибки «could not write file ""pg_stat/pgpro_stats_filters.trace.tmp"": No such file or directory» (не удалось записать файл ""pg_stat/pgpro_stats_filters.trace.tmp"": Такого файла или каталога не существует). Параллельная запись из разных сеансов в файл трассировки теперь работает корректно, и процессы могут выполнять запись в файл трассировки параллельно без конфликтов.
  - Реализована оптимизация, которая в некоторых случаях снижает время блокировки при копировании метрик из локальной памяти в общую.
- Обновлён модуль pg_variables. Команда `DISCARD ALL` теперь очищает все пакеты и переменные в pg_variables.
- Обновлён модуль rum: изменено сообщение об ошибках, возникающих в случаях, когда в запросе `SELECT` используется `ORDER BY`.

E.3.2. Миграция на версию 17.4.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Версия ABI может измениться в корректирующем выпуске Postgres Pro. В таком случае при попытке запустить расширение возникает ошибка `ABI mismatch` (Несоответствие ABI), поэтому обязательно установите новую версию расширения, поставляемую с новым выпуском Postgres Pro, или перекомпилируйте стороннее расширение, чтобы использовать его с текущей версией Postgres Pro.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 17.0.1.

E.4. Postgres Pro Standard 17.2.1

Дата выпуска: 2024-12-03

E.4.1. Обзор

Этот выпуск основан на PostgreSQL 17.2 и Postgres Pro Standard 17.0.1. Все усовершенствования, унаследованные от PostgreSQL 17.2, перечислены в Замечаниях к выпуску PostgreSQL 17.2. Другие важные изменения и улучшения перечислены ниже:

- Добавлена возможность вместо `Append` формировать операцию `MergeAppend`, которая позволяет исключить дополнительную сортировку, необходимую для гарантии сортировки данных. Теперь `MergeAppend` формируется с указанием столбцов для сортировки. Новая возможность применяется только в том случае, если параметр enable_appendorpath имеет значение `ON`.
- Исправлена ошибка при обработке предложения `PLAN` в функции `JSON_TABLE`. Данные, возвращаемые предложениями `NESTED PATH`, ранее могли неверно соединяться с создаваемым пред-

ставлением. В результате при выполнении такого запроса могли быть не выданы требуемые строки, или, наоборот, выданы лишние.

- Устранена ошибка сегментации, которая могла возникнуть при выполнении команды `COPY TO` при использовании модуля online_analyze.
- Устранена проблема, которая могла замедлить выполнение запроса. Причина заключалась в выборе неоптимального индекса оптимизатором из-за отсутствия оценки его стоимости после удаления избыточных предложений с включённым (значение `ON`) параметром enable_appendorpath.
- Добавлена поддержка ОС Ubuntu 24.10.
- Приложение mamonsu обновлено до версии 3.5.9, в которой улучшена производительность и исправлены некоторые ошибки. Основные изменения перечислены ниже:
  - Добавлена поддержка систем, в которых установлена версия *setuptools* выше 67.7.2.
  - Прекращено использование указания `user:group` с точками на этапе перед установкой RPM.
- Расширение pg_portal_modify обновлено до версии 0.3.4.
- Приложение pg_probackup обновлено до версии 2.8.5, в которой улучшена производительность и исправлены некоторые ошибки:
  - Добавлена возможность вывода команды `version` в формате JSON с помощью `--format=json`.
  - Добавлен параметр `--all` для команды `amcheck`, который позволяет выполнить все проверки в одной команде.
  - Добавлена поддержка параметра `include` в файле конфигурации. Содержимое файла, указанного в `include`, добавляется в файл конфигурации вместо этого параметра.
  - Добавлена возможность указывать конкретное количество потоков для выполнения резервного копирования и его последующей проверки, указав их в параметрах `--backup-threads` и `--validate-threads` соответственно.
  - Изменён приоритет значений `PGDATA` и `BACKUP_PATH`. Теперь приоритет имеют значения из параметров командной строки.
  - Добавлена проверка файлов WAL для резервных копий, объединяемых командой `merge`.
  - Сообщение об ошибке, которое выдавалось, когда команда `validate`, запущенная с параметром `--wal`, обнаруживала архив в состоянии `DEGRADED`, было заменено предупреждением, поскольку резервные копии в этом состоянии не влияют на корректность восстановления в целом.
  - Устранена проблема, которая могла возникнуть при слишком длинной командной строке.
  - Устранена ошибка, которая могла возникнуть во время проверки или восстановления, если использовался механизм PITR из предыдущей линии времени.
- Расширение PTRACK обновлено для предотвращения возможных проблем с резервными копиями PTRACK путём автоматического удаления файла `ptrack.map` при отключении PTRACK.

E.4.2. Миграция на версию 17.2.1

Если вы производите обновление выпуска Postgres Pro Standard, базирующегося на той же основной версии PostgreSQL, достаточно просто установить новый выпуск в текущий каталог инсталляции.

Версия ABI может измениться в корректирующем выпуске Postgres Pro. В таком случае при попытке запустить расширение возникает ошибка `ABI mismatch` (Несоответствие ABI), поэтому обязательно установите новую версию расширения, поставляемую с новым выпуском Postgres Pro, или перекомпилируйте стороннее расширение, чтобы использовать его с текущей версией Postgres Pro.

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, обратитесь к инструкциям в Замечаниях к выпуску Postgres Pro Standard 17.0.1.

E.5. Postgres Pro Standard 17.0.1

Дата выпуска: 2024-11-02

E.5.1. Обзор

Этот выпуск основан на PostgreSQL 17 и включает все новые возможности, появившиеся в PostgreSQL 17. Их подробное описание вы можете найти в Замечаниях к выпуску PostgreSQL 17. Ниже перечислены другие основные изменения и усовершенствования:

- Добавлены подкоманды `SPLIT PARTITION` и `MERGE PARTITIONS` в команду `ALTER TABLE`. Подкоманды позволяют улучшить управление секционированными таблицами: первая разделяет одну секцию на несколько, вторая объединяет несколько секций в одну.
- Удалена возможность создания конструкций с вызовом функции `JSON_EXISTS()` с предложением `RETURNING`, которое в более ранних версиях позволяло возвращать значения любого типа. Этот синтаксис не поддерживается стандартом SQL/JSON, согласно которому предикат `JSON_EXISTS()` должен возвращать только TRUE, FALSE или UNKNOWN.
- Добавлено расширение pgpro_autopart, обеспечивающее динамическое создание секций с помощью триггеров для представлений секционированных таблиц.
- Драйвер ODBC обновлён до версии 17.00.0002.
- Модуль aqo обновлён до версии 3.0, в которую включены следующие основные изменения и усовершенствования:
  - Реализован режим «песочницы», позволяющий работать в изолированной среде, не затрагивая основную базу знаний aqo. Этот режим можно включить как на ведущем, так и на резервном узле, установив для параметра aqo.sandbox значение `on`.
  - Добавлен механизм обучения, при котором aqo корректирует оценку количества строк планировщика с помощью собственных предсказаний. Его можно включить с помощью параметра aqo.delta_rows.
- В модуль online_analyze добавлена поддержка команды `COPY`.
- Расширение oracle_fdw обновлено до версии 2.7.0.
- Модуль orafce обновлён до версии 4.13.5.
- Утилита pgpro_controldata обновлена до версии 17.1.0.
- Утилита pg_filedump обновлена до версии 17.1.
- Приложение pg_probackup обновлено до версии 2.8.4.
- Модуль pgpro_pwr обновлён до версии 4.7, в которой улучшена производительность и добавлены новые возможности. Основные изменения перечислены ниже:
  - Механизм *промежуточных выборок* для сбора относительно быстро меняющихся данных.
  - Новые таблицы отчётов, в частности касающиеся состояний сеансов.
  - Поддержка новых статистик Postgres Pro 17.
  - Возможность не сбрасывать статистику расширения по сбору статистики при получении выборки.
- Расширение pgpro_stats обновлено до версии 1.8, в которой добавлена поддержка Postgres Pro 17. Основные изменения перечислены ниже:

  - Обновлены представления `pgpro_stats_statements` и `pgpro_stats_totals`, чтобы включить новые поля, добавленные в модуль pg_stat_statements. Также были обновлены соответствующие функции.
  - Упрощён доступ к представлениям и функциям. В частности, всем пользователям был предоставлен доступ к представлениям `pgpro_stats_archiver`, `pgpro_stats_vacuum_database`, `pgpro_stats_vacuum_tables` и `pgpro_stats_vacuum_indexes`. Ранее для этих представлений требовалось явное предоставление прав доступа. Доступ к выполнению функции `pgpro_stats_trace_reset`, которая ранее могла быть выполнена любым пользователем, теперь ограничен суперпользователями.
- Модуль pg_repack обновлён до версии 1.5.1.
- Расширение pg_wait_sampling обновлено для отслеживания идентификаторов подзапросов и операторов утилит. Также был добавлен параметр `pg_wait_sampling.sample_cpu`, который позволяет указать режим обработки показаний, определяющий, следует ли выполнять обработку показаний процессов, использующих ЦП.
- Расширение PLV8 обновлено до версии 3.2.3.
- Расширение tds_fdw обновлено до версии 2.0.4.
- Функции `pgpro_version`, `pgpro_edition` и `pgpro_build` признаны устаревшими и будут удалены в будущих выпусках. Используйте вместо них параметры конфигурации pgpro_version, pgpro_edition и pgpro_build.
- Удалено признанное устаревшим расширение pg_pathman.
- Прекращена поддержка AlterOS 7, Альт 8.2 СП, Альт 9, Debian 10, SLES 12, Ubuntu 23.10.

Список расширений и дополнительных утилит Postgres Pro Standard, а также основных видимых пользователю отличий от ванильной версии PostgreSQL приведён в Разделе 2.

E.5.2. Миграция на версию 17

Вы можете перейти на Postgres Pro Standard 17 с той же или предыдущей версии PostgreSQL (которая поддерживается выбранным способом обновления) и с предыдущей версии Postgres Pro Standard. То же самое справедливо для перехода на Postgres Pro Standard Certified 17. Способы обновления кластера баз данных описаны в Разделе 17.6. Если у вас возникнут проблемы при переходе на новую версию, обратитесь в службу поддержки Postgres Pro Standard. Обратный переход не поддерживается. Обратите внимание, что переход с Postgres Pro Standard на Postgres Pro Standard Certified той же основной версии (или наоборот) — это обновление на совместимую версию Postgres Pro (за подробным описанием обратитесь к Разделу 17.6).

Для перехода с PostgreSQL или выпуска Postgres Pro Standard, базирующегося на предыдущей основной версии PostgreSQL, сначала установите его последний корректирующий выпуск, а затем выполните выгрузку/восстановление данных, применив pg_dumpall, или воспользуйтесь pg_upgrade.

Если вы мигрируете с Postgres Pro Standard версии 16 или 15 и база данных содержит объекты, использующие недопустимый синтаксис `JSON_EXISTS()` с `RETURNING`, удалите их или замените на корректные, прежде чем выполнять обновление.

Если вы решите использовать pg_upgrade, важно инициализировать новый кластер баз данных с совместимыми параметрами. В частности, обратите внимание на характеристику контрольных сумм в кластере, который вы будете обновлять. Если pg_upgrade создаст какие-либо скрипты SQL в текущем каталоге, выполните их для завершения обновления.

При обновлении инсталляции версии 10 или ниже рекомендуется выполнять выгрузку/восстановление данных. При этом возможные нарушения ограничений вам придётся устранить вручную. Если вариант с выгрузкой данных для вас неприемлем, использовать метод pg_upgrade, при котором в некоторых случаях может нарушиться целостность индексов и ограничений, следует при участии специалистов поддержки Postgres Pro Standard.

> **Примечание**
>
> Во избежание конфликтов не используйте пакет `postgrespro-std-17` для установки исполняемых файлов Postgres Pro, а установите вместо него отдельные пакеты компонентов продукта. В этом случае режим автозапуска сервера, если он требуется, нужно будет включить вручную. Подробнее о предоставляемых пакетах вы можете узнать в Главе 16.

Особенности обновления, унаследованные от ванильной версии PostgreSQL, описаны в Разделе E.12.

E.6. Выпуск 17.6

Дата выпуска: 2025-08-14

В этот выпуск вошли различные исправления, внесённые после версии 17.5. За информацией о нововведениях версии 17 обратитесь к Разделу E.12.

E.6.1. Миграция на версию 17.6

Если используется версия 17.X, выгрузка/восстановление базы не требуется.

Однако если вы используете классы операторов `numeric_minmax_multi_ops` BRIN, после обновления рекомендуется выполнить переиндексацию. Обратитесь к четвёртому пункту в списке изменений ниже.

Если вы обновляете сервер с более ранней версии, чем 17.5, см. также Раздел E.7.

E.6.2. Изменения

- Усиление проверок безопасности в функциях оценки планировщика (Дин Рашид)

  Исправление уязвимости CVE-2017-7484 и последующие исправления направлены на предотвращение применения негерметичных функций к данным статистики по столбцам, на чтение которых у вызывающего пользователя нет прав. В этой защите были найдены две недоработки. Первая относится к секционированию и наследованию иерархий, когда политики RLS в отношении таблиц должны были ограничивать доступ к данным статистики, но не делали этого.

  Вторая недоработка относится к случаям, при которых запрос обращается к таблице через представление, и у владельца представления есть права на чтение нижележащей таблицы, а у вызывающего пользователя прав на доступ к представлению нет. Права владельца представления проходили проверку безопасности, и негерметичная функция применялась к статистике нижележащей таблицы до проверки прав вызывающего пользователя на доступ к представлению. Это было исправлено за счёт того, что проверки безопасности в отношении представлений теперь проводятся в начале планирования. Поэтому теперь ошибки прав могут возникать раньше.

  Проект PostgreSQL благодарит Дина Рашида за сообщение об этой проблеме. (CVE-2025-8713)

- Предотвращение использования скриптов pg_dump для атак на пользователя, выполняющего восстановление (Натан Боссарт)

  Поскольку для операций выгрузки/восстановления обычно нужно запускать команды SQL от имени суперпользователя, целевая инсталляция обязана доверять исходному серверу. Однако из этого не следует, что пользователь операционной системы, использующий psql для восстановления, должен доверять такому серверу. Риск заключается в том, что злоумышленник, получивший права суперпользователя для управления исходным сервером, может отправить с него текст, который будет интерпретирован psql как метакоманды. Таким образом можно получить доступ уровня оболочки к учётной записи восстанавливающего пользователя, независимо от доступа к целевой базе данных.

Чтобы гарантировать, что этого не случится, в psql добавлена команда `\restrict`, которая предотвращает выполнение дальнейших метакоманд, и pg_dump выполняет её до того, как будут получены данные от исходного сервера.

Проект PostgreSQL благодарит Мартина Рахманова, Матье Денэ и РётаК за сообщение об этой проблеме. (CVE-2025-8714)

- Преобразование символов новой строки в пробелы в именах, указанных в комментариях вывода pg_dump (Ной Миш)

  Имена объектов, содержащие символы новой строки, позволяли внедрять произвольные команды SQL в скрипт вывода. (Без предыдущего исправления таким образом могли внедряться и метакоманды psql.) Этот класс проблем был устранён в рамках исправления CVE-2012-0868, однако последующая работа выявила некоторые повторные случаи.

  Проект PostgreSQL благодарит Ноя Миша за сообщение об этой проблеме. (CVE-2025-8715)

- Исправление неправильного расчёта расстояния в опорной функции BRIN `numeric_minmax_multi_ops` (Питер Эйзентраут, Том Лейн)

  На 64-битных платформах результаты были ошибочны иногда, в то время как на 32-битных — крайне часто. Это не приводило к очевидным сбоям, поскольку такая логика используется только для того, чтобы выбрать способ слияния значений в диапазоны. В худшем случае индекс раздувался и становился неэффективным. Однако всё равно рекомендуется переиндексировать любой индекс BRIN, который использует класс оператора `numeric_minmax_multi_ops`.

- Запрет снижения размера принимаемого XML ввода (Микаэль Пакье, Эрик Винхольд)

  В нашем обходном решении для ошибки в ранних выпусках libxml2 2.13.x использовалось место в коде, которое не принимало фрагменты текста, превышающие 10 МБ. У предыдущей версии кода такой проблемы не было. Ранние выпуски предположительно уже не поддерживаются, поэтому возвращаемся к использованию предыдущей версии кода.

- Исправление работы команды `MERGE` при параллельных изменениях (Дин Рашид)

  Если `MERGE` в общем табличном выражении пытается изменить или удалить таблицу с триггерами `BEFORE ROW`, и параллельная операция `UPDATE` или `DELETE` меняет целевую строку, `MERGE` завершится ошибкой (сбой при изменении и возможное некорректное поведение при удалении).

- Исправление ошибок прямого наследования родительской таблицы при команде `MERGE` (Дин Рашид)

  Вставки в такую целевую таблицу могли приводить к сбою или некорректным результатам запроса из-за неправильной обработки действий `WITH CHECK OPTION` и `RETURNING`.

- Триггеры уровня оператора теперь могут становится секциями или потомками в иерархии наследования (Эцуро Фудзита)

  Секциям или дочерним таблицам запрещено иметь триггеры на уровне строк с переходными таблицами, поскольку операция на всём дочернем дереве потребует поддержки отдельной переходной таблицы для каждой подобной дочерней таблицы. Однако проблема не возникает при использовании триггеров на уровне операторов, поскольку срабатывают только триггеры на уровне операторов родительской таблицы. В коде, который проверяет, может ли существующая таблица стать секцией или дочерний таблицей, не сработал ни один из двух видов триггеров.

- Недопущение сбора переходных кортежей для сторонних дочерних таблиц (Эцуро Фудзита)

  Триггеры с переходными таблицами на сторонних таблицах не поддерживаются. Однако ранее не был изучен случай секции или дочерней таблицы, которая является сторонней таблицей. Если у родительской таблицы есть подобный триггер, то с дочерней таблицы собираются неверные переходные кортежи. Вместо этого теперь появляется ошибка о том, что подобное поведение не поддерживается.

- Возможность сброса неизвестных пользовательских параметров с зарезервированным префиксами (Натан Боссарт)

  Ранее, если значение параметра сохранялось с помощью команды `ALTER DATABASE/ROLE/SYSTEM`, то его нельзя было сбросить, если параметр был неизвестен, но у него был зарезервированный префикс. Это может произойти, если раньше этот параметр был в расширении, но после был удалён или обновлён.

- Устранение возможной взаимоблокировки при использовании команды `ALTER SUBSCRIPTION ... DROP PUBLICATION` (Аджин Чериан)

  Обеспечивает блокировку серверных процессов в правильном порядке при отмене источника репликации.

- Сокращение окна условий гонки для создания индексов с конфликтующими именами (Том Лейн)

  При выборе автоматически сгенерированного имени для индекса избегайте конфликтов с подтверждёнными и ещё не подтверждёнными строками `pg_class`. Это позволяет избежать возможного заимствования имени, как в случае с использованием команды `CREATE INDEX`, когда она находится в процессе заполнения индекса или уже завершена, но в рамках ещё не подтверждённой транзакции.

- Предотвращение использования неверных параметров `VACUUM` в случаях очистки нескольких таблиц в рамках одной команды (Натан Боссарт, Микаэль Пакье)

  Параметры `TRUNCATE` и `INDEX_CLEANUP` одной таблицы могут применяться к другим таблицам.

- Обеспечение своевременного обновления карты свободного пространства при очистке таблице без индексов (Масахико Савада)

  Предыдущая оптимизация привела к тому, что такая очистка иногда пропускала подобные таблицы.

- Исправление обработки классов символов в регулярных выражениях `SIMILAR TO` (Лауренц Альбе)

  Код, переводящий выражения, соответствующие шаблону `SIMILAR TO`, в регулярные выражения POSIX, не воспринимал квадратные скобки как вложенный объект. Например, в шаблоне `[[:alpha:]%_]` символы `%` и `_` были восприняты кодом как метасимволы, а не константы.

- Добавление скобок вокруг выражения в предложениях `FETCH FIRST` *выражение* `ROWS WITH TIES` во всех случаях разбора запросов (Хейкки Линнакангас)

  Это позволяет избежать ошибок, если разобранные результаты были синтаксически неверными.

- Ограничение размера очереди запроса `fsync` процесса контрольной точки (Александр Коротков, Сюнэн Чжоу)

  При высоких значениях `shared_buffers` процесс контрольной точки может попытаться выделить больше одного гигабайта памяти для запросов `fsync`, что приведёт к сбою и бесконечному циклу. Чтобы этого избежать, необходимо ограничить размер очереди.

- Устранение бесконечного ожидания в логическом декодировании при чтении частично добавленных записей WAL (Вигнеш Си)

  Если после записи первой части нескольких страниц журнала предзаписи происходит сбой сервера, последующее логическое декодирование потока WAL будет ждать, пока данные запишутся на следующую страницу журнала. Это может произойти даже если сервер уже бездействует.

- Приведение к единообразию написания имён LWLock для `MultiXactOffsetSLRU` и `MultiXactMemberSLRU` (Бертран Друво)

  Ранее это приводило к отображению разных имён событий ожидания в `pg_wait_events` и `pg_stat_activity`, что в теории могло нарушить запросы мониторинга, ассоциирующиеся с этими представлениями.

- Исправление разночтений в использовании кавычек с именами ролей в списке управления доступом (Том Лейн)

  Предыдущее правило зависело от локали, что могло привести к проблемам портирования при переносе значений `aclitem` между инсталляциями. (pg_dump этого не делает, но другие утилиты могут с этим столкнуться.) Для обеспечения единообразия всегда берите в кавычки символы, не относящиеся к ASCII, в выводе `aclitem`. Однако для обратной совместимости кавычки не должны быть обязательными для `aclitem` ввода.

- Запрещение знака равенства (=) в названиях параметров отношения и параметров сторонних данных (Том Лейн)

  Нет явного доказательства использования подобных названий параметров, а допущение использования этого знака приводит к неоднозначности в уже существующих.

- Исправление потенциально неправильной распаковки данных, сжатых с помощью LZ4 (Михаил Грибков)

  Данная ошибка появляется только с плохо сжимаемыми входными данными, поэтому её было сложно отследить.

- Исправление редкого сценария, при котором сканирование индекса методом `btree` ошибочно отмечало записи индексов как нерабочие (Питер Гейган)

- Исправление перераспределения сообщений аннулирования кеша от других транзакций во время логической репликации (Вигнеш Си)

  В ходе исправления ошибки предыдущих релизов был обеспечен ответ приёмника репликации на сообщения аннулирования кеша перекрёстных процессов, предотвращая таким образом использование неактуальных каталогов при обновлении репликации. Однако это исправление также повлияло на то, что эти сообщения перераспределялись, и это приводило к экспоненциальному росту числа сообщений аннулирования, а впоследствии к исчерпанию памяти. Теперь полученные сообщения не перераспределяются.

- Исправление неожиданной остановки сервера при неправильной конфигурации синхронизации слота репликации (Фудзии Масао)

  Управляющий процесс postmaster ранее выводил ошибку (и останавливался), если для `sync_replication_slots` было задано значение `true`, а для `wal_level` значение было ниже `logical`. При обновлённом поведении синхронизация слотов отключается, понижается уровень таких сообщений, что предотвращает остановки postmaster.

- Исправление раннего удаления старых журналов предзаписи в контрольных точках (Виталий Давыдов)

  Если точка перезапуска слота репликации смещается вправо в процессе контрольной точки, уже не используемые сегменты журнала предзаписи могут быть преждевременно удалены, что приведёт к ошибке при восстановлении в случае сбоя базы данных сразу по завершении этого процесса. Теперь они хранятся ещё один цикл контрольной точки.

- Запрет на смещение влево подтверждённого слота репликации (Швета Малик)

  В некоторых случаях клиент репликации может обратиться к более раннему LSN, а потому, возможно, отправить после перезапуска более старый LSN. Это не считается ошибкой, пока клиенту не нужно обращаться к журналу предзаписи, находящемуся между двумя этими точками. Однако этот журнал нельзя переотправлять, поскольку это может привести к дублированию данных, поэтому всегда проверяйте, что слот работает с последним подтверждённым LSN.

- Сокращение излишнего времени ожидания перед запуском новых рабочих процессов логической репликации (Том Лейн)

  В некоторых случаях запуск логической репликации мог занимать гораздо больше времени, чем указанно в `wal_retrieve_retry_interval`, перед запуском нового рабочего процесса.

- Исправлена проблема использования освобождённой памяти во время логической репликации команды `INSERT ... ON CONFLICT` (Итан Мерц, Микаэль Пакье)

  Ранее это приводило к неправильному выводу отчётов по отслеживанию выполнения и, в крайних случаях, сбое процесса отправки WAL.

- Возможность прервать ожидание транзакции на резервном сервере (Кевин Кей Бицзюй)

  Создание слота репликации на резервном сервере может потребовать завершения некоторых активных транзакций на основном сервере, после чего они скопируются на резервном. Поскольку время ожидания в данном случае неопределённое, рекомендуется давать возможность отменять такие операции, но в цикле не было проверки на отмену запроса.

- Запрет на отправку со стороны процессов-передатчиков WAL каскадной логической репликации данных, которые не воспроизведены на резервном сервере (Алексей Махмутов)

  Это позволяет избежать сценариев, при которых передатчики WAL застревают при остановке резервного сервера в ожидании воспроизведения, которого не будет, поскольку процесс запуска сервера уже остановлен.

- Исправление утечки памяти по отношениям во время автоочистки (Том Лейн)

- Исправлены утечки памяти, сохранявшиеся в течение всего сеанса при выполнении `XMLSERIALIZE(... INDENT)` (Дмитрий Коваленко, Том Лейн)

- Устранение возможного сбоя из-за нехватки памяти при выделении места для большой порции через механизм выделения места «bump» (Том Лейн)

- Исправление некоторых сценариев, при которых происходит попытка включения в выборку полей, упакованных в TOAST, из системного каталога без какого-либо снимка (Натан Боссарт)

  Раньше могли возникать сбой проверочного утверждения или ошибки вида «cannot fetch toast data without an active snapshot» (прочитать данные TOAST без активного снимка нельзя).

- Исправлена ошибка проверочного утверждения во время межтабличных обновлений ограничений (Том Лейн, Цзянь Хи)

- Устранение ошибочного утверждения, что тег команды был определён в конце `PortalRunMulti()` (Альваро Эррера)

  Ошибка воспроизводилась в крайних случаях, например при пустом значении подготовленного оператора.

- Исправление ошибок проверочных утверждений при разборе `XMLTABLE` (Ричард Гуо)

- Восстановление возможности запускать выражения PL/pgSQL параллельно (Дипеш Дхамелия)

  Понятие «выражение» в PL/pgSQL трактуется очень широко. Оно включает любые SQL-запросы `SELECT`, возвращающие один столбец и не более одной строки. Поэтому в некоторых случаях, например при оценке агрегатной функции, обрабатывается большой объём данных и было бы полезно делать это параллельными рабочими процессами. Раньше это было возможно, однако из-за предыдущего исправления ошибки это стало невозможно.

- Исправление в особых случаях утечек ресурсов при работе механизма сообщения об ошибках PL/Python (Том Лейн)

Исчерпание памяти при работе механизма сообщения об ошибках Python может привести к ошибке при попытке удалить счётчики ссылок на объекты Python, что приведёт исчерпанию памяти во всём сеансе.

- Исправление функции libpq `PQcancelCreate()` для случаев, когда адрес сервера указан при помощи `hostaddr` (Сергей Корнилов)

  Ранее происходил сбой libpq, если использовался полученный объект отмены.

- Исправлено поведение функции libpq `PQport()`, при котором значение NULL возвращалось, только если прошедшее соединение также было NULL (Даниэле Вараццо)

  Хотя это и задокументированное поведение, в последних версиях libpq значение NULL иногда возвращается, если пользователь не указал порт. Пришлось вернуться к историческому поведению PostgreSQL, при котором в таких случая возвращалась пустая строка. (в версиях 18 и будет возвращаться порт по умолчанию, обычно `"5432"`.)

- Исправление ошибки в случаях, когда аутентификация GSSAPI требует пакеты размером более 16 КБ (Джейкоб Чемпион, Том Лейн)

  Такие большие пакеты для аутентификации требуются пользователям Active Directory, которые состоят во многих её группах. Это ограничение вызывает ошибки соединения с неясными сообщениями об ошибках, например «GSSAPI context establishment error: The routine must be called again to complete its function: Unknown error» (Ошибка установления контекста GSSAPI: Операцию необходимо повторить для завершения её работы: Неизвестная ошибка) .

- Устранение сбоя с зависимостью от времени при передаче данных SSL или GSSAPI (Том Лейн)

  При использовании шифрования SSL и GSSAPI в неблокирующем режиме библиотека libpq иногда выдавала ошибку «SSL error: bad length» (Ошибка SSL: неправильная длина) или «GSSAPI caller failed to retransmit all data needing to be retried» (вызывающей стороне GSSAPI не удалось повторно передать все необходимые данные).

- Исправление обращения по нулевому указателю во время поиска подключения в приложениях ecpg (Александр Алексеев)

  Это может произойти только если некоторые соединения приложения названы, а некоторые — нет.

- Улучшение в psql добавления табуляцией параметров команд `COPY` и `\copy` (Атсуши Торикоши)

  Идентичные дополнения были предложены для `COPY FROM` и `COPY TO`, хотя некоторые параметры работают только для отдельных случаях. Принято решение развести эти случаи, чтобы дать более чёткие рекомендации.

- Исправление ошибок проверочных утверждений для pgbench при получении нескольких сообщений о синхронизации конвейера (Фудзии Масао)

- Исправление дублирующего воспроизведения транзакции при инициализации подписки с помощью pg_createsubscriber (Шлок Кьял)

  Ранее последняя транзакция, обработанная в процессе восстановления подписчика, могла отправляться повторно после начала обычной репликации.

- Обеспечение выгрузки комментариев о ненулевых ограничениях типов доменов в pg_dump (Цзянь Хи, Альваро Эррера)

- Обеспечение выгрузки pg_dump комментариев об ограничениях домена в правильном порядке (Цзянь Хи)

  В некоторых случаях команда комментирования может появиться до создания ограничения.

- Обеспечения стабильного порядка сортировки в приложении pg_dump для всех типов объектов БД (Ной Миш, Андреас Карлссон) pg_dump сортирует объекты по их логическим именам перед тем, как пересортировать по зависимостям. Этот вид сортировки не принимал во внимание полный уникальный ключ, а потом воспринимал некоторые типы объектов как правила и ограничения, что могло приводить к разному порядку сортировки для логически одинаковых БД. Из-за этого сложно было сравнивать базы данных по `diff` файлам вывода pg_dump, поэтому была улучшена логика для стабильной сортировки во всех случаях.

- Исправление неправильного разбора типов объектов в файлах фильтров pg_dump (Фудзии Масао)

  Ключевые слова теперь воспринимаются как комбинация с пробельным символом, а не обрезаются на первом же небуквенно-цифровом символе. Это изменение не влияет на корректные ключевые слова, но позволят правильно распознавать некоторые ошибки. Например, `table-data` будет отклонено, хотя ранее оно воспринималось как `table`.

- Исправлена ошибка восстановления больших объектов расширением pg_restore из выгрузок каталогов, сделанных расширением pg_dump ранее 12 версии PostgreSQL (Павел Стехуле)
- Проверка наличия непоследовательных унаследованных ограничений `NOT NULL` в pg_upgrade (Али Акбар)

  Версии ранее PostgreSQL 18 позволяют удалять унаследованное ограничение `NOT NULL` для столбца. Однако это приводит к невозможности восстановить схему и последующей ошибке pg_upgrade. Теперь такие случаи выявляются во время проверок перед запуском pg_upgrade, что позволяет пользователям вносить исправления до инициализация обновления.

- Убрано требование наличия у целевой установки значения по умолчанию для `max_slot_wal_keep_size` во время выполнения pg_upgrade (Дилип Кумар)
- Исправление ошибки проверочного утверждения, если `track_commit_timestamp` включён во время работы initdb (Хайато Курода, Энди Фан)
- Добавление вывода информации о потере данных статистики в записях WAL `PREPARE TRANSACTION` в pg_waldump (Даниил Давыдов)
- Исправление возможных утечек открытых соединений во время установки соединения `contrib/dblink` (Том Лейн)

  В редких случаях, когда мы упираемся в лимит памяти во время вставки нового объекта соединения в хеш-таблицу `dblink`, может произойти утечка открытого соединения до конца сеанса, что оставит недействительный сеанс на удалённом сервере.

- Обеспечение способности `contrib/pg_prewarm` обрабатывать очень большие значения параметра конфигурации `shared_buffers` (Дарья Шанина)

  Рабочий процесс авторазогрева завершался ошибкой выделения памяти, если для параметра `shared_buffers` было задано значение больше примерно 50 миллионов буферов (400 ГБ).

- Устранение сбоя проверочного утверждения в `contrib/pg_prewarm` (Масахиро Икеда)

  Вызов функции `pg_prewarm()` для отношений без физического хранилища (например, для представлений) приводил к сбою проверочного утверждения, хотя негативно на сборки без проверочных утверждений это не влияло. Теперь для таких случаев добавлена проверка ошибок.

- Устранение пропусков в наборе номеров параметров, используемых в нормализованном запросе `contrib/pg_stat_statements` (Сами Имсейх)
- Устранение утечки памяти в методах DirectModify модуля `contrib/postgres_fdw` (Том Лейн)

  Из-за того, что структура `PGresult` сдерживала результаты команды удалённого изменения, происходила утечка памяти для остального сеанса, если запрос завершался ошибкой между вызовами методов DirectModify. Это могло происходить, если запрос обрабатывал данные предложения `RETURNING`.

- Обеспечение поиска каталогов, указанных в параметрах `--with-includes` и `--with-libraries` скрипта configure, до системных каталогов (Том Лейн)

  Основная причина использования этих параметров в том, чтобы пользовательская версия библиотеки могла переопределить её системную версию. Однако это не работало в некоторых средах из-за некорректного порядка параметров в командах, выполняемых сборочными файлами.

- Исправление проверок скрипта configure для функций `__cpuid()` и `__cpuidex()` (Лукас Фиттл, Микаэль Пакье)

  Скрипту configure не удавалось выявить эти предназначенные для Windows функции, поэтому они не использовались. Это приводило к снижению скорости расчётов CRC ниже необходимого уровня, поскольку доступность инструкций для аппаратного обеспечения нельзя было проверить. На практике влияние этой ошибки было ограничено, так как для производственных сборок под Windows инструментарий Autoconf обычно не используется.

- Устранение ошибки сборки с параметром `--with-pam` на платформах на базе Solaris (Том Лейн)

  Solaris не соответствует другим платформам Unix с точки зрения API для аутентификации PAM. Это несоответствие проявляется в предупреждении компилятора «inconsistent pointer» (непоследовательный указатель), которое пока не было исправлено. Однако в наборе компиляторов GCC 14 это ошибка, а не предупреждение, поэтому добавляем исправление кода.

- Обеспечение возможности переноса кода на GNU Hurd (Михаэль Банк, Кристоф Берг, Самуэль Тибо)

  Исправление предположений о `IOV_MAX` и `O_RDONLY`, которые не работают на GNU Hurd.

- Обеспечение полного соответствия использования функции `memset_s()` стандарту C11 (Том Лейн)

  Это предотвращает сбой компиляции на некоторых платформах.

- Отключение предупреждений совместимости при использовании Meson для сборки с MSVC (Питер Эйзентраут)

- Предотвращение предупреждений компилятора неинициализированных значений в коде сравнения JSONB (Том Лейн)

- Исключение предупреждений об устаревании при сборке с использованием libxml2 2.14 и позднее (Микаэль Пакье)

- Устранение проблем компиляции `pg_locale.h` на C++ (Джон Нейлор)

  Заголовочные файлы PostgreSQL обычно должны заворачиваться в `extern "C" { ... }` для включения в расширения, написанные на C++. Это не работало для файла `pg_locale.h`, так как он использует заголовки libicu. Тем не менее это можно обойти за счёт подавления объявлений только на C++ в таких заголовках. Расширения на C++, которым необходимо использовать API libicu на C++, могут делать это, добавляя заголовки libicu перед `pg_locale.h`.

E.7. Выпуск 17.5

Дата выпуска: 2025-05-08

В этот выпуск вошли различные исправления, внесённые после версии 17.4. За информацией о нововведениях версии 17 обратитесь к Разделу E.12.

E.7.1. Миграция на версию 17.5

Если используется версия 17.X, выгрузка/восстановление базы не требуется.

Однако если вы используете ограничения ссылающихся на себя внешних ключей в секционированных таблицах, может потребоваться пересоздать такие ограничения, чтобы удостовериться в их корректном применении. Обратитесь ко второму пункту в списке изменений ниже.

Кроме того, если вы используете фильтры Блума в индексах BRIN, после обновления рекомендуется выполнить переиндексацию. Обратитесь к третьему пункту в списке изменений ниже.

Если вы обновляете сервер с более ранней версии, чем 17.1, см. также Раздел E.11.

E.7.2. Изменения

- Устранение выхода за границы буфера на один байт при проверке некорректно закодированных строк, которые якобы приведены в кодировке GB18030 (Ной Миш, Андрес Фройнд)

  Если неполный многобайтовый символ появлялся за пределами памяти, на сервере и в использующих libpq приложениях с невысокой вероятностью мог возникать сбой из-за ошибки сегментации SIGSEGV. (CVE-2025-4207)

- Исправление обработки ссылающихся на себя внешних ключей в секционированных таблицах (Альваро Эррера)

  При создании или присоединении секций для ограничения внешнего ключа не создавались необходимые записи каталога, если это ограничение ссылалось на ту же секционированную таблицу. По этой причине ограничение не применялось в полной мере.

  Чтобы устранить проблему, удалите и пересоздайте ссылающиеся на себя внешние ключи в секционированных таблицах, если секции были созданы или присоединены после создания ограничения. Обратите внимание, что строки, нарушающие ограничение, уже могут содержаться в таблицах, поэтому пересоздать ограничение в этом случае не получится и придётся исправить эти строки, чтобы попробовать устранить проблему ещё раз.

- Предотвращение потери данных во время слияния сжатых сводок индекса BRIN при помощи функции `brin_bloom_union()` (Томаш Вондра)

  В коде не учитывались результаты распаковки из-за их неидентичности входным данным. По этой причине некоторые данные не попадали в объединённую сводку, что приводило к потере строк при поиске по индексу.

  Эта ошибка присутствовала ещё с PostgreSQL 14, где появилась возможность использовать фильтры Блума в индексах BRIN. Однако это место кода тогда практически не задействовалось. Теперь в PostgreSQL 17 вероятность натолкнуться на эту ошибку существенно возросла, поскольку код используется при параллельном построении индексов.

- Устранение непредвиденных ошибок «attribute has wrong type» (атрибут имеет неверный тип) в запросах `UPDATE`, `DELETE` и `MERGE`, при которых переменные «вся строка» таблицы ссылаются на представления или функции в предложении `FROM` (Том Лейн)

- Исправление слияния с секционированной таблицей при помощи команды `MERGE` с действиями `DO NOTHING` (Тендер Ван)

  В некоторых случаях возникала ошибка «unknown action in MERGE WHEN clause» (неизвестное действие в предложении MERGE WHEN).

- Устранение сбоя команды `INSERT`, который мог возникать, если у таблицы был столбец, определённый со свойством `GENERATED`, с типом данных домена и ограничения домена запрещали значения `NULL` (Цзянь Хи)

  Ошибка ограничения возникала даже при нормальных результатах генерирующего выражения.

- Исправление обработки ссылок на имена внешних CTE внутри предложения `WITH`, присоединённого к команде `INSERT`/`UPDATE`/`DELETE`/`MERGE`, которая сама находится внутри предложения `WITH` (Том Лейн)

Анализатор запросов не выявлял недопустимые случаи рекурсивных запросов и не учитывал такие ссылки при сортировке CTE в корректном порядке выполнения.

- Исправление некорректной обработки приведений типов в ключах выражений конструкторов JSON (Амит Ланготе)
- Запрет параллельных вызовов функции `array_agg()`, если типом её аргумента является анонимная запись (Ричард Гуо, Том Лейн)

  Протокол взаимодействия параллельных рабочих процессов не поддерживает определение конкретного типа записи, возвращаемой рабочим процессом.
- Исправление вывода некорректных результатов конструкциями `ARRAY(`*`подзапрос`*`)` и `ARRAY[`*`выражение, ...`*`]` при передаче данных типа `int2vector` или `oidvector` (Том Лейн)

  Это исправление восстанавливает поведение, которое демонстрировалось до версии PostgreSQL 9.5. Результат теперь снова выводится с типом `int2vector[]` или `oidvector[]`.
- Исправление возможных некорректных выводов ошибок о недействительных аффиксах во время разбора словарей Ispell (Джейкоб Брейзил)
- Исправление команды `ALTER TABLE ADD COLUMN` для корректной обработки типа данных домена со значением по умолчанию (Цзянь Хи, Том Лейн, Тендер Ван)

  Если у доменного типа было значение по умолчанию, при добавлении столбца этого типа (без явного указания предложения `DEFAULT`) значение домена по умолчанию к существующим строкам не применялось. Вместо этого в новом столбце подставлялось значение `NULL`.
- Исправление некорректного поведения при дублировании имён столбцов в действии `ON DELETE SET DEFAULT` или `SET NULL` ограничения внешнего ключа (Том Лейн)
- Улучшение текста сообщения об ошибке, которое выводится при попытках недопустимого изменения свойств ограничения внешнего ключа (Альваро Эррера)
- Исправление ошибки, возникавшей при сбросе флага `relhassubclass` временной таблицы, для которой указано предложение `ON COMMIT DELETE ROWS` (Ной Миш)
- Добавление недостающего разбора аргумента `INDENT` функции `XMLSERIALIZE()` (Джим Джонс)

  Ранее представления или правила, использующие `XMLSERIALIZE(... INDENT)`, выгружались без предложения `INDENT`, что приводило к некорректным результатам после восстановления.
- Предотвращение преждевременного вычисления аргументов агрегатной функции, которой одновременно переданы аргументы `FILTER` и `ORDER BY` (или `DISTINCT`) (Дэвид Роули)

  При передаче `ORDER BY` или `DISTINCT` возможна предварительная сортировка входных значений агрегатной функции вместо сортировки внутри узла `Agg` плана. Однако это может быть проблемой, если на вход агрегатной функции передаются данные, которые могут привести к ошибке вычисления (например, операция деления с одним из делителей равным нулю), и указано предложение `FILTER`, призванное исключить такие ошибки. Ранее при предварительной сортировке выражения вычислялись до выполнения `FILTER`, и ошибки не исключались. Теперь предварительная сортировка не используется, если указано предложение `FILTER` и входные данные сложнее, чем простые переменные или константы.
- Исправление ошибочных выводов об ограничениях столбцов `NOT NULL` при внешних соединениях (Ричард Гуо)

  В некоторых случаях планировщик отбрасывал условие запроса `IS NOT NULL`, хотя оно применялось после внешнего соединения и поэтому не являлось избыточным.
- Недопущение некорректных оптимизаций на основе проверок `IS [NOT] NULL`, применяющихся к составным значениям (Брюс Момджян)

- Исправление ошибки планировщика при определении более одного хешируемого подвыражения `ScalarArrayOpExpr` внутри выражения верхнего уровня (Давид Гайер)

  Это приводило к неэффективному выполнению дополнительных подвыражений, которые могли быть обработаны через хеш-таблицу (а именно предложений `IN`, `NOT IN` или `= ANY`, которым передавались только константы).

- Исправление некорректной оценки размера таблицы с низким фактором заполнения (Томаш Вондра)

  Когда планировщик оценивал количество строк в таблице, которая ещё никогда не анализировалась, он учитывал значение фактора заполнения, но не гарантировал как минимум одну строку на страницу. Поэтому низкое значение фактора заполнения могло приводить к необоснованно низкой оценке.

- Отключение оптимизации «пропуск выборки» в сканировании кучи по битовой карте (Маттиас ван де Меент)

  Оказалось, что из-за этой оптимизации могли возвращаться мёртвые кортежи, когда параллельные операции очистки помечали страницу как полностью видимую.

- Устранение проблем производительности при запуске поиска по индексу GIN с большим числом поисковых ключей (Том Лейн, Винод Шридхаран)

  Поиск по индексируемым предложениям с большим числом поисковых ключей (например, `jsonbcol ?| array[...]` с десятками тысяч элементов массива) запускался через $O(N^2)$ времени. В течение этого временного промежутка его также нельзя было отменить.

- Выявление недостающих опорных процедур в классе операторов индекса BRIN и вывод ошибки вместо сбоя (Альваро Эррера)

- Реагирование на прерывания (например, отмену запроса) во время ожидания асинхронных подпланов узла плана `Append` (Хейкки Линнакангас)

  Ранее ничего не происходило, пока один из подпланов не был готов.

- Более частая передача статистики ввода-вывода от активных передатчиков WAL (Бертран Друво)

  Ранее для процесса-передатчика WAL в представлении `pg_stat_io` статистика ввода-вывода отображалась только после его завершения. Теперь статистика обновляется максимум с секундной задержкой.

- Устранение условий гонки при обработке параметра конфигурации `synchronous_standby_names` сразу после запуска сервера (Максим Мельников, Микаэль Пакье)

  В течение короткого промежутка времени после запуска системы обслуживающие процессы могли не дожидаться синхронного подтверждения транзакций, несмотря на установленный параметр `synchronous_standby_names`.

- Исправление обработки возможных изменений параметра `io_combine_limit` внутри запроса (Томас Манро)

- Устранение бесконечного цикла при значении INT_MAX для `scram_iterations` (Кевин К. Биджу)

- Предотвращение возможных сбоев из-за двойного преобразования подзапроса функции `json_array()` (Том Лейн)

- Устранение сбоя функции `pg_strtof()` при передаче `NULL` в `endptr` (Александр Лахин, Том Лейн)

- Устранение сбоя из-за нехватки памяти при установлении значений для некоторых GUC (Даниэль Густафссон)

- Устранение сбоя при нехватке памяти в стеммере Snowball (Максим Коротков)
- Исправление чрезмерного выделения памяти для структур `SpecialJoinInfo` на этапе планирования запроса (Ричард Гуо)

  Это приводило к сбоям во время планирования запроса при включённой возможности соединения таблиц с учетом секционирования.
- Запрет копирования аннулированных слотов репликации (Шлок Кьял)

  Это позволяет избежать проблем, когда аннулированный слот указывает на уже удалённый WAL.
- Запрет восстановления слотов логической репликации на резервных серверах, не находящихся в режиме горячего резерва (Масахико Савада)

  Это позволяет предотвратить сценарий, при котором слот может оставаться действительным после повышения резервного сервера до ведущего даже при слишком низком значении параметра конфигурации `wal_level`.
- Предотвращение слишком сильного продвижения значения `catalog_xmin` в «ускоренном» режиме логического декодирования (Хоу Чжицзе)

  Ранее это могло приводить к очистке удалённых записей каталога, несмотря на то, что они потенциально могли быть необходимы процессу чтения WAL.
- Предотвращение потери данных при выполнении DDL-операций без сильной блокировки для таблиц, участвующих в логической репликации (Шлок Кьял, Хайато Курода)

  Изменения каталога, вызванные командами DDL, не отражались в процессах декодирования WAL, из-за чего последующие изменения декодировались с использованием устаревших данных каталога. Это потенциально могло приводить к повреждению данных.
- Предотвращение некорректного сброса источника репликации при ошибке рабочего процесса применения изменений, когда ошибка перехватывается и рабочий процесс не завершается (Хайато Курода)

  В результате этой ошибки могли применяться дублирующиеся данные.
- Устранение сбоя при логической репликации, если для секционированной таблицы подписчика создан индекс BRIN (Том Лейн)
- Запрет создания дубликата снимка при поиске по индексу во время логической репликации (Хейкки Линнакангас)
- Улучшение выявления подписок, принимающих изменения от разных источников (Хоу Чжицзе, Шлок Кьял)

  Если для таблицы, на которую создаётся подписка, уже существуют подписки в рамках других публикаций, при создании подписки выводится предупреждение, поскольку в таком случае могут поступать дублирующие данные. Изменение улучшает эту логику и позволяет выявлять случаи, когда в рамках другой публикации создана подписка для родительской секции или дочерней таблицы.
- Исправление некорректной информации о контрольной точке в сообщении об ошибке, выводимом в случае неверного выбора линии времени для восстановления (Дэвид Стил)

  Если запрошенная линия времени недостижима, значения выводимых в сообщении контрольной точки и линии времени должны считываться из файла `backup_label`, если он создан. Ранее в сообщение попадали значения из управляющего файла, что корректно при восстановлении из управляющего файла без файла `backup_label`, но некорректно при наличии файла `backup_label`.
- Исправление порядка операций в функции `smgropen()` (Андрес Фройнд)

Теперь гарантируется, что перед вызовом обработчика `smgr_open` объект `SMgrRelation` полностью инициализирован. Так он может быть корректно очищен, если при вызове обработчика возникнет сбой.

- Удаление некорректного проверочного утверждения в функции `pgstat_report_stat()` (Микаэль Пакье)
- Исправление слишком строгого проверочного утверждения в функции `gistFindCorrectParent()` (Хейкки Линнакангас)
- Предотвращение сбоя проверочного утверждения во время параллельной очистки, если для параметра конфигурации `maintenance_work_mem` установлено очень небольшое значение (Масахико Савада)
- Устранение сбоя проверочного утверждения, возникавшего в редких случаях на резервных серверах при перезапуске ведущего (Хейкки Линнакангас)
- Исправление ошибки «unexpected plan node type» (неожиданный тип узла плана) в PL/pgSQL при определении прокручиваемого курсора для простого запроса `SELECT` *`выражение`* (Андрей Лепихов)
- Запрет удаления отдельных секций индекса в режиме `--clean` утилиты pg_dump (Цзянь Хи)

  Сервер не принимает такие команды `DROP`. Однако реальных последствий это не вызывает, поскольку секции всё равно удалятся при последующем удалении их родительских таблиц или секционированного индекса. Тем не менее выводимая ошибка о попытке удаления мешала во время восстановления в режиме `--single-transaction`.
- Запрет генерирования недействительных команд `GRANT` для ролей утилитой pg_dumpall, если в каталоге `pg_auth_members` содержатся недействительные OID ролей (Том Лейн)

  Вместо этого теперь выводится предупреждение, и вхождение каталога пропускается. Такое решение позволяет лучше справляться с повреждением каталога, которое могло возникать в ветвях старых версий в результате условий гонки между командами `GRANT` и `DROP ROLE`.
- Использование корректной функции для высвобождения памяти, выделенной libpq, для утилит pg_amcheck и pg_upgrade (Микаэль Пакье, Ранье Вилела)

  Эти недоработки могли приводить к сбоям в некоторых конфигурациях сборок для Windows, например в отладочной сборке libpq, используемой неотладочной сборкой вызывающего приложения.
- Исправление планирования параллельного перестроения индексов утилитой reindexdb (Александр Коротков)

  Предыдущая реализация не позволяла достичь ожидаемого уровня параллельности.
- Устранение сбоя при передаче повреждённых входных данных функции `heap_page_items()` модуля `contrib/pageinspect` (Дмитрий Коваленко)
- Устранение сбоя проверочного утверждения в функции `pg_freespace()` модуля `contrib/pg_freespacemap` (Тендер Ван)

  Вызов функции `pg_freespace()` для отношений без физического хранилища (например, для представлений) приводил к сбою проверочного утверждения, хотя негативно на сборки без проверочных утверждений это не влияло. Теперь для таких случаев добавлена проверка ошибок.
- Запрет подтягивать условия ограничений из подзапросов в модуле `contrib/postgres_fdw` (Александр Пыхалов)

  Это исправление позволяет предотвратить возникавшую в редких случаях ошибку «unexpected expression in subquery output» (непредвиденное выражение в результате подзапроса).

- Исправление ошибки сборки, возникавшей при наличии старой версии файла `libpq_fe.h` в системных каталогах `include` (Том Лейн)
- Исправление ошибки сборки в macOS 15.4 (Том Лейн, Питер Эйзентраут)

  Обновление macOS повредило проверку конфигурации для функции `strchrnul()`.
- Исправление маркировки данных отдельных буферов потоков чтения для Valgrind (Томас Манро)

  Это исправление не влияет на код ядра выпущенных версий PostgreSQL, однако расширения, использующие данные отдельных буферов, могли сталкиваться с ложными ошибками во время тестирования под Valgrind.
- Устранение предупреждений Valgrind, связанных с кодом хеширования строк (Джон Нейлор)
- Обновление данных часовых поясов до версии tzdata 2025b, включающее изменение правил перехода на летнее время в Чили, а также корректировку исторических данных для Ирана (Том Лейн)

  Появился новый часовой пояс America/Coyhaique для области Айсен в Чили, который позволяет учитывать для неё часовой пояс UTC-03 без перехода на летнее время и соответствующее расхождение с поясом America/Santiago.

E.8. Выпуск 17.4

Дата выпуска: 2025-02-20

В этот выпуск вошло несколько исправлений, внесённых после версии 17.3. За информацией о нововведениях версии 17 обратитесь к Разделу E.12.

E.8.1. Миграция на версию 17.4

Если используется версия 17.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 17.1, см. также Раздел E.11.

E.8.2. Изменения

- Улучшение поведения функций с обработкой экранирования libpq (Андрес Фройнд, Том Лейн)

  Изменения, внесённые для CVE-2025-1094, имели одно серьёзное упущение: функции `PQescapeLiteral()` и `PQescapeIdentifier()` не учитывали параметр длины строк и вместо этого всегда выполняли чтение до завершающего нулевого символа входной строки. Это приводило к включению нежелательного текста в вывод при усечении строки вызывающим кодом через параметр длины, а в некоторых особенно неудачных случаях могло вызывать сбои из-за считывания за пределами памяти.

  Кроме того, все функции с обработкой экранирования были изменены так, чтобы при обнаружении недопустимой кодировки недопустимая последовательность заменялась только на первый байт предполагаемого символа, а не на весь символ. Это снижает риск возникновения проблем, если вызывающее приложение выполняет дополнительную обработку строки с экранированием.
- Устранение небольшой утечки памяти в pg_createsubscriber (Ранье Вилела)
- Исправление системы сборки meson для корректного определения доступности системного заголовка `bsd_auth.h` (Назир Билал Явуз)

E.9. Выпуск 17.3

Дата выпуска: 2025-02-13

В этот выпуск вошли различные исправления, внесённые после версии 17.2. За информацией о нововведениях версии 17 обратитесь к Разделу E.12.

E.9.1. Миграция на версию 17.3

Если используется версия 17.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 17.1, см. также Раздел E.11.

E.9.2. Изменения

- Усиление `PQescapeString` и родственных функций для защиты от некорректно закодированных входных строк (Андрес Фройнд, Ной Миш)

  Функции с обработкой экранирования, предоставляемые libpq, теперь полностью проверяют допустимость кодировки входных данных. При обнаружении недопустимых символов выдаётся ошибка, если это возможно. Для функций, не возвращающих ошибки, выходная строка корректируется, чтобы гарантировать, что сервер распознает недопустимую кодировку, и при промежуточной обработке ошибочный байт не будут принят за одинарную кавычку, обратную косую черту и т. д.

  Цель этого изменения — защита от возможных атак методом SQL-инъекций, если одна из этих функций используется для экранирования специально сконструированного входного значения. Если полученная строка отправляется напрямую на сервер PostgreSQL (который в любом случае проверит её кодировку), опасности нет. Однако риск возникает, если строка проходит через psql или другой код на стороне клиента. Ранее такой код не проверял кодировку достаточно тщательно, и во многих случаях оставалось неясно, как следовало бы реагировать на обнаруженные ошибки.

  Это исправление эффективно только в том случае, если функция с обработкой экранирования, сервер и любые промежуточные процессы используют одну и ту же кодировку. Особенно важно следить за использованием одной кодировки приложениям, которые вставляют недоверенные входные данные в SQL-запросы.

  Приложения и драйверы, экранирующие недоверенные входные данные без использования функций libpq, могут столкнуться с подобными проблемами. Сначала им следует проверять, что входные данные соответствуют ожидаемой сервером кодировке.

  Проект PostgreSQL благодарит Стивена Фьюэра за сообщение об этой проблеме. (CVE-2025-1094)

- Восстановление автоматического усечения имён баз данных и пользователей в запросах на подключение (Натан Боссарт)

  Это отменяет изменение версии 17, которое вызывало проблемы у некоторых пользователей. Имена, превышающие допустимую длину, должны усекаться с учётом кодировки, но на данный момент было возвращено прежнее поведение слепого усечения на `NAMEDATALEN-1` байт.

- Исключение параллельных рабочих процессов из проверок и ограничений прав подключения (Том Лейн)

  Теперь при запуске параллельного рабочего процесса не проверяются привилегии `datalowconn`, `rolcanlogin` и `ACL_CONNECT`. Предполагается, что достаточно ведущему процессу пройти аналогичные проверки. Это позволяет избежать, например, неожиданных завершений параллельных запросов с ошибкой, если ведущий процесс выполнялся от имени роли без права входа. Кроме того, ограничения `ReservedConnections`, `datconnlimit` и `rolconnlimit` теперь применяются только к обычным обслуживающим процессам, и только они учитываются при подсчёте количества подключений. Эти ограничения предназначены для предотвращения чрезмерного потребления слотов подключений для обычных процессов, но у параллельных рабочих процессов и других специальных процессов есть собственные пулы слотов с собственными проверками ограничений.

- Удаление суффикса «Lock» из имён событий ожидания LWLock (Бертран Друво)

  В результате реорганизации кода имена событий, связанных с блокировкой, непреднамеренно стали отображаться в представлении `pg_stat_activity` с суффиксом «Lock», что, помимо прочего, привело к проблемам при соединении с `pg_wait_events`.

- Устранение возможной ошибки при сканировании индекса B-дерева с условием ScalarArrayOp (`= ANY`), при которой возвращались не все соответствующие условию кортежи (Питер Гейган)

- Исправление возможной проблемы с повторным использованием устаревших результатов в оконных агрегатных функциях (Дэвид Роули)

  Оконная агрегатная функция с ограничивающим условием выполнения и результатом типа, передаваемого по ссылке, могла ошибочно возвращать результат из предыдущей секции вместо выполнения нового вычисления.

- Поддержка синхронизации `TransactionXmin` с `MyProc->xmin` (Хейкки Линнакангас)

  Ранее процесс мог попытаться получить доступ к уже удалённым данным. Одним из известных последствий являлись временные ошибки типа «could not access status of transaction» (не удалось получить состояние транзакции).

- Устранение условий гонки, которые могли привести к ошибке при добавлении новой записи каталога в список кеша каталога (Хейкки Линнакангас)

  Это могло приводить, например, к тому, что в текущем сеансе не могла использоваться только что созданная функция.

- Предотвращение возможного повреждения каталога при очистке системного каталога одновременно с его изменением (Ной Миш)

- Устранение повреждения данных при возникновении ошибки во время усечения отношений (Томас Манро)

  Файловые операции, необходимые для усечения отношений, могли завершиться ошибкой, оставив несогласованное состояние на диске (например, фактически восстанавливая удалённые данные). Это невозможно предотвратить полностью, но теперь ошибки такого рода получили уровень PANIC, так что после их появления можно запустить восстановление согласованности данных путём повторного воспроизведения WAL до момента, непосредственно предшествующего попытке усечения. Это не лучшее решение, но такие ошибки возникают достаточно редко, чтобы считать такой подход приемлемым.

- Предотвращение запуска контрольных точек во время усечения отношений (Роберт Хаас)

  Это позволяет избежать условий гонки, при которых изменённый файл мог не сохраниться на диске до завершения контрольной точки, что создавало риск повреждения данных, если вскоре после этого происходил сбой операционной системы.

- Исключение возможной потери изменений `pg_database`.`datfrozenxid` в случае выполнения `VACUUM` одновременно с командой `REASSIGN OWNED`, которая изменяет владельца этой базы данных (Кирилл Решке)

- Исправление передачи некорректных значений `tg_updatedcols` триггерам `AFTER UPDATE` (Том Лейн)

  В некоторых случаях битовая карта `tg_updatedcols` могла описывать набор столбцов, изменённых предшествующей командой в той же транзакции, что заставляло неверно срабатывать триггер.

  Кроме того, предотвращено увеличение используемого объёма памяти, вызванное созданием слишком большого количества копий битовой карты `tg_updatedcols`.

- Исключение отсоединения секции, если у неё есть собственное ограничение внешнего ключа, ссылающееся на секционированную таблицу (Амул Сул)

В обычных случаях внешние ключи определяются на верхнем уровне секционированной таблицы. Ранее, если один из ключей определялся на уровне секции и ссылался на секционированную таблицу, при отсоединении такой секции соответствующие записи `pg_constraint` изменялись некорректно. Это приводило к ошибкам вида «could not find ON INSERT check triggers of foreign key constraint» (не удалось найти триггеры проверки ON INSERT ограничения внешнего ключа).

- Исправление функции `pg_get_constraintdef` для корректной поддержки ограничений `NOT NULL` доменов (Альваро Эррера)
- Исправление обработки кодов формата `FF`*n* функцией `to_timestamp` (Том Лейн)

  Ранее, если непосредственно перед `FF`*n* находился код целочисленного формата, могли использоваться все доступные цифры, оставляя `FF`*n* без значений.
- Реализация проверки заключения имён переменных в двойные кавычки при разборе предложения `PASSING` в функции запроса SQL/JSON (Дин Рашид)
- Реализация проверки заключения имён пространств имён XML в двойные кавычки при разборе выражения `XMLTABLE()` (Дин Рашид)
- Добавление параметра `ldapscheme` в вывод функции `pg_hba_file_rules()` (Лауренц Альбе)
- Исправление планирования предварительно отсортированных операций `UNION` в случаях, когда не все типы данных входных столбцов совпадают (Дэвид Роули)

  Эта ошибка могла приводить к сортировке данных с использованием неправильного оператора сортировки, что могло иметь различные последствия: от отсутствия видимых проблем до аварийных сбоев.
- Исключение объединения операций `UNION`, если их правила сортировки столбцов не согласованы (Том Лейн)

  Ранее правила сортировки игнорировались при принятии решения о том, безопасно ли объединять шаги `UNION` в одну многократную операцию `UNION`. Это могло быть допустимо до введения недетерминированных правил сортировки, но теперь ситуация изменилась, поскольку используемая сортировка может влиять на определение уникальности.
- Предотвращение ошибок планировщика вида «wrong varnullingrels» (неправильные varnullingrel) после «подтягивания» подзапроса, который находится под внешним соединением (Том Лейн)
- Игнорирование маркировки отношений для обнуления при поиске статистики (Ричард Гуо)

  Это упущение могло приводить к невозможности использования соответствующей статистики по выражениям или к ошибкам вида «corrupt MVNDistinct entry» (повреждённая запись MVNDistinct).
- Исправление обработки пропущенных выражений в логике отсечения секций (Том Лейн)

  Это упущение могло приводить к ошибкам вида «unrecognized node type» (нераспознанный тип узла) и потенциально к другим проблемам при выполнении запросов к секционированным таблицам.
- Выделение собственного слота для рабочего процесса slotsync (Том Лейн, Хоу Чжицзе)

  Это было упущено при добавлении рабочего процесса slotsync, в результате чего слот фактически выделялся из пула, предназначенного для обычных внутренних процессов, что в свою очередь могло привести к сбою запуска рабочего процесса или к последующим сбоям запросов на подключение, которые должны были успешно выполняться в соответствии с настроенными параметрами, если количество обычных внутренних процессов приближалось к `max_connections`.
- Снятие ограничения размера таблиц dshash в 1 ГБ (Маттиас ван де Меент)

Это позволяет избежать ошибок типа «invalid DSA memory alloc request size» (недопустимый запрошенный размер разделяемой памяти DSA). Такая ситуация могла возникнуть, например, в транзакциях, обрабатывающих несколько миллионов таблиц.

- Предотвращение возможного целочисленного переполнения в функции `bringetbitmap()` (Джеймс Хантер, Евгений Горбанёв)

  Поскольку результат используется только в статистических целях, последствия этой ошибки были в основном косметическими.

- Исправление вычисления количества сегментов SLRU (Юрий Соколов)

  Ошибка приводила к использованию меньшего количества сегментов, чем предполагалось, что увеличивало конкуренцию, но не вызывало функциональные сбои.

- Исключение поведения, при котором уже установленная блокировка процесса мешала процессу postmaster обрабатывать события сокета (Томас Манро)

  Чрезвычайно большая нагрузка на серверы, запускающие и завершающие работу процессов, могла помешать процессу postmaster своевременно реагировать на входящие клиентские подключения.

- Предотвращение зацикливания резервных серверов потоковой репликации при чтении записей WAL, переходящих на другую страницу (Кётаро Хоригути, Александр Кукушкин)

  Это могло произойти, если запись продолжалась на странице, которую необходимо было прочитать из другого источника WAL.

- Устранение непреднамеренного повышения уровня ошибок FATAL до PANIC в начале запуска процессов (Ной Миш)

  Это исключает некоторые маловероятные случаи, которые могли привести к ошибкам вида «PANIC: proc_exit() called in child process» (ПАНИКА: в дочернем процессе вызван proc_exit()).

- Исключение случаев, в которых оператор семейства операторов или опорная процедура могли стать недействительными (Том Лейн)

  В некоторых случаях тип данных мог быть удалён, а ссылки на его OID по-прежнему оставались в `pg_amop` или `pg_amproc`. Это могло приводить к проблемам в будущем, так как попытка удалить владеющее оператором семейство завершалась ошибкой, и pg_dump генерировал некорректный вывод при выгрузке семейства операторов. В указанном исправлении при создании и изменении семейств/классов операторов добавляются необходимые зависимости, так что при удалении типа данных также удаляются любые зависимые элементы семейства операторов. Однако на уже существующие семейства операторов это исправление не распространяется, поэтому в `DROP OPERATOR FAMILY` также было добавлено временное решение, чтобы предотвратить сбой при удалении семейства, в котором есть недействительные члены.

- Устранение множественных утечек памяти в выводе логического декодирования (Вигнеш Си, Масахико Савада, Боюй Янг)

- Устранение небольшой утечки памяти при изменении параметра `application_name` или `cluster_name` (Тофиг Алиев)

- Предотвращение сбоя в случаях, когда фоновый процесс пытается проверить новое значение `synchronized_standby_slots` (Альваро Эррера)

- Предотвращение целочисленного переполнения при проверке условия `wal_skip_threshold` (Том Лейн)

  Транзакция, которая создавала очень большое отношение, могла по ошибке решить обеспечить надёжность путём копирования отношения в WAL вместо того, чтобы синхронизировать его с помощью fsync, тем самым лишая смысла `wal_skip_threshold`. (Это имеет значение только в случае, когда для `wal_level` установлено значение `minimal`, в противном случае всё равно требуется копия WAL.)

- Исключение небезопасного порядка операций во время поиска в кеше (Ной Миш)

  Единственным известным последствием такого порядка было обычно безобидное предупреждение «you don't own a lock of type ExclusiveLock» (нет блокировки типа ExclusiveLock) при вызове команды `GRANT TABLESPACE`.

- Предотвращение потенциального использования освобождённой памяти при параллельной очистке (Валлимахараджан Г, Джон Нейлор)

  Эта ошибка не имела последствий в стандартных сборках, но теоретически представляла опасность.

- Устранение возможных ошибок вида «failed to resolve name» (не удалось разрешить имя) при использовании JIT на старых платформах ARM (Томас Манро)

  Они могли появляться из-за несоответствия значения по умолчанию для `-moutline-atomics` между компиляторами gcc и clang. В Debian и Ubuntu поставляются компиляторы gcc и clang, ориентированные на armv8-a, но обработка атомарных операций по умолчанию различается.

- Устранение сбоев проверочных утверждений в запросах `WITH RECURSIVE ... UNION` (Дэвид Роули)

- Предотвращение сбоев проверочных утверждений при разборе правил в случаях, когда конечный запрос содержит операции с множествами (Ман Цзэн, Том Лейн)

- Предотвращение сбоев проверочных утверждений в особых случаях при запуске параллельных запросов (Том Лейн)

- Устранение сбоев проверочных утверждений во время завершения работы при записи файла статистики (Микаэль Пакье)

- Устранение предупреждений valgrind, связанных с кодом хеширования строк (Джон Нейлор)

- Предотвращение в функции `NULLIF()` передачи указателя на развёрнутый объект для чтения и записи в функцию равенства для типа данных (Том Лейн)

  Если функция равенства изменяла или удаляла объект, для которого передан указатель на чтение-запись, могли возникать ошибки, если функция `NULLIF()` возвращала этот объект. Встроенные функции равенства не вызывали проблем, но в функциях на PL/pgSQL происходил сбой.

- Обеспечение применения предварительной обработки выражения к значению по умолчанию null в `INSERT` (Том Лейн)

  Если целевой столбец типа домен, планировщик должен добавлять шаг приведения к типу домен, а не просто константу null. Ранее для этого выражения пропускались некоторые требуемые шаги обработки. Для доменов, основанных на встроенных типах данных, о последствиях не сообщалось, но теоретически могла возникнуть ошибка с доменами, основанными на типах из расширений.

- Предотвращение потери данных при начале массовой записи в слой отношений, который уже содержит данные (Маттиас ван де Меент)

  Любые существующие данные обнулялись. В PostgreSQL это не представляло проблемы, но могло проявляться в некоторых расширениях.

- Предотвращение сбоя в случаях, когда серверный процесс пытался выполнить перебор по общему префиксному дереву, которое он не создавал (Масахико Савада)

  В ядре PostgreSQL нет кода, вызывавшего такой сбой, однако он мог оказаться в расширениях.

- Устранение утечек памяти в PL/Python (Мэт Арье, Том Лейн)

  Многократное использование `PLyPlan.execute` или `plpy.cursor` приводило к утечке памяти во время вызова функции PL/Python.
- Исправление PL/Tcl для корректной компиляции с Tcl 9 (Питер Эйзентраут)
- Исключение в препроцессоре ecpg возможной некорректной обработки курсоров, ссылающихся на переменные вне текущей области (Том Лейн)
- Корректная выдача предупреждений в ecpg об отсутствии поддержки `COPY ... FROM STDIN` (Рё Канбаяши)

  Ранее предупреждение не выдавалось из-за опечатки.
- Обеспечение корректной обработки имён путей к файлам, закодированных в SJIS, в psql (Том Лейн)

  У некоторых двухбайтовых символов в SJIS второй байт равнозначен обратному слешу в ASCII (`\`). Эти символы повреждались при нормализации имени пути, из-за чего файлы с такими символами в именах становились недоступны.
- Добавление в psql дополнения табуляцией для `COPY (MERGE INTO)` (Цзянь Хи)
- Исключение использования неправильной версии `pqsignal()` в pgbench и psql (Фудзии Масао, Том Лейн)

  Эта ошибка могла приводить к неправильному поведению при использовании параметра `-T` в pgbench или команды `\watch` в psql из-за того, что прерванные системные вызовы не возобновлялись должным образом.
- Исправление некорректного выполнения некоторых вложенных конструкций `\if` в pgbench (Михаил Николаев)

  Команда `\if` в не той (не выполнявшейся) ветви `\if` ошибочно обрабатывалась так же, как `\elif`.
- Исправление возможного некорректного отображения сообщений о прогрессе выполнения во время инициализации таблицы в pgbench (Юси Огивара, Тацуо Исии, Фудзии Масао)
- Повышение устойчивости pg_controldata к повреждённым файлам `pg_control` (Ян Ильясов, Антон Волошин)

  Поскольку pg_controldata пытается выводить содержимое `pg_control`, даже если не пройдена проверка CRC, должно быть предусмотрено корректное поведение при недопустимых значениях полей. Это исправление исключает некоторые проблемы, вызываемые недопустимыми метками времени и отрицательными размерами сегментов WAL.
- Устранение возможного сбоя в pg_dump из-за последовательностей идентификаторов, которые связаны с таблицами, принадлежащими расширениям (Том Лейн)
- Устранение утечки памяти в pg_restore при работе с данными, сжатыми с использованием zstd (Том Лейн)

  Утечка происходила во время каждой операции распаковки, поэтому она была наиболее заметна в выгрузках, содержащих много таблиц или больших объектов.
- Исправление pg_basebackup для корректной обработки файлов `pg_wal.tar`, размер которых превышает 2 ГБ в Windows (Давиндер Сингх, Томас Манро)
- Использование соответствующих SQL-стандарту тел функций в объявлениях SQL-функций расширения `contrib/earthdistance` (Том Лейн, Ронан Данклау)

  Это изменение позволяет разрешать ссылки на `contrib/cube` во время создания расширения, что снижает риск связанных с путями поиска сбоев и возможных атак.

В частности, это восстанавливает удобство использования этих функций в контекстах сгенерированных столбцов, для которых PostgreSQL версии 17 ограничивает путь поиска по соображениям безопасности. Были получены сообщения о неудачных попытках обновления баз данных до версии 17 из-за этой проблемы. Это исправление было включено в версию 16 для предоставления обходного решения: предварительное обновление расширения `earthdistance` до этой версии должно позволить успешно выполнить обновление.

- Обнаружение несоответствия версий между SQL-объявлениями `contrib/pageinspect` и соответствующей общей библиотекой (Томаш Вондра)

  Ранее такое несоответствие могло привести к сбою при вызове `brin_page_items()`. Теперь вместо этого выдаётся ошибка с рекомендацией обновить расширение.

- Многократное повторение запроса на отмену удалённого запроса в `contrib/postgres_fdw`, если первый запрос мог выполниться неуспешно (Том Лейн)

  Это изменение устраняет условия гонки, при которых попытка отменить только что отправленный запрос происходила до того, как удалённый сервер начинал его обработку, в результате чего первоначальный запрос на отмену игнорировался.

- Изменение проверок конфигурации, определяющих переключатели компилятора, необходимые для доступа к инструкциям ARM CRC (Том Лейн)

  На платформах ARM, где базовый процессор не поддерживает инструкции CRC, необходимо использовать переключатель `-march`, чтобы заставить компилятор скомпилировать такие инструкции. В последних версиях gcc отклоняет ранее используемое значение, что приводило к возврату к программному CRC без уведомления об этом.

- Исправление системы сборки meson для поддержки старых библиотек OpenSSL в Windows (Дарек Слюсарчик)

  Добавление поддержки имён устаревших библиотек `ssleay32` и `libeay32`.

- Обеспечение экспорта всех функций libcommon и libpgport в сборках Windows с использованием meson (Владлен Пополитов, Хейкки Линнакангас)

  Это исправляет ошибки сборки вида «unresolved external symbol» (неразрешённый внешний символ) для расширений.

- Исправление процесса конфигурации meson для корректного определения заголовочного файла OSSP `uuid.h` в MSVC (Эндрю Дунстан)

- Установка `pgevent` в *`pkglibdir`*, а не в *`bindir`* при сборке с помощью meson (Питер Эйзентраут)

  Это соответствует поведению системы сборки на основе make и старой системы сборки MSVC.

- Установка `sepgsql.sql` в `share/contrib/`, а не в `share/extension/` при сборке с помощью meson (Питер Эйзентраут)

  Это соответствует поведению системы сборки на основе make.

- Обновление данных часовых поясов до версии tzdata 2025a, включающее изменение правил перехода на летнее время в Парагвае, а также корректировку исторических данных для Филиппин (Том Лейн)

E.10. Выпуск 17.2

Дата выпуска: 2024-11-21

В этот выпуск вошло несколько исправлений, внесённых после версии 17.1. За информацией о нововведениях версии 17 обратитесь к Разделу E.12.

E.10.1. Миграция на версию 17.2

Если используется версия 17.X, выгрузка/восстановление базы не требуется.

Если вы обновляете сервер с более ранней версии, чем 17.1, см. также Раздел E.11.

E.10.2. Изменения

- Устранение сбоя ABI для расширений, работающих со структурой `ResultRelInfo` (Том Лейн)

  Предыдущие корректирующие выпуски непреднамеренно нарушили бинарную совместимость с timescaledb и несколькими другими расширениями. Восстановите затронутую структуру до её предыдущего размера, чтобы такие расширения не пришлось перестраивать.

- Восстановление функциональности команды `ALTER {ROLE|DATABASE} SET role` (Том Лейн, Ной Миш)

  Исправление для CVE-2024-10978 случайно привело к тому, что параметры для `role` не применялись, если поступали из неинтерактивных источников, включая предыдущие команды `ALTER {ROLE|DATABASE}` и переменную среды `PGOPTIONS`.

- Устранение случаев, в которых значение `restart_lsn` слота логической репликации могло уменьшаться (Масахико Савада)

  Ранее перезапуск логической репликации иногда мог приводить к повторному вычислению точки перезапуска слота как более старого значения, чем было объявлено ранее в `pg_replication_slots`. Из-за этого файлы WAL могли быть удалены на основе более позднего значения `restart_lsn`, и в этом случае репликация не могла перезапуститься.

- Предотвращение удаления нужных файлов WAL во время выполнения pg_rewind (Полина Бунгина, Александр Кукушкин)

  Раньше в некоторых случаях утилита pg_rewind могла удалить важные файлы WAL из восстановленного пониженного ведущего узла. В частности, это происходило, если эти файлы были помечены для архивации (т. е. для них были созданы файлы `.ready`), но ещё не были заархивированы. Затем эти файлы перерабатывались и не появлялись на повышенном узле, но они могли понадобиться для восстановления на пониженном узле. Если pg_rewind удаляла их, восстановление было уже невозможно.

- Устранение условий гонки, связанных с удалением записей общих статистик (Кётаро Хоригути, Микаэль Пакье)

  Эти условия могли приводить к потере статистических данных, сбоям проверочных утверждений или ошибкам вида «can only drop stats once» (статистику можно удалить только один раз).

- Подсчёт количества сканирований индексов `contrib/bloom` в представлениях статистики, таких как счётчик `pg_stat_user_indexes.idx_scan` (Масахиро Икеда)

- Устранение сбоя при проверке изменений параметров классов операторов индекса (Александр Коротков)

  Некоторые формы `ALTER TABLE` завершались ошибкой, если у таблицы был индекс с параметрами классов операторов, отличными от значений по умолчанию.

- Предотвращение сбоев проверочных утверждений, вызванных отключёнными подграфами NFA при разборе регулярных выражений (Том Лейн)

  Эта ошибка не имела видимых последствий в сборках без проверочных утверждений.

E.11. Выпуск 17.1

Дата выпуска: 2024-11-14

В этот выпуск вошли различные исправления, внесённые после версии 17.0. За информацией о нововведениях версии 17 обратитесь к Разделу E.12.

E.11.1. Миграция на версию 17.1

Если используется версия 17.X, выгрузка/восстановление базы не требуется.

Тем не менее, если вы когда-либо выполняли отсоединение секции от секционированной таблицы, которая ссылалась на другую секционированную таблицу при помощи внешнего ключа, и старая секция не была удалена, возможно, вам придётся исправлять повреждение каталогов и/или данных, как подробно описано ниже в пятом пункте списка изменений.

Кроме того, в редких случаях, когда серверная переменная `LC_CTYPE` принимает значение `C`, а в `LC_COLLATE` указана другая локаль, индексы, построенные по текстовым столбцам, должны быть перестроены, как описано ниже в шестом пункте списка изменений.

E.11.2. Изменения

- Пометка кешированных планов как зависимых от выполняющих их ролей, когда защита на уровне строк (RLS) применяется для ссылки на таблицу не на верхнем уровне (Натан Боссарт)

  Если общее табличное выражение (CTE), подзапрос, подссылка, представление с контекстом безопасности вызывающего или проекция приведения в запросе ссылались на таблицу с политикой защиты на уровне строк, результирующий план не помечался как потенциально зависимый от выполняющей его роли. Это могло приводить к тому, что последующие запросы в том же сеансе выполнялись с некорректным планом, а скрытые строки возвращались или возвращаемые скрывались.

  Проект PostgreSQL благодарит Вольфганга Вальтера за сообщение об этой проблеме. (CVE-2024-10976)

- Библиотека libpq теперь игнорирует сообщения об ошибках, получаемые во время согласования протоколов SSL или GSS (Джейкоб Чемпион)

  Злоумышленники могут подменять реальное выходное сообщение сервера об ошибке, получаемое до того, как завершено согласование криптографического протокола. Это может приводить к различным угрозам безопасности, например, злоумышленник может подменить результат запроса, а невнимательный пользователь спутает его с реальным результатом. Лучше всего игнорировать такие сообщения об ошибках и доверять только сообщениям libpq о сбое соединения.

  Проект PostgreSQL благодарит Джейкоба Чемпиона за сообщение об этой проблеме. (CVE-2024-10977)

- Исправление нежелательного взаимодействия команд `SET SESSION AUTHORIZATION` и `SET ROLE` (Том Лейн)

  Согласно стандарту SQL, команда `SET SESSION AUTHORIZATION` имеет побочный эффект в виде выполнения команды `SET ROLE NONE`. Это было реализовано некорректно, что приводило к дополнительному нежелательному взаимодействию между двумя командами. В частности, откат транзакции, в которой была выполнена команда `SET SESSION AUTHORIZATION`, приводил к откату `ROLE` до `NONE`, даже если ранее было другое значение, и действующий идентификатор пользователя мог отличаться от идентификатора до выполнения транзакции. Временное включение `session_authorization` в предложении `SET` функции приводило к тому же результату. Дополнительная ошибка заключалась в том, что при исследовании `current_setting('role')` параллельным рабочим процессом он видел `none`, даже если должен был видеть иное значение.

  Проект PostgreSQL благодарит Тома Лейна за сообщение об этой проблеме. (CVE-2024-10978)

- Запрет на изменение переменных окружения доверенным кодом PL/Perl (Эндрю Дунстан, Ной Миш)

Возможность изменять переменные окружения процессов, такие как `PATH`, позволяет злоумышленнику выполнять произвольный код. Поэтому теперь это запрещено для «доверенных» процедурных языков. Чтобы исправить проблему с `plperl`, замените `%ENV` на связанный хеш, который запрещает вносить изменения с выводом предупреждения. Недоверенный язык `plperlu` сохраняет возможность изменять переменные окружения.

Проект PostgreSQL благодарит Коби Абрамса за сообщение об этой проблеме.
(CVE-2024-10979)

- Исправление обновлений состояния каталога для ограничений внешнего ключа при присоединении или отсоединении секций таблицы (Жеан-Гийом де Рорте, Альваро Эррера)

  Если целевая таблица секционирована, для ссылающейся таблицы, которая является отдельной, а не секцией, необходимы другие записи каталога. Команды `ATTACH/DETACH PARTITION` выполняли это преобразование некорректно. В частности, после выполнения команды `DETACH` для новой отдельной таблицы терялись триггеры обеспечения целостности внешнего ключа, из-за чего в таблице могли появляться строки, не выполняющие требования по ограничению внешнего ключа. Кроме того, последующее повторное присоединение командой `ATTACH` могло приводить к неожиданным ошибкам.

  Это можно исправить, выполнив команду `ALTER TABLE DROP CONSTRAINT` для новой отдельной таблицы в отношении каждого ограничения с ошибкой и добавив ограничение заново. Ошибка при повторном добавлении ограничения означает появление ошибочных данных. В этом случае необходимо заново и вручную обеспечить целостность целевой и ссылающейся таблиц, а затем ещё раз добавить ограничение.

  Используйте запрос ниже, чтобы выявить нарушенные ограничения и подготовить команды для создания таких ограничений заново:

  ```
  SELECT conrelid::pg_catalog.regclass AS "constrained table",
         conname AS constraint,
         confrelid::pg_catalog.regclass AS "references",
         pg_catalog.format('ALTER TABLE %s DROP CONSTRAINT %I;',
                           conrelid::pg_catalog.regclass, conname) AS "drop",
         pg_catalog.format('ALTER TABLE %s ADD CONSTRAINT %I %s;',
                           conrelid::pg_catalog.regclass, conname,
                           pg_catalog.pg_get_constraintdef(oid)) AS "add"
  FROM pg_catalog.pg_constraint c
  WHERE contype = 'f' AND conparentid = 0 AND
     (SELECT count(*) FROM pg_catalog.pg_constraint c2
      WHERE c2.conparentid = c.oid) <>
     ((SELECT count(*) FROM pg_catalog.pg_inherits i
      WHERE (i.inhparent = c.conrelid OR i.inhparent = c.confrelid) AND
        EXISTS (SELECT 1 FROM pg_catalog.pg_partitioned_table
                WHERE partrelid = i.inhparent)) +
      CASE WHEN pg_catalog.pg_partition_root(conrelid) = confrelid THEN
                (SELECT count(*) FROM pg_catalog.pg_partition_tree(confrelid)
                  WHERE level = 1)
           ELSE 0 END);
  ```

  Поскольку один или несколько этапов выполнения команды `ADD CONSTRAINT` могут завершиться ошибкой, сохраните вывод запроса в файл и попробуйте заново повторить каждый этап.

- Исправление теста для локали `C`, когда значения переменных `LC_COLLATE` и `LC_CTYPE` отличаются (Джефф Девис)

  При использовании `libc` как провайдера локали по умолчанию, тест, проверяющий использование локали `C` для сравнения, ранее некорректно проверял значение `LC_CTYPE`, а не `LC_COLLATE`. Это не влияло на обычный сценарий работы, когда значения совпадали или оба отличались от `C` (или псевдонима `POSIX`). Однако, если `LC_CTYPE` имела значение `C`, а в

`LC_COLLATE` была указана другая локаль, запросы могли выполняться некорректно и было возможно повреждение индексов, построенных по строкам. После установки этого обновления пользователям БД с такими значениями переменных следует перестроить затронутые проблемой индексы. Обратный случай, когда `LC_COLLATE` принимала значение `C`, а в `LC_CTYPE` указана другая локаль, приводил к снижению производительности, но не к ошибкам.

- Запрет использования соединений или группировки с учётом секционирования, если сортировка в запросе для столбца ключа не соответствует сортировке ключа секционирования (Цзянь Хи, Веббо Хан)

  Такие планы ранее могли приводить к некорректным результатам.

- Предотвращение сбоя планировщика после преобразования теста `IS NULL` для столбца `NOT NULL` в константу `FALSE` (Ричард Гуо)

  Эта проблема обычно вызывала такие ошибки, как «variable not found in subplan target lists» (переменная не найдена в целевых списках подплана).

- Предотвращение возможного сбоя планировщика во время встраивания SQL-функции, аргументы которой содержат определённые связанные с массивами конструкции (Том Лейн, Натан Боссарт)
- Исправление возможных некорректных выводов или ошибки планировщика «wrong varnullingrels» (неправильные varnullingrels) для операций `MERGE ... WHEN NOT MATCHED BY SOURCE` (Дин Рашид)
- Исправление возможной ошибки «could not find pathkey item to sort» (не удалось найти элемент ключей пути для сортировки), когда необходима сортировка вывода запроса с оператором `UNION ALL`, а столбец сортировки является выражением (Андрей Лепихов, Том Лейн)
- Исправление сканирования индекса-B-дерева для ScalarArrayOp в особых случаях (Питер Гейган)

  Вывод мог быть некорректным, когда прокручиваемые курсоры с планом такого вида прокручивались до начальной точки и снова прокручивались вперёд.

- Исправление сбоя проверочного утверждения или сбивающего с толку сообщения об ошибке выполнения команды `COPY (`*`запрос`*`) TO ...`, когда *`запрос`* переписывается правилом `DO INSTEAD NOTIFY` (Тендер Ван, Том Лейн)
- Исправление проверки параметров `FORCE_NOT_NULL` и `FORCE_NULL` команды `COPY` (Джоэл Джейкобсон)

  Некоторые некорректные варианты использования теперь запрещены, как и должно было быть.

- Исправление сбоя сервера в случаях, когда вызов `json_objectagg()` содержит изменчивую функцию (Амит Ланготе)
- Исправление выявления асимметричных данных во время параллельного соединения по хешу (Томас Манро)

  После пересекционирования внутренней стороны соединения по хешу из-за слишком большого количества накопленных кортежей в одной секции проверялось, что все кортежи секции попадали в одну и ту же дочернюю секцию. Так предполагалось, что у всех кортежей одно хеш-значение и последующее пересекционирование не может привнести улучшений. В некоторых случаях эта проверка не работала, из-за чего выполнялось бессмысленное многократное пересекционирование, что в итоге приводило к ошибке исчерпания ресурсов.

- Предотвращение сбоев, когда командой `ALTER DATABASE SET` задаётся серверный параметр, для которого поиск выполняется на основе пути, например `default_text_search_config` (Джефф Девис)
- Предотвращение повторных поисков классов операторов и сравнений во время создания нового индекса для секционированной таблицы (Том Лейн)

Исправление вызвано главным образом тем, что иногда поиск выполнялся с ограничением `search_path`, что вызывало непредвиденные сбои, когда команда `CREATE INDEX` ссылалась на объекты за пределами схемы `pg_catalog`.

Кроме того, благодаря этому исправлению комментарии к родительскому секционированному индексу больше не копируются в дочерние индексы.

- Добавление недостающей зависимости указанного в `CREATE TABLE ... USING` невстроенного метода доступа от секционированной таблицы (Микаэль Пакье)

  Удаление метода доступа должно было быть запрещено при существовании зависимой от него таблицы, но было возможно, что вызывало странное поведение. Обратите внимание, что это исправление решает проблему только для секционированных таблиц, созданных после обновления.

- Запрет использования имён локалей, содержащих не ASCII-символы (Томас Манро)

  Проблема возникала только в Windows, поскольку такие имена локалей больше нигде не используются. Основная сложность заключается в том, что неясно в какой кодировке они представлены (так как используемая кодировка определена в самой локали). Из-за этого недоразумения в последних выпусках PostgreSQL в библиотеке времени выполнения могли случаться прерывания операций.

  При появлении нового сообщения об ошибке следует либо создать новую локаль с именем только из набора ASCII при помощи программы Locale Builder в Windows, либо использовать имена локалей, соответствующие языковому тегу BCP 47, например, `tr-TR`.

- Устранение условий гонки при фиксации сериализуемой транзакции (Хейкки Линнакангас)

  Некорректная обработка недавно зафиксированной транзакции могла приводить к сбою проверочного утверждения или ошибке «could not access status of transaction» (не удалось получить состояние транзакции).

- Устранение условий гонки при выполнении команды `COMMIT PREPARED`, которые приводили к наличию потерянных файлов двухфазной фиксации (2PC) (У Чэнвэнь)

  Параллельное выполнение нескольких команд `PREPARE TRANSACTION` могло приводить к тому, что команда `COMMIT PREPARED` не удаляла файл состояния двухфазной фиксации на диске для завершённой транзакции. Мгновенно к негативным последствиям это не приводило, однако при последующем восстановлении после сбоя могла возникать ошибка «could not access status of transaction» (не удалось получить состояние транзакции), что требовало ручного удаления потерянного файла для восстановления службы.

- Предотвращение некорректного обращения к памяти после пропуска нерабочего индекса TOAST во время выполнения команды `VACUUM FULL` (Тендер Ван)

  Список, отслеживающий подлежащие перестройке индексы, некорректно обновлялся в этом месте кода, что могло приводить сбоям проверочных утверждений или отказам в будущем.

- Исправление возможных потерь обновлений каталога «на месте» (Ной Миш)

  При обычном обновлении строк новая версия строки записывается, чтобы сохранить возможность отката транзакции. Однако при обновлении некоторых системных каталогов транзакции намеренно не затрагиваются, и в таком случае строка обновляется на месте. Данное исправление устраняет условия гонки, которые могли приводить к потере обновлений на месте. Одним из примеров была потеря того, что для `pg_class.relhasindex` задано значение true, что приводило к невозможности обновления нового индекса и, следовательно, его повреждению.

- Сброс кеша каталогов в конце процесса восстановления (Ной Миш)

  Таким образом предотвращаются сценарии, при которых обновления каталога на месте могли быть потеряны из-за использования устаревших данных из кеша каталога.

- Предотвращение параллельного выполнения запросов при сдерживании прерываний (Франческо Деграсси, Ной Миш, Том Лейн)

  С этой ситуацией невозможно было столкнуться во время обычной работы, но можно было воспроизвести при помощи тестовых сценариев, например использовав SQL-функцию в качестве опорной для B-дерева (что слишком медленно для промышленной эксплуатации). При успешном воспроизведении возникала ситуация бесконечного ожидания.

- Игнорирование ещё не определённых порталов в представлении `pg_cursors` (Том Лейн)

  Определённый пользователем код, исследующий это представление, мог вызываться при настройке нового курсора. Если это случалось, происходило обращение по нулевому указателю. Проблема решена определением представления так, чтобы исключать курсоры, которые ещё не полностью настроены.

- Предотвращение ошибки «unexpected table_index_fetch_tuple call during logical decoding» (непредвиденный вызов table_index_fetch_tuple во время логического декодирования) при декодировании транзакции, содержащей вставку значения столбца по умолчанию (Такэси Идэриха, Хоу Чжицзе)

- Уменьшение потребления памяти при логическом декодировании (Масахико Савада)

  Теперь используется блок по умолчанию меньшего размера для хранения данных кортежей, получаемых во время логического декодирования. Таким образом снижается потеря памяти, которая, как сообщается, была существенной, возникала во время длительных транзакций и даже приводила к сбоям из-за нехватки памяти.

- Исправление поведения стабильных функций, вызываемых из списка аргументов оператора `CALL`, где `CALL` находится в блоке `EXCEPTION` PL/pgSQL (Том Лейн)

  Как и до исправления в предыдущем квартальном выпуске, таким функциям ранее передавался неправильный снимок, в результате чего они получали устаревшие значения строк, которые уже изменились с момента начала внешней транзакции.

- Разбор параметра подключения `keepalives` в libpq так же, как и других целочисленных параметров (Юто Сасаки)

  В используемом коде запрещались завершающие пробелы в значении параметра, в отличие от других случаев. Это оказалось проблемой, например, в использовании ecpg.

- Исправление чтения за пределами выборки при разборе некорректных даты и времени в ecpglib (Брюс Момджян, Павел Некрасов)

  Ранее было возможно попытаться считать расположение непосредственно перед началом константного массива. Однако практические следствия этой ошибки незначительны.

- Исправление команд описания psql для поддержки серверов версий до 9.4 (Том Лейн)

  Выполнение команд, при которых отображается список управления доступом (ACL), завершалось ошибкой на очень старых версиях сервера PostgreSQL из-за использования функции, отсутствующей в этих версиях.

- Предотвращение зависания в psql, если в команде `\watch` указан интервал менее 1 мс (Андрей Бородин, Микаэль Пакье)

  Вместо этого такой интервал будет обрабатываться как равный нулю (без задержки между выполнениями).

- Устранение неспособности найти пароль репликации в `~/.pgpass` (Том Лейн)

  Утилитам pg_basebackup и pg_receivewal не удавалось сопоставить вхождение в `~/.pgpass`, для которого в поле имени БД содержалось значение `replication`, если не были указаны параметры `-d` или `--dbname`. Такое поведение приводило к непредвиденному запросу пароля.

- Отображение ошибки в pg_combinebackup, если файл инкрементальной резервной копии находится в каталоге, где должна содержаться полная копия (Роберт Хаас)
- Запрет на создание имён файлов с двумя косыми чертами в pg_combinebackup (Роберт Хаас)

  Проблем в работе это не вызывало, однако в сообщениях об ошибках дублировались косые черты, что могло сбивать с толку.
- Предотвращение попытки перестройки индексов временных таблиц, индексов в vacuumdb и при параллельной перестройке индексов с помощью reindexdb (Вайбхав С, Микаэль Пакье, Фудзии Масао, Натан Боссарт)

  Перестройка индексов временных таблиц других сеансов невозможна, но проверка на пропуск отсутствовала в некоторых местах кода, что вело к нежелательным сбоям.
- Исправление некорректного кода, генерируемого LLVM, на платформах ARM64 (Томас Манро, Антонин Боннефой)

  При использовании JIT-компиляции на платформах ARM генерируемый код не поддерживал расстояния перемещения больше 32 бит, из-за чего он мог размещаться неудачно и вызывать сбои сервера в системах с большим объёмом памяти.
- Исправление нескольких случаев, в которых допускалось, что время начала процесса (представленное `time_t`) уместится в значение типа `long` (Макс Джонсон, Натан Боссарт)

  На платформах, где тип `long` вмещает 32 бита (особенно на Windows), после 2038 года произойдёт сбой. В основном он проявит себя косметически, но в особенности зависнет команда `pg_ctl start`.
- Обновление данных часовых поясов до версии tzdata 2024b (Том Лейн)

  В этом выпуске tzdata изменяются старые имена часовых поясов, совместимые с System V, теперь они дублируют соответствующие географические зоны. Например, `PST8PDT` теперь является псевдонимом для `America/Los_Angeles`. Единственное видимое последствие заключается в том, что для временных отметок до введения стандартизированных часовых поясов часовой пояс будет означать среднее солнечное время для заданного места. Например, в `PST8PDT` ввод с типом данных `timestamptz`, такой как `1801-01-01 00:00`, раньше отображался как `1801-01-01 00:00:00-08`, а теперь — как `1801-01-01 00:00:00-07:52:58`.

  Кроме того, выполнены корректировки исторических данных для Мексики, Монголии и Португалии. Часовой пояс `Asia/Choibalsan` теперь является псевдонимом для `Asia/Ulaanbaatar`, а не отдельным часовым поясом, в основном потому что разница между ними, как оказалось, основана на недостоверных данных.

E.12. Выпуск 17

Дата выпуска: 2024-09-26

E.12.1. Обзор

PostgreSQL 17 содержит много новых возможностей и улучшений, в том числе:

- Новая система управления памятью для `VACUUM`, потребляющая меньше памяти и повышающая производительность очистки.
- Расширение синтаксиса SQL/JSON, в том числе добавление функций-конструкторов, предикатов и функции `JSON_TABLE()` для преобразования данных типа JSON в табличное представление.
- Различные улучшения производительности запросов, в том числе для последовательного чтения с использованием потокового ввода-вывода, записи при большой параллельной активности и поиска по нескольким значениям в индексах btree.
- Улучшения логической репликации, в том числе:

  - Управление отработкой отказа
  - Утилита pg_createsubscriber для создания логических реплик из резервных серверов физической репликации
  - pg_upgrade теперь сохраняет слоты логической репликации на публикующих серверах и полное состояние подписки на подписчиках. Теперь после обновления на будущие основные версии логическая репликация будет продолжаться без запроса копирования для повторной синхронизации.
- Новый параметр подключения `sslnegotiation=direct` на стороне клиента, который устанавливает TLS-соединение напрямую без предварительного согласования.
- Поддержка инкрементального резервного копирования в pg_basebackup.
- Новый параметр `ON_ERROR ignore` в `COPY`, позволяющий продолжать копирование даже в случае ошибки.

Предыдущие пункты и другие новые возможности PostgreSQL 17 более подробно описаны в следующих разделах.

E.12.2. Миграция на версию 17

Тем, кто хочет перенести данные из любой предыдущей версии, необходимо выполнить выгрузку/загрузку данных с помощью pg_dumpall либо использовать pg_upgrade или логическую репликацию. Общую информацию о переходе на более новую основную версию можно найти в Разделе 17.6.

В версии 17 реализован ряд изменений, которые могут повлиять на совместимость с предыдущими выпусками. Рассмотрите следующие несовместимые аспекты:

- Использование функциями безопасного пути поиска (search_path) во время операций обслуживания (Джефф Девис)

  Это изменение помогает предотвратить небезопасный доступ при выполнении операций по обслуживанию базы данных: `ANALYZE`, `CLUSTER`, `CREATE INDEX`, `CREATE MATERIALIZED VIEW`, `REFRESH MATERIALIZED VIEW`, `REINDEX` или `VACUUM`. Если функции, используемые в индексах по выражениям или в материализованных представлениях, обращаются к объектам, которые находятся в нестандартных схемах, то теперь при создании таких функций необходимо явно указывать путь поиска.

- Ограничение использования `ago` — теперь только в конце значений типа `interval` (Джозеф Кошаков)

  Кроме того, теперь запрещается многократное использование пустых единиц измерения в значениях типа interval.

- Удаление серверной переменной `old_snapshot_threshold` (Томас Манро)

  Эта переменная позволяла процессу VACUUM удалять строки, которые потенциально были видимыми для выполняющихся транзакций, и если транзакции пытались получить к ним доступ, возникали ошибки типа «snapshot too old» (снимок слишком стар). Если найдётся более удачная реализация, функциональность может быть добавлена обратно в PostgreSQL в будущем.

- Изменение обработки статуса суперпользователя для начального пользователя сеанса командой `SET SESSION AUTHORIZATION` (Джозеф Кошаков)

  Теперь поведение команды зависит от наличия статуса суперпользователя у пользователя в момент выполнения команды `SET SESSION AUTHORIZATION`, а не на момент подключения к базе данных.

- Удаление функциональности симуляции поддержки пользователей на уровне отдельных БД (Натан Боссарт)

Эта функциональность (`db_user_namespace`) использовалась редко.

- Удаление дополнительного модуля adminpack (Даниэль Густафссон)

  Ранее этот модуль использовался инструментом pgAdmin III, поддержка которого закончилась.

- Удаление значения `fsync_writethrough` для wal_sync_method в Windows (Томас Манро)

  Это значение было эквивалентно `fsync` в Windows.

- Изменение в обработке границ файлов двумя функциями, связанными с именами файлов WAL (Кётаро Хоригути, Андрес Фройнд, Брюс Момджян)

  Раньше функции `pg_walfile_name()` и `pg_walfile_name_offset()` возвращали номер предыдущего сегмента LSN, когда LSN был на границе сегмента файла; теперь они возвращают текущий сегмент LSN.

- Удаление ставшей ненужной серверной переменной `trace_recovery_messages` (Бхарат Рупиредди)
- Удаление столбца `element_types.domain_default` из информационной схемы (Питер Эйзентраут)
- Изменение меток для вывода режимов блокирования в модуле pgrowlocks (Брюс Момджян)
- Удаление `buffers_backend` и `buffers_backend_fsync` из `pg_stat_bgwriter` (Бхарат Рупиредди)

  Эти столбцы считаются избыточными по отношению к аналогичным столбцам в представлении `pg_stat_io`.

- Переименование столбцов статистики времени ввода-вывода, связанного с чтением и записью блоков данных, в представлении pg_stat_statements (Назир Билал Явуз)

  Столбец `blk_read_time` теперь называется `shared_blk_read_time`, а `blk_write_time` — `shared_blk_write_time`.

- Отображение цели сбора статистики по умолчанию как `NULL` в столбцах `pg_attribute.attstattarget` и `pg_statistic_ext.stxstattarget` (Питер Эйзентраут)
- Переименование `pg_collation.colliculocale` в `colllocale` и `pg_database.daticulocale` в `datlocale` (Джефф Девис)
- Переименование столбцов `max_dead_tuples` в `max_dead_tuple_bytes`, `num_dead_tuples` в `num_dead_item_ids` и добавление `dead_tuple_bytes` в представление `pg_stat_progress_vacuum` (Масахико Савада)
- Переименование столбцов SLRU в системном представлении `pg_stat_slru` (Альваро Эррера)

  Кроме того, изменились и имена столбцов, которые принимает функция `pg_stat_reset_slru()`.

E.12.3. Изменения

Ниже вы найдёте подробный список изменений, произошедших между предыдущим основным выпуском и выпуском PostgreSQL 17.

E.12.3.1. Сервер

E.12.3.1.1. Оптимизатор

- Возможность для оптимизатора улучшать планы с CTE с учётом статистики и порядка сортировки столбцов, из которых выводятся строки в ранних частях запроса (Цзянь Гуо, Ричард Гуо, Том Лейн)
- Улучшение оптимизации запросов с условиями `IS NOT NULL` и `IS NULL` (Дэвид Роули, Ричард Гуо, Энди Фан)

Теперь оптимизатор удаляет ограничение `IS NOT NULL` для запросов по столбцам `NOT NULL` и исключает сканирование по столбцам `NOT NULL` при использовании условия `IS NULL`.

- Возможность отсекать секции при использовании условий `IS [NOT] UNKNOWN` для логических столбцов (Дэвид Роули)
- Улучшение оптимизации запросов с использованием операторов включения в диапазон <@ и @> (Ким Йохан Андерссон, Цзянь Хи)
- Возможность преобразования коррелирующих подзапросов `IN` в JOIN (Энди Фан, Том Лейн)
- Улучшение оптимизации запросов с предложением `LIMIT` для секционированных таблиц, таблиц с наследованием и для `UNION ALL` (Энди Фан, Дэвид Роули)
- Увеличение числа случаев, в которых можно запускать узлы запросов в параллельном режиме (Том Лейн)
- Возможность упорядочивать столбцы `GROUP BY`, чтобы порядок совпадал с `ORDER BY` (Андрей Лепихов, Фёдор Сигаев)

  Это поведение отключается серверной переменной enable_group_by_reordering.
- Возможность запросов с `UNION` (без `ALL`) использовать MergeAppend (Дэвид Роули)
- Корректировка вычисления количества сортируемых строк в планах MergeAppend (Александр Кузьменков)
- Использование индексов GiST и SP-GiST для инкрементальной сортировки (Мирослав Бендик)

  Это улучшение особенно полезно в запросах с `ORDER BY`, когда первый столбец имеет индекс GiST или SP-GiST, а остальные — нет.
- Добавление столбцов в представление `pg_stats` для вывода информации о гистограммах для диапазонных типов данных (Егор Рогов, Соумйадип Чакраборти)

E.12.3.1.2. Индексы

- Более эффективная обработка индексами btree запросов множества значений, таких как значения, указанные в предложениях `IN` с использованием констант (Питер Гейган, Маттиас ван де Меент)
- Создание индексов BRIN с использованием параллельных исполнителей (Томаш Вондра, Маттиас ван де Меент)

E.12.3.1.3. Общая производительность

- Более эффективное удаление и заморозка кортежей в процессе `VACUUM` (Мелани Плейгман, Хейкки Линнакангас)

  Теперь в процессе очистки генерируется меньше трафика WAL.
- Более эффективное хранение ссылок на кортежи процессом очистки (Масахико Савада, Джон Нейлор)

  Кроме того, для процесса очистки удалено скрытое ограничение памяти в 1 ГБ, ранее действовавшее даже при повышенных значениях maintenance_work_mem или autovacuum_work_mem.
- Оптимизация очистки отношений без индексов (Мелани Плейгман)
- Увеличение значения по умолчанию до 2 МБ для vacuum_buffer_usage_limit (Томас Манро)
- Улучшение производительности при проверке ролей с членством во множестве ролей (Натан Боссарт)
- Улучшение производительности записи в WAL в условиях высокой конкуренции за ресурсы (Бхарат Рупиредди)
- Улучшение производительности при передаче клиенту больших блоков данных (Мелих Мутлу)

- Возможность объединять операции чтения данных файловой системы с помощью io_combine_limit (Томас Манро, Андрес Фройнд, Мелани Плейгман, Назир Билал Явуз)

E.12.3.1.4. Мониторинг

- Новое системное представление `pg_stat_checkpointer` (Бхарат Рупиреддди, Антон Мельников, Александр Коротков)

  В это представление были перенесены некоторые столбцы из `pg_stat_bgwriter`.
- Улучшение управления сбросом статистики (Атсуши Торикоши, Бхарат Рупиреддди)

  Теперь функции `pg_stat_reset_shared()` (без аргументов) и `pg_stat_reset_shared(NULL)` могут сбрасывать всю общую статистику, а функции `pg_stat_reset_shared('slru')` и `pg_stat_reset_slru()` (без аргументов) — сбрасывать статистику SLRU-кеша, что было возможно и ранее с помощью `pg_stat_reset_slru(NULL)`.
- Добавление в журнал сообщений, связанных с восстановлением WAL из резервных копий (Андрес Фройнд)
- Добавление строки журнала log_connections для подключений по методу `trust` (Джейкоб Чемпион)
- Добавление сообщений журнала о получении и освобождении слотов репликации процессами walsender (Бхарат Рупиреддди)

  Чтобы включить запись таких сообщений, используется серверная переменная log_replication_commands.
- Добавление системного представления `pg_wait_events`, выводящего информацию о типах событий ожидания (Бертран Друво)

  Оно предоставляет описание событий ожидания, которые отображаются в `pg_stat_activity`.
- Добавление событий ожидания для задержек контрольных точек (Томас Манро)
- Возможность отслеживать обработку индексов в процессе очистки (Сами Имсейх)

  Эту информацию можно найти в столбцах `indexes_total` и `indexes_processed` системного представления `pg_stat_progress_vacuum`.

E.12.3.1.5. Права

- Предоставление права на выполнение операций по обслуживанию БД (Натан Боссарт)

  Для этого можно выдать право `MAINTAIN` на уровне таблицы или предоставить членство в предопределённой роли `pg_maintain`, в результате чего будут доступны следующие операции: `VACUUM`, `ANALYZE`, `REINDEX`, `REFRESH MATERIALIZED VIEW`, `CLUSTER` и `LOCK TABLE`.
- Возможность для ролей с членством в роли `pg_monitor` выполнять `pg_current_logfile()` (Павло Голуб, Натан Боссарт)

E.12.3.1.6. Конфигурация сервера

- Добавление системной переменной allow_alter_system для отключения возможности использовать `ALTER SYSTEM` (Йелте Феннема-Нио, Габриэль Бартолини)
- Возможность использовать `ALTER SYSTEM` для установки нераспознанных пользовательских серверных переменных (Том Лейн)

  Это также можно делать с помощью `GRANT ON PARAMETER`.
- Добавление серверной переменной transaction_timeout, ограничивающей длительность транзакций (Андрей Бородин, Япинь Ли, Цзюньван Жао, Александр Коротков)
- Добавление платформонезависимого провайдера правил сортировки builtin (Джефф Девис)

  Он поддерживает правила сортировки `C` и `C.UTF-8`.

- Добавление серверной переменной huge_pages_status, выводящей информацию об использовании огромных страниц (Джастин Призби)

  Используется, когда для huge_pages задано значение `try`.

- Добавление серверной переменной, отключающей событийные триггеры (Даниэль Густафссон)

  Переменная event_triggers позволяет временно отключать событийные триггеры для отладки.

- Возможность регулировать размеры SLRU-кешей (Андрей Бородин, Дилип Кумар, Альваро Эррера)

  Добавлены серверные переменные commit_timestamp_buffers, multixact_member_buffers, multixact_offset_buffers, notify_buffers, serializable_buffers, subtransaction_buffers и transaction_buffers. commit_timestamp_buffers, transaction_buffers и subtransaction_buffers автоматически масштабируются в зависимости от значения shared_buffers.

E.12.3.1.7. Потоковая репликация и восстановление

- Поддержка инкрементального резервного копирования файловой системы (Роберт Хаас, Якуб Вартак, Томаш Вондра)

  Для создания инкрементальных резервных копий используется новый ключ pg_basebackup, `--incremental`. Новое приложение pg_combinebackup позволяет работать с базовыми и инкрементальными резервными копиями системы.

- Возможность создания файлов сводок WAL (Роберт Хаас, Натан Боссарт, Хуберт Любачевски)

  Эти файлы содержат информацию о номерах блоков, изменённых в определённом диапазоне LSN, которые используются при инкрементальном резервном копировании. Для настройки функциональности добавлены серверные переменные summarize_wal и wal_summary_keep_time, а для просмотра информации о файлах — функции `pg_available_wal_summaries()`, `pg_wal_summary_contents()` и `pg_get_wal_summarizer_state()`.

- Добавление системного идентификатора в файлы манифестов копий файловой системы (Амул Сул)

  Это полезно для обнаружения попыток использования неверных журналов WAL.

- Указание значения параметра строки подключения `dbname` при записи информации о подключении в файл `postgresql.auto.conf` программой pg_basebackup (Вигнеш Си, Хайато Курода)

- Добавление столбца `pg_replication_slots.invalidation_reason` для отображения причины аннулирования слотов (Швета Малик, Бхарат Рупиредди)

- Добавление столбца `pg_replication_slots.inactive_since` для отображения время неактивности слотов (Бхарат Рупиредди)

- Добавление функции `pg_sync_replication_slots()` для синхронизации слотов логической репликации (Хоу Чжицзе, Швета Малик, Аджин Чериан, Питер Эйзентраут)

- Добавление свойства `failover` в протокол репликации (Хоу Чжицзе, Швета Малик)

E.12.3.1.8. Логическая репликация

- Добавление приложения pg_createsubscriber для создания логической реплики из резервного сервера физической репликации (Эйлер Тавейра)

- Возможность миграции рабочих слотов и подписок логической репликации pg_upgrade (Хайато Курода, Хоу Чжицзе, Вигнеш Си, Жюльен Руо, Шлок Кьял)

  Теперь репликация может продолжаться сразу после миграции; поддерживается для кластеров PostgreSQL, начиная с версии 17.

- Автоматическое переключение слотов логической репликации при обработке отказа (Хоу Чжицзе, Швета Малик, Аджин Чериан)

  Для включения функциональности предназначен необязательный пятый аргумент функции `pg_create_logical_replication_slot()`.
- Добавление серверной переменной sync_replication_slots, включающей синхронизацию слотов логической репликации при автоматическом переключении для отработки отказа (Швета Малик, Хоу Чжицзе, Питер Смит)
- Возможность управления поведением при обработке отказа логической репликации в командах `CREATE/ALTER SUBSCRIPTION` (Швета Малик, Хоу Чжицзе, Аджин Чериан)
- Возможность использования хеш-индексов на стороне подписчика для применения изменений логической репликации (Хайато Курода)

  Ранее для этой цели могли использоваться только индексы btree.
- Улучшение производительности логического декодирования для случаев с множеством подтранзакций (Масахико Савада)
- Реализация перезапуска рабочих процессов, применяющих изменения логической репликации, если у владельца подписки отозваны права суперпользователя (Вигнеш Си)

  При этом требуется повторная аутентификация.
- Добавление параметра `flush` в функцию `pg_logical_emit_message()` (Микаэль Пакье)

  С этим параметром сообщения гарантированно сохраняются.
- Указание физических резервных серверов, которые должны быть синхронизированы перед тем, как данные станут видимыми для подписчиков (Хоу Чжицзе, Швета Малик)

  Серверы указываются в новой переменной synchronized_standby_slots.
- Добавление столбца с типом рабочего процесса в `pg_stat_subscription` (Питер Смит)

E.12.3.2. Служебные команды

- Добавление нового параметра `ON_ERROR ignore` в `COPY`, позволяющего игнорировать строки с ошибками (Дамир Белялов, Атсуши Торикоши, Александр Шульгин, Цзянь Хи, Юго Нагата)

  По умолчанию поведение осталось прежним: `ON_ERROR stop`.
- Добавление нового параметра `LOG_VERBOSITY` в `COPY` для вывода игнорируемых строк с ошибками `COPY FROM` (Бхарат Рупиредди)
- Вывод количества пропущенных строк при обработке командой `COPY FROM` (Атсуши Торикоши)

  Эта информация выводится в столбце системного представления `pg_stat_progress_copy.tuples_skipped`.
- Возможность указывать, что все значения в столбцах в команде `COPY FROM` должны быть принудительно установлены в NULL или NOT NULL (Чжан Минли)
- Поддержка столбцов идентификации для секционированных таблиц (Ашутош Бапат)
- Поддержка ограничений-исключений для секционированных таблиц (Пол А. Юнгвирт)

  Чтобы ограничения-исключения работали корректно для секционированных таблиц, они должны сравнивать части ключей секционирования на равенство, при этом другие столбцы могут использовать специфичные для ограничений-исключений сравнения.
- Добавление более понятного метода использования `ALTER TABLE` для указания столбца ориентира статистики по умолчанию (Питер Эйзентраут)

  Добавлен синтаксис `ALTER TABLE ... SET STATISTICS DEFAULT`; устаревший синтаксис `SET STATISTICS -1` всё ещё поддерживается.

- Возможность изменять выражение для генерируемых столбцов с помощью `ALTER TABLE` (Амул Сул)

  Для этого используется команда `ALTER TABLE ... ALTER COLUMN ... SET EXPRESSION`.
- Возможность указывать метод доступа для секционированных таблиц (Джастин Призби, Соумйадип Чакраборти, Микаэль Пакье)
- Добавление значения `DEFAULT` для `ALTER TABLE .. SET ACCESS METHOD` (Микаэль Пакье)
- Поддержка событийных триггеров, срабатывающих при установлении соединения с БД (Константин Книжник, Михаил Грибков)
- Поддержка событийных триггеров для `REINDEX` (Гаррет Торнбург, Цзянь Хи)
- Поддержка использования синтаксиса в скобках для параметров `CLUSTER`, если имя таблицы не указано (Натан Боссарт)

E.12.3.2.1. `EXPLAIN`

- Возможность выводить информацию об использовании памяти оптимизатором в `EXPLAIN` (Ашутош Бапат)

  Вывести эту информацию можно при помощи параметра `MEMORY`.
- Добавление в `EXPLAIN` параметра `SERIALIZE` для вывода стоимости преобразования данных для передачи по сети (Степан Руц, Маттиас ван де Меент)
- Возможность для `EXPLAIN` отображать статистику времени чтения и записи локальных блоков в выводе с параметром `BUFFERS` (Назир Билал Явуз)
- Улучшение отображения узлов SubPlan и выходных параметров в `EXPLAIN` (Том Лейн, Дин Рашид)
- Отображение информации `deform_counter` при использовании JIT-компиляции в выводе `EXPLAIN` (Дмитрий Долгов)

E.12.3.3. Типы данных

- Поддержка значений `+/-infinity` в типе данных `interval` (Джозеф Кошаков, Цзянь Хи, Ашутош Бапат)
- Возможность использовать тип `ENUM`, добавленный посредством `ALTER TYPE`, если тип был создан в той же транзакции (Том Лейн)

  Ранее такой сценарий не допускался.

E.12.3.4. MERGE

- Возможность модифицировать изменяемые представления командой `MERGE` (Дин Рашид)
- Добавление `WHEN NOT MATCHED BY SOURCE` в `MERGE` (Дин Рашид)

  Условие `WHEN NOT MATCHED` и ранее поддерживалось для целевых строк.
- Возможность использовать предложение `RETURNING` в `MERGE` (Дин Рашид)

  Новая функция `merge_action()` в `RETURNING` позволяет получить информацию о DML-операции для каждой строки.

E.12.3.5. Функции

- Добавление функции `JSON_TABLE()` для преобразования данных типа `JSON` в табличное представление (Никита Глухов, Фёдор Сигаев, Олег Бартунов, Александр Коротков, Эндрю Дунстан, Амит Ланготе, Цзянь Хи)

  Эта функция может использоваться в предложении `FROM` запросов `SELECT` в качестве источника кортежей.

- Добавление функций-конструкторов SQL/JSON: `JSON()`, `JSON_SCALAR()` и `JSON_SERIALIZE()` (Никита Глухов, Фёдор Сигаев, Олег Бартунов, Александр Коротков, Эндрю Дунстан, Амит Ланготе)
- Добавление функций запросов SQL/JSON: `JSON_EXISTS()`, `JSON_QUERY()` и `JSON_VALUE()` (Никита Глухов, Фёдор Сигаев, Олег Бартунов, Александр Коротков, Эндрю Дунстан, Амит Ланготе, Питер Эйзентраут, Цзянь Хи)
- Добавление методов jsonpath для преобразования значений типа `JSON` в другие типы `JSON` (Дживан Чок)

  Добавлены следующие методы jsonpath: `.bigint()`, `.boolean()`, `.date()`, `.decimal([precision [, scale]])`, `.integer()`, `.number()`, `.string()`, `.time()`, `.time_tz()`, `.timestamp()` и `.timestamp_tz()`.
- Добавление спецификаторов формата часового пояса для `to_timestamp()` (Том Лейн)

  `TZ` принимает аббревиатуры часовых поясов или числовые смещения, а `OF` принимает только числовые смещения.
- Возможность указывать часовой пояс сеанса как `AS LOCAL` (Вик Фиринг)

  Это полезно для преобразования временных меток с добавлением или удалением информации о часовом поясе, без явного указания часового пояса в формате строки.
- Добавление функций `uuid_extract_timestamp()` и `uuid_extract_version()` для работы с UUID (Андрей Бородин)
- Добавление функций, генерирующих случайные числа в заданном диапазоне (Дин Рашид)

  Добавлены функции `random(min, max)`, принимающие значения `integer`, `bigint` и `numeric`.
- Добавление функций для преобразования целых чисел в строковые представления в двоичной и восьмеричной системах (Эрик Радман, Натан Боссарт)

  Добавлены функции `to_bin()` и `to_oct()`.
- Добавление функций, связанных с информацией о Unicode (Джефф Девис)

  Функция `unicode_version()` возвращает версию Unicode, `icu_unicode_version()` возвращает версию ICU, а `unicode_assigned()` проверяет, назначены ли символам коды Unicode.
- Добавление функции `xmltext()`, преобразующей текст в отдельный текстовый узел `XML` (Джим Джонс)
- Добавление функции `to_regtypemod()`, возвращающей модификатор типа для указанной спецификации типа (Дэвид Уилер, Эрик Винхольд)
- Добавление функции `pg_basetype()`, возвращающей базовый тип домена (Стив Чавес)
- Добавление функции `pg_column_toast_chunk_id()`, возвращающей идентификатор TOAST для значения (Юго Нагата)

  Если значение хранится не в виде TOAST, функция возвращает `NULL`.

E.12.3.6. PL/pgSQL

- Возможность в PL/pgSQL использовать спецификации `%TYPE` и `%ROWTYPE` для представления массивов типов, которые не являются массивами (Цюань Цзунлян, Павел Стехуле)
- Возможность в PL/pgSQL использовать спецификацию `%TYPE` для ссылок на столбцы составного типа (Том Лейн)

E.12.3.7. libpq

- Добавление функции для изменения паролей ролям в libpq (Джо Конвей)

Добавлена функция `PQchangePassword()`, которая хеширует новый пароль перед отправкой на сервер.

- Добавление функций закрытия порталов и подготовленных операторов (Йелте Феннема-Нио)

  Добавлены следующие функции: `PQclosePrepared()`, `PQclosePortal()`, `PQsendClosePrepared()` и `PQsendClosePortal()`.

- Добавление в libpq API, который позволяет отправлять как блокирующие, так и неблокирующие запросы на отмену с возможностью шифрования, если оно уже используется в соединении (Йелте Феннема-Нио)

  Ранее поддерживались только блокирующие запросы на отмену без шифрования.

- Добавление в libpq функции `PQsocketPoll()`, предназначенной для опроса состояния сетевого сокета (Тристан Партин, Том Лейн)
- Добавление в libpq функции `PQsendPipelineSync()`, предназначенной для отправки точки синхронизации в режиме конвейера (Антон Кирилов)

  Эта функция подобна `PQpipelineSync()`, но она позволяет отложить передачу данных на сервер до достижения лимита размера выходного буфера.

- Добавление в libpq функции `PQsetChunkedRowsMode()` для получения результатов запроса порциями (Даниэль Верите)§
- Возможность устанавливать TLS-соединения без предварительного согласования (Грег Старк, Хейкки Линнакангас, Питер Эйзентраут, Микаэль Пакье, Даниэль Густафссон)

  Чтобы включить эту функциональность, используется параметр `sslnegotiation=direct` на стороне клиента, при этом требуется поддержка ALPN; поддерживается только для серверов PostgreSQL, начиная с версии 17.

- Более наглядное отображение пустых прав и прав по умолчанию в выводе psql (Эрик Винхольд, Лауренц Альбе)

  Команда `\dp` теперь выводит `(none)` для пустых прав; права по умолчанию по-прежнему отображаются как пустые.

- Корректная обработка указания `\pset null` командами с обратной косой чертой (Эрик Винхольд, Лауренц Альбе)

  Ранее в таких командах указание `\pset null` игнорировалось.

- Возможность остановить команду psql `\watch` после получения минимального количества строк (Грег Сабино Маллейн)

  Для этого используется параметр `min_rows`.

- Возможность прерывать попытки подключения psql через CTRL+C (Тристан Партин)
- Поддержка `FETCH_COUNT` для команд, отличных от `SELECT`, в psql (Даниэль Верите)
- Улучшение дополнения табуляцией в psql (Дагфинн Ильмари Маннсакер, Жиль Даролд, Кристоф Хейсс, Стив Чавес, Вигнеш Си, Павел Борисов, Цзянь Хи)

E.12.3.9. Серверные приложения

- Добавление приложения pg_walsummary для выгрузки файлов сводок WAL (Роберт Хаас)
- Возможность восстанавливать большие объекты pg_dump порциями (Том Лейн)

  Если в базе данных много больших объектов, это нововведение поможет избежать ограничения на размер транзакции. Кроме того, восстановление можно выполнять параллельно.

- Добавление в pg_dump ключа `--exclude-extension` (Аюш Ватса)

- Поддержка включения/исключения объектов через файл приложениями pg_dump, pg_dumpall и pg_restore (Павел Стехуле, Даниэль Густафссон)

  Для этого используется ключ `--filter`.
- Добавление ключа `--sync-method` в несколько клиентских приложений (Джастин Призби, Натан Боссарт)

  Параметр добавлен в следующие приложения: initdb, pg_basebackup, pg_checksums, pg_dump, pg_rewind и pg_upgrade.
- Добавление в pg_restore ключа `--transaction-size` для восстановления объекта в нескольких транзакциях (Том Лейн)

  Это нововведение позволяет сохранить преимущества работы с порциями транзакций, одновременно предотвращая проблемы, связанные с большими блоками транзакций.
- Изменение параметра режима отладки в pgbench с `-d` на `--debug` (Грег Сабино Маллейн)

  Ключ `-d` теперь используется для указания имени базы данных, для этого же предназначен и новый ключ `--dbname`.
- Добавление в pgbench ключа `--exit-on-abort` для завершения работы при прерывании работы любого клиента (Юго Нагата)
- Добавление в pgbench команды `\syncpipeline`, позволяющей отправлять сообщения синхронизации (Антонин Боннефой)
- Поддержка удаления файлов истории резервного копирования для pg_archivecleanup (Атсуши Торикоши)

  Для этого используется ключ `--clean-backup-history`.
- Добавление нескольких длинных ключей в pg_archivecleanup (Атсуши Торикоши)

  Добавлены следующие ключи: `--debug`, `--dry-run` и `--strip-extension`.
- Возможность для pg_basebackup и pg_receivewal использовать имя базы данных в спецификации подключения (Йелте Феннема-Нио)

  Это нововведение особенно полезно для пулов соединений, учитывающих имена баз данных.
- Добавление в pg_upgrade ключа `--copy-file-range` (Томас Манро)

  Ключ поддерживается в ОС Linux и FreeBSD.
- Возможность для reindexdb с ключом `--index` параллельно обрабатывать индексы разных таблиц (Максим Орлов, Светлана Деревянко, Александр Коротков)
- Возможность для reindexdb, vacuumdb и clusterdb обрабатывать объекты, соответствующие шаблону, во всех базах данных (Натан Боссарт)

  Для этого предназначен новый ключ `--all`.

E.12.3.10. Исходный код

- Прекращение поддержки OpenSSL 1.0.1 (Микаэль Пакье)
- Прохождение тестов в режиме OpenSSL FIPS (Питер Эйзентраут)
- Использование инструкций CPU AVX-512 для подсчёта битов (Пол Амонсон, Натан Боссарт, Антс Аасма)
- Требование версии LLVM 10 или выше (Томас Манро)
- Использование внутренних инструкций CRC для 64-битных процессоров LoongArch (Сюйдун Ян)

- Прекращение поддержки AIX (Хейкки Линнакангас)
- Удаление параметра сборки PostgreSQL, специфичного для Microsoft Visual Studio (Микаэль Пакье)

  Теперь в среде Visual Studio возможна сборка только с Meson.
- Удаление параметра configure `--disable-thread-safety` (Томас Манро, Хейкки Линнакангас)

  Предполагается, что на всех платформах есть необходимая поддержка потоков.
- Удаление параметра configure `--with-CC` (Хейкки Линнакангас)

  Теперь указать компилятор можно только в переменной среды `CC`.
- Исключение получения данных с завершающим нулём функциями для пользовательских типов данных (Дэвид Роули)
- Добавление инкрементального анализатора `JSON` для работы с огромными документами `JSON` (Эндрю Дунстан)
- Преобразование верхнеуровневого файла `README` в формат Markdown (Натан Боссарт)
- Удаление ставшего ненужным верхнеуровневого файла `INSTALL` (Том Лейн)
- Удаление параметра make `distprep` (Питер Эйзентраут)
- Добавление поддержки make для разделяемых библиотек Android (Питер Эйзентраут)
- Добавление внутренней поддержки точек внедрения (Микаэль Пакье)

  Точки внедрения используются для отладки сервера; их поддержка должна быть включена во время компиляции.
- Добавление реестра сегментов динамической разделяемой памяти (Натан Боссарт)

  Теперь разделяемые библиотеки, не загруженные при запуске сервера, могут координировать использование динамической разделяемой памяти.
- Исправление `emit_log_hook` так, чтобы этот обработчик использовал то же значение времени, что и другие записи журнала для того же запроса (Камбам Винай, Микаэль Пакье)
- Улучшение документации по использованию `jsonpath` для проверок предикатов (Дэвид Уилер)

E.12.3.11. Дополнительные модули

- Возможность проталкивать запросы JOIN с условиями, не связанными с объединением, на сторонние серверы и на нестандартное сканирование (Ричард Гуо, Эцуро Фудзита)

  Обёртки сторонних данных и методы нестандартного сканирования необходимо адаптировать для поддержки этого изменения.
- Возможность проталкивать подзапросы `EXISTS` и `IN` на сторонние серверы postgres_fdw (Александр Пыхалов)
- Увеличение значения по умолчанию для стоимости кортежей от обёрток сторонних данных (Дэвид Роули, Умайр Шахид)

  Это значение используется оптимизатором.
- Возможность прерывать операции, выполняемые с помощью dblink (Ной Миш)
- Возможность создания хеш-индексов по столбцам ltree (Томми Павличек)

  Теперь также возможно соединение и агрегирование по хешу по столбцам ltree.
- Возможность включать пробелы и кавычки в правила преобразования символов в unaccent (Микаэль Пакье)

Изменён синтаксис файла `unaccent.rules`.

- Возможность для amcheck проверять нарушения уникальных ограничений с помощью нового ключа `--checkunique` (Анастасия Лубенникова, Павел Борисов, Максим Орлов)
- Прохождение тестов citext в режиме OpenSSL FIPS (Питер Эйзентраут)
- Прохождение тестов pgcrypto в режиме OpenSSL FIPS (Питер Эйзентраут)
- Удаление некоторых неиспользуемых макросов SPI (Бхарат Рупиредди)
- Возможность для `ALTER OPERATOR` изменять дополнительные атрибуты оптимизации (Томми Павличек)

  Это обновление полезно для разработки расширений.
- Возможность для расширений определять пользовательские события ожидания (Масахиро Икеда)

  Эта возможность была добавлена для postgres_fdw и dblink.
- Добавление в pg_buffercache функции `pg_buffercache_evict()`, позволяющей вытеснять страницы из общих буферов (Палак Чатурведи, Томас Манро)

  Это полезно при тестировании.

E.12.3.11.1. pg_stat_statements

- Замена параметров `CALL` в pg_stat_statements местозаполнителями (Сами Имсейх)
- Замена имён точек сохранения, хранящихся в `pg_stat_statements`, местозаполнителями (Грег Сабино Маллейн)

  Это значительно сокращает объём данных, записываемых командами `SAVEPOINT`, `RELEASE SAVEPOINT` и `ROLLBACK TO SAVEPOINT`.
- Замена GID двухфазной фиксации, хранящихся в `pg_stat_statements`, местозаполнителями (Микаэль Пакье)

  Это значительно сокращает объём данных, записываемых командами `PREPARE TRANSACTION`, `COMMIT PREPARED` и `ROLLBACK PREPARED`.
- Отслеживание команды `DEALLOCATE` в `pg_stat_statements` (Дагфинн Ильмари Маннсакер, Микаэль Пакье)

  Имена, используемые в `DEALLOCATE`, хранятся в `pg_stat_statements` в виде местозаполнителей.
- Добавление в `pg_stat_statements` столбцов со статистикой времени чтения и записи локальных блоков (Назир Билал Явуз)

  Добавлены столбцы `local_blk_read_time` и `local_blk_write_time`.
- Добавление информации `deform_counter` при использовании JIT-компиляции в `pg_stat_statements` (Дмитрий Долгов)
- Добавление необязательного четвёртого аргумента (`minmax_only`) в функцию `pg_stat_statements_reset()`, который позволяет сбросить только статистику максимальных/минимальных значений (Андрей Зубков)

  По умолчанию значение этого аргумента — `false`.
- Добавление в `pg_stat_statements` столбцов `stats_since` и `minmax_stats_since`, в которых отслеживается время создания записи и время последнего сброса минимальных/максимальных значений (Андрей Зубков)

E.12.4. Благодарственный список

Перечисленные ниже (в алфавитном порядке) лица сделали вклад в этот выпуск, разрабатывая, совершенствуя и рецензируя код, принимая правки, проводя тестирование или сообщая о проблемах.

Абхиджит Менон-Сен (Abhijit Menon-Sen)
Аджин Чериан (Ajin Cherian)
Аднан Даутович (Adnan Dautovic)
Айдар Имамов (Aidar Imamov)
Айзек Морленд (Isaac Morland)
Акаш Шанкаран (Akash Shankaran)
Акшат Джаймини (Akshat Jaimini)
Алаа Аттия (Alaa Attya)
Алекс Ворк (Alex Work)
Алекс Малек (Alex Malek)
Алекс Ше (Alex Hsieh)
Александр Алексеев (Aleksander Alekseev)
Александр Кожемякин (Alexander Kozhemyakin)
Александр Коротков (Alexander Korotkov)
Александр Кузьменков (Alexander Kuzmenkov)
Александр Лахин (Alexander Lakhin)
Александр Пыхалов (Alexander Pyhalov)
Александр Шульгин (Alex Shulgin)
Алексей Орлов (Aleksej Orlov)
Алексей Палажченко (Alexey Palazhchenko)
Альваро Эррера (Álvaro Herrera)
Альфонс Кемпер (Alfons Kemper)
Алёна Рыбакина (Alena Rybakina)
Амадео Гальярдо (Amadeo Gallardo)
Амит Капила (Amit Kapila)
Амит Ланготе (Amit Langote)
Амул Сул (Amul Sul)
Анастасия Лубенникова (Anastasia Lubennikova)
Анатолий Зарецкий (Anatoly Zaretsky)
Андреас Карлссон (Andreas Karlsson)
Андреас Ульбрих (Andreas Ulbrich)
Андрей Билле (Andrew Bille)
Андрей Бородин (Andrey Borodin)
Андрей Зубков (Andrei Zubkov)
Андрей Лепихов (Andrey Lepikhov)
Андрей Рачицкий (Andrey Rachitskiy)
Андрей Соколов (Andrey Sokolov)
Андрес Фройнд (Andres Freund)
Антон Волошин (Anton Voloshin)
Антон Кирилов (Anton Kirilov)
Антон Мельников (Anton Melnikov)
Антонин Боннефой
Антонин Хоуска (Antonin Houska)
Антс Аасма (Ants Aasma)
Антти Лампинен (Antti Lampinen)
Арамаки Зяке (Aramaki Zyake)
Артем Анисимов (Artem Anisimov)
Артур Закиров (Artur Zakirov)
Атсуши Торикоши (Atsushi Torikoshi)
Аттила Гуляш (Attila Gulyás)
Ашутош Бапат (Ashutosh Bapat)
Ашутош Шарма (Ashutosh Sharma)

Аюш Ватса (Ayush Vatsa)
Аюш Тивари (Ayush Tiwari)
Бартош Хрол (Bartosz Chrol)
Бенуа Райдер (Benoît Ryder)
Бернд Хелмле (Bernd Helmle)
Бертран Друво (Bertrand Drouvot)
Бо Андресон (Bo Andreson)
Боуэн Ши (Bowen Shi)
Бошоми Феникс (Boshomi Phenix)
Боюй Ян (Boyu Yang)
Брюс Момджян (Bruce Momjian)
Бхарат Рупиредди (Bharath Rupireddy)
Валери Вулард (Valerie Woolard)
Валлимахараджан Г (Vallimaharajan G)
Вася Бойцов (Vasya Boytsov)
Вигнеш Си (Vignesh C)
Видуши Гупта (Vidushi Gupta)
Вик Фиринг (Vik Fearing)
Виктор Вагнер (Victor Wagner)
Виктор Егоров (Victor Yegorov)
Виктор Лейс (Viktor Leis)
Виктория Шепард (Victoria Shepard)
Вилл Мортенсен (Will Mortensen)
Вилли Манн (Willi Mann)
Винаяк Покале (Vinayak Pokale)
Виталий Буровой (Vitaly Burovoy)
Войтех Бенеш (Vojtech Benes)
Вольфганг Вальтер (Wolfgang Walther)
Вэй Ван (Wei Wang)
Вэй Сунь (Wei Sun)
Вэньцзян Чжан (Wenjiang Zhang)
Габриэль Бартолини (Gabriele Bartolini)
Гаррет Торнбург (Garrett Thornburg)
Грант Гричан (Grant Gryczan)
Грег Нанкарроу (Greg Nancarrow)
Грег Сабино Маллейн (Greg Sabino Mullane)
Грег Старк (Greg Stark)
Гуржит Сингх (Gurjeet Singh)
Гэвин Панелла (Gavin Panella)
Гэвин Флауэр (Gavin Flower)
Давид Гайер (David Geier)
Даг Лем (Dag Lem)
Дагфинн Ильмари Маннсакер (Dagfinn Ilmari Mannsåker)
Дамир Белялов (Damir Belyalov)
Даниель Вестерман (Daniel Westermann)
Даниил Шелепанов (Daniel Shelepanov)
Даниэль Верите (Daniel Vérité)
Даниэль Густафссон (Daniel Gustafsson)
Даниэль Фредуль (Daniel Fredouille)
Даррен Раш (Darren Rush)
Деврим Гюндюз (Devrim Gündüz)
Дейв Крамер (Dave Cramer)
Дейв Пейдж (Dave Page)
Денис Ерохин (Denis Erokhin)
Денис Лаксельд (Denis Laxalde)
Джастин Призби (Justin Pryzby)
Джейкоб Спидел (Jacob Speidel)
Джейкоб Чемпион (Jacob Champion)

Джеймс Коулман (James Coleman)
Джеймс Панг (James Pang)
Джереми Шнайдер (Jeremy Schneider)
Джефф Девис (Jeff Davis)
Джефф Джейнс (Jeff Janes)
Дживан Чок (Jeevan Chalke)
Джим Джонс (Jim Jones)
Джим Кинер (Jim Keener)
Джим Нэсби (Jim Nasby)
Джо Конвей (Joe Conway)
Джозеф Кошаков (Joseph Koshakow)
Джон Моррис (John Morris)
Джон Нейлор (John Naylor)
Джон Рассел (John Russell)
Джон Сюй (John Hsu)
Джон Экинс (John Ekins)
Джонатан С Кац (Jonathan S. Katz)
Джош Купершмидт (Josh Kupershmidt)
Джошуа Д. Дрейк (Joshua D. Drake)
Джошуа Уэхара (Joshua Uyehara)
Джоэл Джейкобсон (Joel Jacobson)
Джубили Янг (Jubilee Young)
Дилип Кумар (Dilip Kumar)
Димитриос Апостолу (Dimitrios Apostolou)
Дин Рашид (Dean Rasheed)
Дмитрий Васильев (Dmitry Vasiliev)
Дмитрий Долгов (Dmitry Dolgov)
Дмитрий Коваль (Dmitry Koval)
Дмитрий Ячник (Dzmitry Jachnik)
Доминик Девьен (Dominique Devienne)
Донмин Лю (Dongming Liu)
Дрю Каллахан (Drew Callahan)
Дрю Кимбалл (Drew Kimball)
Дунхан Линь (Donghang Lin)
Дэвид Дж. Джонстон (David G. Johnston)
Дэвид Кристенсен (David Christensen)
Дэвид Кук (David Cook)
Дэвид Перес (David Perez)
Дэвид Роули (David Rowley)
Дэвид Стил (David Steele)
Дэвид Уилер (David Wheeler)
Дэвид Хиллман (David Hillman)
Дэвид Чжан (David Zhang)
Евгений Коньков (Eugen Konkov)
Евгений Морозов (Evgeny Morozov)
Егор Рогов (Egor Rogov)
Егор Чиндяскин (Egor Chindyaskin)
Егор Юзефович (Yahor Yuzefovich)
Екатерина Кирьянова (Ekaterina Kiryanova)
Елена Индрупская (Elena Indrupskaya)
Ерки Ээссаар (Erki Eessaar)
Жак Комбринк (Jacques Combrink)
Жиль Даролд (Gilles Darold)
Жиль Парк (Gilles Parc)
Жуй Жао (Rui Zhao)
Жюльен Руо (Julien Rouhaud)
Зубейр Эрилмаз (Zubeyr Eryilmaz)
Ив Колен (Yves Colin)

Иван Картышов (Ivan Kartyshov)
Иван Коломбет (Ivan Kolombet)
Иван Лазарев (Ivan Lazarev)
Иван Панченко (Ivan Panchenko)
Иван Трофимов (Ivan Trofimov)
Израэл Барт (Israel Barth Rubio)
Илья Ненашев (Ilya Nenashev)
Итан Мерц (Ethan Mertz)
Иэн Барвик (Ian Barwick)
Йелте Феннема-Нио (Jelte Fennema-Nio)
Кайдо Вайкла (Kaido Vaikla)
Камбам Винай (Kambam Vinay)
Карен Таларико (Karen Talarico)
Карина Лицкевич (Karina Litskevich)
Карл О. Пинц (Karl O. Pinc)
Кашиф Зишан (Kashif Zeeshan)
Квинь Тран (Quynh Tran)
Ким Йохан Андерссон (Kim Johan Andersson)
Кирилл Решке (Kirill Reshke)
Кирк Волак (Kirk Wolak)
Кирк Паркер (Kirk Parker)
Кисун Квон (Kisoon Kwon)
Клаудио Фрейре (Claudio Freire)
Колин Кейн (Colin Caine)
Конг Ман (Kong Man)
Константин Книжник (Konstantin Knizhnik)
Кори Хинкер (Corey Huinker)
Кохэй КайГай (Kohei KaiGai)
Кохэй Суту (Kouhei Sutou)
Коэн Де Груте (Koen De Groote)
Крис Трэверс (Chris Travers)
Кристиан Маурер (Christian Maurer)
Кристиан Сторк (Christian Stork)
Кристоф Берг (Christoph Berg)
Кристоф Куртуа (Christophe Courtois)
Кристоф Хейсс (Christoph Heiss)
Кристофер Клайн (Christopher Kline)
Кришнакумар Р (Krishnakumar R)
Кунтал Гхош (Kuntal Ghosh)
Курт Коловсон (Curt Kolovson)
Курт Реккс (Kurt Roeckx)
Кэмерон Фогт (Cameron Vogt)
Кэри Хуан (Cary Huang)
Кётаро Хоригути (Kyotaro Horiguchi)
Лан Лю (Lang Liu)
Ларс Канис (Lars Kanis)
Лауренц Альбе (Laurenz Albe)
Лаури Лаанметс (Lauri Laanmets)
Легс Мэншн (Legs Mansion)
Ли Дон Ук (Lee Dong Wook)
Лукас Фиттл (Lukas Fittl)
Магнус Хагандер (Magnus Hagander)
Майквел Грасси (Maiquel Grassi)
Майкл Бондаренко (Michael Bondarenko)
Майкл Ван (Michael Wang)
Максим Богук (Maxim Boguk)
Максим Буайе (Maxime Boyer)
Максим Орлов (Maxim Orlov)

Максим Яблоков (Maxim Yablokov)
Манос Эммануилидис (Manos Emmanouilidis)
Мариан Круцина (Marian Krucina)
Марина Полякова (Marina Polyakova)
Марк Гертин (Mark Guertin)
Марк Дилгер (Mark Dilger)
Марк Слоан (Mark Sloan)
Маркос Пегораро (Marcos Pegoraro)
Маркус Винанд (Markus Winand)
Марлин Рейтерер (Marlene Reiterer)
Марсель Хофштеттер (Marcel Hofstetter)
Мартин Маркес (Martín Marqués)
Мартин Нэш (Martin Nash)
Мартин Шлоссарек (Martin Schlossarek)
Масаки Кувамура (Masaki Kuwamura)
Масао Фудзии (Masao Fujii)
Масахико Савада (Masahiko Sawada)
Масахиро Икеда (Masahiro Ikeda)
Матеус Алькантара (Matheus Alcantara)
Матс Киндаль (Mats Kindahl)
Маттиас Кун (Matthias Kuhn)
Маттиас ван де Меент (Matthias van de Meent)
Махендракар Шринивасарао (Mahendrakar Srinivasarao)
Мейсон Шарп (Mason Sharp)
Мелани Плейгман (Melanie Plageman)
Мелих Мутлу (Melih Mutlu)
Мерлин Монкьюр (Merlin Moncure)
Мика Гейт (Micah Gate)
Микаэль Пакье (Michael Paquier)
Мингли Чжан (Mingli Zhang)
Мирослав Бендик (Miroslav Bendik)
Михаил Грибков (Mikhail Gribkov)
Михаил Жилин (Michael Zhilin)
Михаил Николаев (Michail Nikolaev)
Михал Бартак (Michal Bartak)
Михал Клечек (Michal Kleczek)
Михаэль Банк (Michael Banck)
Моааз Ассали (Moaaz Assali)
Мураликришна Бандару (Muralikrishna Bandaru)
Назир Билал Явуз (Nazir Bilal Yavuz)
Натан Боссарт (Nathan Bossart)
Нгиги Вайтака (Ngigi Waithaka)
Никита Глухов (Nikita Glukhov)
Никита Калинин (Nikita Kalinin)
Никита Малахов (Nikita Malakhov)
Николай Самохвалов (Nikolay Samokhvalov)
Николай Шаплов (Nikolay Shaplov)
Никхил Бенеш (Nikhil Benesch)
Нил Тиффин (Neil Tiffin)
Нитин Ядав (Nitin Jadhav)
Нихил Радж (Nikhil Raj)
Ниша Мунд (Nisha Moond)
Нишант Шарма (Nishant Sharma)
Ной Миш (Noah Misch)
Нориёси Синода (Noriyoshi Shinoda)
Оле Педер Брандцег (Ole Peder Brandtzæg)
Олег Бартунов (Oleg Bartunov)
Олег Самойлов (Olleg Samoylov)

Олег Сибиряков (Oleg Sibiryakov)
Олег Целебровский (Oleg Tselebrovskiy)
Ондер Каладжи (Onder Kalaci)
Ондрей Навратил (Ondrej Navratil)
Пабло Харо (Pablo Kharo)
Павел Борисов (Pavel Borisov)
Павел Кулаков (Pavel Kulakov)
Павел Лузанов (Pavel Luzanov)
Павел Стехуле (Pavel Stehule)
Павло Голуб (Pavlo Golub)
Палак Чатурведи (Palak Chaturvedi)
Панделис Теодосиу (Pantelis Theodosiou)
Педро Гальегос (Pedro Gallegos)
Пит Сторер (Pete Storer)
Питер Гейган (Peter Geoghegan)
Питер Смит (Peter Smith)
Питер Эйзентраут (Peter Eisentraut)
Пол Амонсон (Paul Amonson)
Пол Юнгвирт (Paul Jungwirth)
Пшемыслав Штох (Przemyslaw Sztoch)
Пьер Дюкроке (Pierre Ducroquet)
Пьер Фортен (Pierre Fortin)
Рагхувир Девулапалли (Raghuveer Devulapalli)
Райан Макгвайр (Rian McGuire)
Ранье Вилела (Ranier Vilela)
Рид Томпсон (Reid Thompson)
Ридван Коркмаз (Ridvan Korkmaz)
Ричард Весли (Richard Vesely)
Ричард Гуо (Richard Guo)
Роберт Скотт (Robert Scott)
Роберт Трит (Robert Treat)
Роберт Хаас (Robert Haas)
Роберто Мелло (Roberto Mello)
Робинс Таракан (Robins Tharakan)
Роман Лозко (Roman Lozko)
Ронан Данклау (Ronan Dunklau)
Рё Мацумура (Ryo Matsumura)
Рёга Йошида (Ryoga Yoshida)
Саймон Риггс (Simon Riggs)
Сами Имсейх (Sami Imseih)
Самир Кумар (Sameer Kumar)
Самуэль Дюссо (Samuel Dussault)
Санджай Минни (Sanjay Minni)
Сатору Коидзуми (Satoru Koizumi)
Сванте Рихтер (Svante Richter)
Светлана Деревянко (Svetlana Derevyanko)
Себастьян Скалаки (Sebastian Skalacki)
Седрик Вильмен (Cédric Villemain)
Сергей Глухов (Sergei Glukhov)
Сергей Корнилов (Sergei Kornilov)
Сергей Прохоренко (Sergey Prokhorenko)
Сергей Саргсян (Sergey Sargsyan)
Сергей Шиндерук (Sergey Shinderuk)
Сильвен Френдез (Sylvain Frandaz)
Син Го (Xing Guo)
Синъя Като (Shinya Kato)
Соумйадип Чакраборти (Soumyadeep Chakraborty)
Степан Неретин (Stepan Neretin)

Степан Руц (Stepan Rutz)
Стефан Ташуар (Stephane Tachoires)
Стефан Шильдкнехт (Stéphane Schildknecht)
Стив Аткинс (Steve Atkins)
Стив Чавес (Steve Chavez)
Стивен Фрост (Stephen Frost)
Стэн Ху (Stan Hu)
Сурадж Хамкар (Suraj Khamkar)
Сурадж Хараге (Suraj Kharage)
Сюйдун Ян (Xudong Yang)
Сян Лю (Xiang Liu)
Сяожань Ван (Xiaoran Wang)
Такаюки Цунакава (Takayuki Tsunakawa)
Тацуо Исии (Tatsuo Ishii)
Тацуро Ямада (Tatsuro Yamada)
Тендер Ван (Tender Wang)
Тим Кэри-Смит (Tim Carey-Smith)
Тим Нидэм (Tim Needham)
Тим Палмер (Tim Palmer)
Тобиас Бусман (Tobias Bussmann)
Том Браун (Thom Brown)
Том Лейн (Tom Lane)
Томас Манро (Thomas Munro)
Томаш Вондра (Tomas Vondra)
Томми Павличек (Tommy Pavlicek)
Томонари Кацумата (Tomonari Katsumata)
Тристан Партин (Tristan Partin)
Тристан Рааб (Tristen Raab)
Тунг Нгуен (Tung Nguyen)
Уве Биндер (Uwe Binder)
Умайр Шахид (Umair Shahid)
Фабрицио де Ройес Мелло (Fabrízio de Royes Mello)
Фабьен Коэльо (Fabien Coelho)
Фаер Эмералд (Fire Emerald)
Фариас де Оливейра (Farias de Oliveira)
Фелипе Поззер (Feliphe Pozzer)
Филип Уорнер (Philip Warner)
Филипп Сальвисберг (Philipp Salvisberg)
Флавьен Гюдез (Flavien Guedez)
Флорис Ван Ни (Floris Van Nee)
Франк Штрайциг (Frank Streitzig)
Франческо Деграсси (Francesco Degrassi)
Фёдор Сигаев (Teodor Sigaev)
Хаджиме Мацунага (Hajime Matsunaga)
Хайато Курода (Hayato Kuroda)
Хайвел Карвер (Hywel Carver)
Хайин Тан (Haiying Tang)
Хайме Казанова (Jaime Casanova)
Хал Такахара (Hal Takahara)
Ханефи Ональди (Hanefi Onaldi)
Ханну Кросинг (Hannu Krosing)
Ханс Бушман (Hans Buschmann)
Хао Ву (Hao Wu)
Хао Чжан (Hao Zhang)
Хейкки Линнакангас (Heikki Linnakangas)
Хемант Сандрана (Hemanth Sandrana)
Химаншу Упадхьяя (Himanshu Upadhyaya)
Хината Мицуру (Mitsuru Hinata)

Хиронобу Судзуки (Hironobu Suzuki)
Хольгер Райзе (Holger Reise)
Хорди Гутьеррес (Jordi Gutiérrez)
Хорст Рейтерер (Horst Reiterer)
Хоу Чжицзе (Hou Zhijie)
Хуберт Любачевски (Hubert Lubaczewski)
Хунсюй Ма (Hongxu Ma)
Хунъюй Сонг (Hongyu Song)
Цзинсянь Ли (Jingxian Li)
Цзинтан Чжан (Jingtang Zhang)
Цзинчжоу Фу (Jingzhou Fu)
Цзумин Цзян (Zuming Jiang)
Цзюньван Жао (Junwang Zhao)
Цзянь Гуо (Jian Guo)
Цзянь Хи (Jian He)
Цюань Цзунлян (Quan Zongliang)
Чанхун Фей (Changhong Fei)
Чепмен Флэк (Chapman Flack)
Чжихун Юй (Zhihong Yu)
Чэнси Сунь (Chengxi Sun)
Шанталь Келлер (Chantal Keller)
Шаочжун Ши (Shaozhong Shi)
Швета Малик (Shveta Malik)
Шихао Чжун (Shihao Zhong)
Шлок Кьял (Shlok Kyal)
Шон Томас (Shaun Thomas)
Шраван Велагандула (Sravan Velagandula)
Шрути Говда (Shruthi Gowda)
Шубхам Ханна (Shubham Khanna)
Шулинь Чжоу (Shulin Zhou)
Шэй Роджански (Shay Rojansky)
Эван Макбет (Evan Macbeth)
Эйлер Тавейра (Euler Taveira)
Элизабет Кристенсен (Elizabeth Christensen)
Эмре Хасегели (Emre Hasegeli)
Энди Фан (Andy Fan)
Эндрю Алсуп (Andrew Alsup)
Эндрю Аткинсон (Andrew Atkinson)
Эндрю Дунстан (Andrew Dunstan)
Эндрю Кейн (Andrew Kane)
Энтони Сюй (Anthony Hsu)
Эрик Винхольд (Erik Wienhold)
Эрик Кир (Eric Cyr)
Эрик Мутта (Eric Mutta)
Эрик Радман (Eric Radman)
Эрик Ридж (Eric Ridge)
Эрик Рижкерс (Erik Rijkers)
Эцуро Фудзита (Etsuro Fujita)
Юго Нагата (Yugo Nagata)
Юйхан Цю (Yuhang Qiu)
Юки Сэйно (Yuki Seino)
Юки Фудзи (Yuuki Fujii)
Юн Ли (Yong Li)
Юнтао Хуан (Yongtao Huang)
Юрий Рашковский (Yurii Rashkovskii)
Юрий Соколов (Yura Sokolov)
Юя Ватари (Yuya Watari)
Якуб Вартак (Jakub Wartak)

Ян Ильясов (Ian Ilyasov)
Яни Рахкола (Jani Rahkola)
Япинь Ли (Japin Li)
Ярослав Сабуров (Yaroslav Saburov)
Яцзюнь Ху (Yajun Hu)

E.13. Предыдущие выпуски

Замечания к выпускам предыдущих версий опубликованы на сайте. На момент выпуска Postgres Pro Standard 17 поддерживались следующие версии:

- Postgres Pro Standard 16: *https://postgrespro.ru/docs/postgrespro/16/release.html*
- Postgres Pro Standard 15: *https://postgrespro.ru/docs/postgrespro/15/release.html*
- Postgres Pro Standard 14: *https://postgrespro.ru/docs/postgrespro/14/release.html*
- Postgres Pro Standard 13: *https://postgrespro.ru/docs/postgrespro/13/release.html*
- Postgres Pro Standard 12: *https://postgrespro.ru/docs/postgrespro/12/release.html*

Приложение F. Дополнительно поставляемые модули и расширения, поставляемые в `postgrespro-std-17-contrib`

В этом приложении, Приложении G и Приложении I содержится информация о необязательных компонентах, поставляемых в составе дистрибутива Postgres Pro Standard. В их число входят средства портирования, утилиты анализа и подключаемые функции, не включённые в состав основной системы Postgres Pro. Они отделены в основном потому, что адресованы ограниченной аудитории или находятся в экспериментальном состоянии, не подходящем для основного дерева кода. Однако это всё не умаляет их полезность.

В этом приложении и Приложении G описываются расширения и другие подключаемые серверные модули. В Приложении I описываются вспомогательные программы.

Многие компоненты предоставляют дополнительные пользовательские функции, операторы и типы, упакованные в *расширения*. Чтобы использовать одно из таких расширений, когда его исполняемый код установлен, необходимо зарегистрировать новые объекты SQL в СУБД. Для этого нужно воспользоваться командой CREATE EXTENSION. В чистой базе данных можно просто выполнить:

```
CREATE EXTENSION имя_расширения;
```

При этом новые объекты SQL будут зарегистрированы только в текущей базе данных, так что эту команду нужно выполнять в каждой базе данных, в которой планируется использовать функциональность этого расширения. Также можно запустить его в базе данных `template1`, чтобы установленное расширение копировалось во все впоследствии создаваемые базы по умолчанию.

Для всех этих расширений команду `CREATE EXTENSION` должен выполнять суперпользователь, если только расширение не помечено как «доверенное». Доверенные расширения могут устанавливать любые пользователи, имеющие право `CREATE` в текущей базе данных. В следующих разделах, где описываются расширения, отмечено, какие из них являются доверенными. Вообще говоря, доверенными расширениями считаются те, которые не предоставляют доступ к функциональности за рамками базы данных.

Следующие расширения являются доверенными при установке по умолчанию:

btree_gin
btree_gist
citext
cube
dict_int
fuzzystrmatch
hstore
intarray
isn
lo
ltree
pgcrypto
pg_trgm
seg
tablefunc
tcn
tsm_system_rows
tsm_system_time
unaccent
uuid-ossp

Многие расширения позволяют устанавливать свои объекты в схему по выбору. Для этого нужно добавить `SCHEMA` *`имя_схемы`* в команду `CREATE EXTENSION`. По умолчанию объекты устанавливаются в текущую схему для создаваемых объектов, которой по умолчанию становится `public`.

Однако некоторые из этих компонентов не являются «расширениями» в этом смысле, а подключаются к серверу по-другому, например, через параметр конфигурации shared_preload_libraries. Подробнее об этом говорится в документации каждого компонента.

В этом приложении содержится информация о модулях и расширениях, которые доступны в Postgres Pro Standard как отдельные подпакеты `postgrespro-std-17-contrib`. Обратите внимание, что расширение toastapi для добавления пользовательских реализаций TOAST также доступно в `postgrespro-std-17-contrib` (за подробной информацией обратитесь к Подразделу 63.2.3).

F.1. amcheck — модуль с инструментами, проверяющими целостность таблиц и индексов

Модуль `amcheck` предоставляет функции, позволяющие проверять логическую целостность структуры отношений.

Функции проверки B-дерева контролируют различные *инварианты* в структуре представления определённых отношений. Правильность работы функций методов доступа, стоящих за сканированием индекса и другими важными операциями, зависит от всегда соблюдаемых инвариантов. Например, определённые функции проверяют, помимо остальных вещей, что все страницы B-дерева содержат элементы в «логическом» порядке (например, индекс-B-дерево, построенный по столбцу `text`, должен содержать кортежи, упорядоченные в лексическом порядке с учётом правила сортировки). Если этот конкретный инвариант каким-то образом нарушается, следует ожидать, что бинарный поиск на затронутой странице введёт в заблуждение процедуру сканирования индекса, что приведёт к неверным результатам запросов SQL. Если нарушения структуры не обнаруживаются, эти функции отрабатывают без ошибок. На время выполнения этих функций search_path временно меняется на `pg_catalog, pg_temp`.

Проверка выполняется теми же процедурами, что используются при сканировании индекса, и это может быть код пользовательского класса операторов. Например, проверка индекса-B-дерева задействует сравнения, выполняемые одной или несколькими опорными функциями B-дерева под номером 1. Подробнее опорные функции класса операторов описываются в Подразделе 36.16.3.

В отличие от функций проверки B-дерева, которые сообщают о повреждениях, выдавая ошибки, функция проверки кучи `verify_heapam` проверяет таблицу и пытается вернуть набор строк: по одной строке на каждое обнаруженное повреждение. Тем не менее, если будут повреждены средства, которые использует функция `verify_heapam`, она не сможет продолжить работу и выдаст ошибку.

Права на выполнение функций `amcheck` можно выдать и не суперпользователю, но перед этим следует тщательно учесть аспекты безопасности и конфиденциальности данных. Хотя эти функции прежде всего выдают информацию о структуре данных и характере найденных повреждений, а не выводят собственно повреждённые данные, если злоумышленник сможет выполнять эти функции, а особенно если ему также удастся вызвать повреждение данных, он сможет узнать что-то о самих данных из полученной информации.

F.1.1. Функции

`bt_index_check(index regclass, heapallindexed boolean, checkunique boolean) returns void`

`bt_index_check` проверяет, соблюдаются ли в целевом индексе-B-дереве различные инварианты. Пример использования:

```
test=# SELECT bt_index_check(index => c.oid, heapallindexed => i.indisunique),
               c.relname,
               c.relpages
FROM pg_index i
JOIN pg_opclass op ON i.indclass[0] = op.oid
JOIN pg_am am ON op.opcmethod = am.oid
JOIN pg_class c ON i.indexrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE am.amname = 'btree' AND n.nspname = 'pg_catalog'
-- Не проверять временные таблицы (они могут относиться к другим сеансам):
AND c.relpersistence != 't'
-- Функция может выдать ошибку без этих условий:
AND c.relkind = 'i' AND i.indisready AND i.indisvalid
ORDER BY c.relpages DESC LIMIT 10;
 bt_index_check |             relname             | relpages
----------------+---------------------------------+----------
                | pg_depend_reference_index       |       43
                | pg_depend_depender_index        |       40
```

```
                 | pg_proc_proname_args_nsp_index  |       31
                 | pg_description_o_c_o_index      |       21
                 | pg_attribute_relid_attnam_index |       14
                 | pg_proc_oid_index               |       10
                 | pg_attribute_relid_attnum_index |        9
                 | pg_amproc_fam_proc_index        |        5
                 | pg_amop_opr_fam_index           |        5
                 | pg_amop_fam_strat_index         |        5
(10 rows)
```

Этот пример демонстрирует сеанс проверки 10 самых больших индексов системных каталогов в базе данных «test». Проверка всех кортежей кучи на предмет наличия соответствующих кортежей индекса запрашивается только для тех из этих индексов, которые являются уникальными. Так как ошибки не было, все проверенные индексы представляются логически целостными. Естественно, этот запрос можно легко изменить, чтобы функция `bt_index_check` вызывалась для всех индексов в базе, которые поддерживают эту проверку.

Функция `bt_index_check` запрашивает блокировку `AccessShareLock` для целевого индекса и отношения, которому он принадлежит. Это тот же режим блокировки, что запрашивается для отношений обычными операторами `SELECT`. `bt_index_check` не проверяет инварианты, существующие в иерархии потомок/родитель, но проверяет представление всех кортежей кучи в индексе в виде индексных кортежей, когда параметр *`heapallindexed`* равен `true`. Если *`checkunique`* имеет значение `true`, `bt_index_check` проверяет, что в каждом индексе с ограничением уникальности нет одновременно видимых дубликатов. Когда в работающей производственной среде требуется регулярная лёгкая проверка на наличие нарушений, использование `bt_index_check` часто будет подходящим компромиссом между полнотой проверки и минимизацией влияния на производительность и доступность приложения.

`bt_index_parent_check(index regclass, heapallindexed boolean, rootdescend boolean, checkunique boolean) returns void`

Функция `bt_index_parent_check` проверяет, соблюдаются ли в целевом объекте, индексе B-дереве, различные инварианты. Кроме того, если аргумент *`heapallindexed`* равен `true`, эта функция проверяет наличие в индексе всех кортежей из кучи, которые должны в него попасть. Если аргумент *`checkunique`* равен `true`, `bt_index_check` проверяет, что в каждом индексе с ограничением уникальности нет одновременно видимых дубликатов. Когда необязательный параметр *`rootdescend`* равен `true`, при проверке для каждого кортежа на уровне листьев производится ещё один поиск, начиная с корневой страницы. Проверки, которые может производить `bt_index_parent_check`, включают в себя все проверки, выполняемые функцией `bt_index_check`. Функцию `bt_index_parent_check` можно считать более полноценным вариантом `bt_index_check`: в отличие от `bt_index_check`, `bt_index_parent_check` проверяет ещё и инварианты, существующие в иерархии родитель/потомок, в том числе отсутствие потерянных связей в структуре индекса. `bt_index_parent_check` следует общему соглашению и выдаёт ошибку в случае обнаружения логической несогласованности или другой проблемы.

Функция `bt_index_parent_check` запрашивает в целевом индексе блокировку `ShareLock` (также `ShareLock` запрашивается и в основном отношении). Эти блокировки предотвращают одновременное изменение данных командами `INSERT`, `UPDATE` и `DELETE`. Эти блокировки также препятствуют одновременной обработке нижележащего отношения командой `VACUUM` и другими вспомогательными командами. Заметьте, что эта функция удерживает блокировки только во время выполнения, а не на протяжении всей транзакции.

Дополнительные проверки, проводимые функцией `bt_index_parent_check`, более ориентированы на выявление различных патологических случаев. В том числе это может быть неправильно реализованный класс операторов B-дерева, используемый проверяемым индексом, или, гипотетически, неизвестные ошибки в нижележащем коде метода доступа индекса-B-дерева. Заметьте, что функцию `bt_index_parent_check` нельзя применять, когда включён режим горячего резерва (то есть на физических репликах в режиме «только чтение»), в отличие от `bt_index_check`.

Подсказка

Функции `bt_index_check` и `bt_index_parent_check` выводят отладочные сообщения о процессе проверки на уровнях важности `DEBUG1` и `DEBUG2`. Эти сообщения содержат подробные сведения о проверке, которые могут представлять интерес для разработчиков PostgreSQL. Они могут быть полезны и для опытных пользователей в качестве дополнительного контекста в случае обнаружения несогласованности. Чтобы получать сообщения о процессе проверки с подходящим уровнем детализации, выполните до запуска проверяющего запроса в интерактивном psql:

```
SET client_min_messages = DEBUG1;
```

`verify_heapam(relation regclass, on_error_stop boolean, check_toast boolean, skip text, startblock bigint, endblock bigint, blkno OUT bigint, offnum OUT integer, attnum OUT integer, msg OUT text) returns setof record`

Проверяет таблицу, последовательность или материализованное представление на наличие структурных повреждений (когда страницы отношения содержат данные в неправильном формате) и логических повреждений (когда структура страниц не повреждена, но и не соответствует остальной части кластера).

В качестве необязательных аргументов принимаются:

`on_error_stop`

Если true, проверка на наличие повреждений останавливается в конце первого блока, в котором обнаружены какие-либо повреждения.

По умолчанию false.

`check_toast`

Если true, поля TOAST проверяются по TOAST-таблице целевого отношения.

Эта проверка может быть длительной. Кроме того, если TOAST-таблица или её индекс повреждены, проверка TOAST-полей потенциально может привести к сбою в работе сервера, хотя во многих случаях это просто вызовет ошибку.

По умолчанию false.

`skip`

Если указано значение, отличное от `none`, проверка на наличие повреждений пропускает блоки, которые помечены как полностью видимые или полностью замороженные, в зависимости от указанного значения. Допустимые варианты: `all-visible`, `all-frozen` и `none`.

По умолчанию `none`.

`startblock`

Если этот параметр задан, проверка на наличие повреждений начинается с указанного блока и пропускает все предыдущие блоки. Если в *`startblock`* указано значение вне диапазона блоков целевой таблицы, выдаётся ошибка.

По умолчанию проверка начинается с первого блока.

`endblock`

Если этот параметр задан, проверка на наличие повреждений заканчивается на указанном блоке и пропускает все оставшиеся блоки. Если в *`endblock`* указано значение вне диапазона блоков целевой таблицы, выдаётся ошибка.

По умолчанию проверяются все блоки.

Для каждого обнаруженного повреждения `verify_heapam` возвращает строку со следующими столбцами:

`blkno`

Номер блока, содержащего повреждённую страницу.

`offnum`

Номер смещения повреждённого кортежа.

`attnum`

Номер атрибута повреждённого столбца в кортеже, если повреждён именно столбец, а не весь кортеж.

`msg`

Сообщение с описанием выявленной ошибки.

F.1.2. Дополнительная проверка *heapallindexed*

Когда аргумент *`heapallindexed`* функций проверки B-дерева равен `true`, для таблицы, связанной с отношением целевого индекса, добавляется дополнительная фаза проверки. Она включает «фиктивную» операцию `CREATE INDEX`, которая проверяет присутствие всех гипотетических новых индексных кортежей по временной сводной структуре в памяти (она создаётся при необходимости на первом этапе проверки). Сводная структура «помечает» каждый кортеж, который находится в целевом индексе. На высоком уровне идея проверки *`heapallindexed`* состоит в том, чтобы убедиться, что новый индекс, равнозначный целевому, содержит только те записи, которые можно найти в существующей структуре.

С дополнительным этапом *`heapallindexed`* связаны значительные издержки: проверка обычно будет выполняться в несколько раз дольше. Однако никакие новые блокировки уровня отношения при проверке *`heapallindexed`* не запрашиваются.

Сводная структура ограничивается по объёму значением `maintenance_work_mem`. Для выявления несогласованности в представленных в индексе кортежах с вероятностью упущений в пределах 2% требуется приблизительно 2 байта памяти на кортеж. По мере уменьшения объёма памяти в пересчёте на кортеж этот процент медленно растёт. Этот подход значительно ограничивает издержки такой проверки, и при этом лишь немного уменьшается вероятность выявления проблемы, особенно в инсталляциях, где эта проверка включена в процедуру регулярного обслуживания. Даже если единичное отсутствие или повреждение кортежа упущено, есть все шансы выявить его при очередной проверке.

F.1.3. Эффективное использование amcheck

Модуль `amcheck` может быть полезен для выявления различных типов проблем, которые могут остаться незамеченными при включении контрольных сумм страниц данных. В частности это:

- Структурные несоответствия, возникающие при некорректной реализации класса операторов.

  В том числе это проблемы, возникающие при изменении правил сравнения в операционной системе. Сравнения данных сортируемого типа, например `text`, должны быть постоянными (как и все сравнения, применяемые при сканировании индекса-B-дерева), что подразумевает неизменность правил сортировки в операционной системе. Проблемы могут возникать при обновлениях правил в операционной системе, хотя такие случаи редки. Чаще проявляются несоответствия порядка сортировки между ведущим и ведомым сервером, например, из-за различий *основных* версий используемых операционных систем. Возникающие расхождения обычно наблюдаются только на ведомых серверах, так что и выявить их обычно можно только на них.

Когда возникает подобная проблема, она может затрагивать не абсолютно все индексы, построенные с порочным правилом сортировки, просто потому что *индексированные* значения могут иметь тот же абсолютный порядок, независящий от различий поведения. За дополнительными сведениями об использовании в Postgres Pro правил сортировки и локалей операционной системы обратитесь к Разделу 22.1 и Разделу 22.2.

- Несоответствия структуры между индексами и проиндексированными отношениями в куче (когда выполняется проверка *`heapallindexed`*).

  Во время обычных операций перекрёстная проверка индексов по отношениям в куче не производится. Симптомы повреждения данных в куче могут быть неочевидными.

- Повреждения, вызванные гипотетическими неизвестными ошибками в нижележащем коде методов доступа, коде сортировки и управления транзакциями Postgres Pro.

  Автоматическая проверка структурной целостности индексов играет важную роль в общем тестировании новых или предлагаемых возможностей Postgres Pro, с которыми может возникнуть логическая несогласованность. Такую же роль играет проверка структуры таблицы и связанной информации о видимости и состоянии транзакций. И поэтому одна из очевидных стратегий тестирования — регулярно вызывать функции `amcheck` при проведении стандартных регрессионных тестов.

- Ошибки в файловой системе или подсистеме хранения, когда просто не включены контрольные суммы.

  Заметьте, что `amcheck` рассматривает страницу в том виде, как она представлена в некотором буфере разделяемой памяти к моменту проверки, если при обращению к нужному блоку он уже находится в разделяемом буфере. Вследствие этого, `amcheck` не обязательно видит данные, находящиеся в файловой системе в момент проверки. Заметьте, что когда контрольные суммы включены, `amcheck` может выдать ошибку из-за несоответствия контрольных сумм, если в буфер будет считываться испорченный блок.

- Повреждения, вызванные дефектными чипами ОЗУ или вообще подсистемой памяти.

  Postgres Pro не защищает от ошибок памяти; предполагается, что в эксплуатируемом вами сервере установлена память с ECC (Error Correcting Codes, Коды исправления ошибок) или лучшая защита. Однако память ECC обычно защищает только от ошибок в одном бите и не следует считать её *абсолютной* защитой от сбоев, приводящих к повреждению памяти.

  Когда выполняется проверка *`heapallindexed`*, в целом значительно увеличивается шанс выявления ошибок в отдельных битах, так как она тестирует точное двоичное равенство и сверяет проиндексированные атрибуты с кучей.

Причиной повреждения структуры данных может стать неисправность оборудования или же перезапись или изменение файлов отношения сторонним ПО. Этот вид повреждения также можно обнаружить с помощью контрольных сумм страниц данных.

Даже если страницы отношений имеют правильный формат, их внутренняя структура не повреждена и данные соответствуют контрольной сумме, это не гарантирует отсутствие логических повреждений. Повреждения такого типа невозможно обнаружить с помощью контрольных сумм. Примеры: значения TOAST в основной таблице, для которых нет соответствующей записи в таблице TOAST; кортежи в основной таблице с идентификатором транзакции, который старше, чем самый старый действительный идентификатор транзакции в базе данных или кластере.

В производственных системах выявляются разнообразные причины логических повреждений: ошибки в серверном ПО PostgreSQL, неисправные и непродуманные инструменты резервного копирования и восстановления, ошибки пользователей.

Повреждение отношений особенно опасно в работающей производственной среде, то есть в той среде, где высокий риск совсем нежелателен. Поэтому для диагностики повреждения данных без излишнего риска была разработана функция `verify_heapam`. Она не может обеспечить защиту от всех сбоев сервера, так как даже выполнение вызывающего запроса может быть небезопасным на сильно повреждённой системе. Если сами каталоги повреждены, то вероятны проблемы с доступом к таблицам.

Вообще говоря, `amcheck` может доказать только наличие повреждений, но не доказать их отсутствие.

F.1.4. Исправление повреждений

Когда `amcheck` сигнализирует о повреждении данных, ложные срабатывания практически исключены. `amcheck` считает ошибочными ситуации, которые никогда не должны наблюдаться по определению, поэтому ошибки `amcheck`, как правило, требуют тщательного анализа.

Общего метода устранения проблем, которые может выявить `amcheck`, не существует. Начать нужно с поиска корня проблемы, приводящей к нарушению инварианта. Полезную роль в диагностике повреждений, которые выявляет `amcheck`, может сыграть pageinspect. Одна лишь команда `REINDEX` может быть неэффективна, когда потребуется исправить повреждения.

F.2. aqo — оптимизация запросов по стоимости выполнения

Модуль aqo представляет собой расширение Postgres Pro Standard для оптимизации запросов по стоимости выполнения. Используя методы машинного обучения, а точнее модификацию алгоритма k-NN, aqo улучшает оценку количества строк, что может способствовать выбору лучшего плана и, как следствие, ускорению запросов.

F.2.1. Описание

Модуль aqo может собирать статистику по всем выполняемым запросам, за исключением запросов, обращающихся к системным отношениям. Если запросы различаются только константами, они считаются относящимися к одному классу. Для каждого класса модуль aqo сохраняет для машинного обучения качество оценки количества строк, время планирования, время и статистику выполнения. На основе этих данных aqo строит новый план выполнения и использует его для следующего запроса того же класса. В тестах aqo показал значительное увеличение производительности для сложных запросов.

Модуль aqo сохраняет все данные обучения (aqo_data), запросы (aqo_query_texts), параметры запросов (aqo_queries) и статистику выполнения запросов (aqo_query_stat) в файлах. При запуске aqo эти данные загружаются в разделяемую память. Вы можете обращаться к данным aqo, используя функции и представления.

При включённом параметре aqo.advanced и запуске aqo в режиме `intelligent` или `learn` каждому классу запросов для его идентификации и разделения собранной статистики присваивается уникальное хеш-значение, вычисляемое на основе дерева запросов. Если `aqo.advanced` отключён, статистика для всех неотслеживаемых классов запросов хранится в общем классе запросов с хешем 0.

С каждым классом запросов связано отдельное пространство, называемое *пространством признаков*, в котором собирается статистика для данного класса запросов. Для идентификации этого пространства признаков используется хеш-значение (`fs`), называемое *основным хеш-значением*. Оно является общим для запросов, отличающихся только именами таблиц, что позволяет агрегировать данные обучения по таким запросам. С каждым пространством признаков связаны *подпространства признаков*, в которых собирается информация об избирательности и количестве строк для каждого узла плана запроса. Для идентификации каждого подпространства также используется хеш-значение (`fss`).

Параметры оптимизации для каждого запроса хранятся в представлении aqo_queries.

F.2.1.1. Ограничения

В настоящее время расширение aqo имеет следующие ограничения:

- Оптимизация запросов с использованием aqo не поддерживается на ведомых серверах.
- Оптимизация запросов с использованием aqo не поддерживается для запросов, обращающихся только к временным объектам.
- Оптимизация запросов с использованием aqo не поддерживается для запросов, содержащих функции `IMMUTABLE`.
- Модуль aqo не собирает статистику по репликам, поскольку они доступны только для чтения. Однако он может использовать статистику выполнения запросов с ведущего сервера при работе с физической репликой.
- Режимы `learn` и `intelligent` не должны работать на уровне кластера с запросами, имеющими динамически генерируемую структуру, поскольку в этих режимах сохраняются все идентификаторы классов запросов, которые различны для всех таких запросов. Тем не менее могут использоваться динамически генерируемые константы.

F.2.2. Установка и подготовка

Расширение aqo включено в состав Postgres Pro Standard. Установив Postgres Pro Standard, выполните следующие действия, чтобы подготовить aqo к работе:

1. Добавьте `aqo` в параметр shared_preload_libraries в файле `postgresql.conf`:

   ```
   shared_preload_libraries = 'aqo'
   ```

   Библиотеку aqo нужно предварительно загрузить при запуске сервера, так как адаптивная оптимизация запросов должна быть включена для всего кластера.
2. Создайте расширение aqo, выполнив следующий запрос:

   ```
   CREATE EXTENSION aqo;
   ```

Когда расширение будет создано, вы можете приступить к оптимизации запросов.

Чтобы отключить aqo в текущей базе данных, выполните:

```
DROP EXTENSION aqo;
```

Чтобы отключить aqo на уровне кластера, выполните следующее:

```
ALTER SYSTEM SET aqo.enable = off;
SELECT pg_reload_conf();
```

Чтобы удалить все данные из хранилища aqo, включая собранную статистику, вызовите функцию `aqo_reset()`. Чтобы удалить данные из текущей базы данных, выполните

```
SELECT aqo_reset();
```

Чтобы удалить все данные из хранилища aqo, выполните

```
SELECT aqo_reset(NULL);
```

Чтобы расширение aqo не загружалось при перезапуске сервера, удалите строку

```
shared_preload_libraries = 'aqo'
```

из файла `postgresql.conf`.

> **Важно**
>
> Чтобы избежать ошибок во время физической репликации при переносе данных aqo с ведущего сервера на реплику, убедитесь, что на серверах установлены одинаковые версии модуля. Если установлены разные версии aqo, необходимо задать значение `off` для параметра aqo.wal_rw на обоих серверах, однако в таком случае репликация выполняться не будет.

F.2.2.1. Конфигурирование

Поведение aqo в основном регулируется параметрами конфигурации aqo.enable, aqo.mode и aqo.advanced. Значения этих параметров по умолчанию позволяют начать обучение aqo в основном режиме, как только для параметра `aqo.enable` устанавливается значение `on`.

Чтобы динамически изменить в текущем сеансе любой из этих параметров, например режим, выполните следующую команду:

```
SET aqo.mode = 'режим';
```

Здесь *режим* — название режима работы, который будет использоваться.

F.2.3. Использование

F.2.3.1. Использование основного режима aqo

По умолчанию параметр aqo.advanced отключён. При этом устанавливается рекомендуемый основной режим, в котором статистика собирается для узлов плана (определяемых `fss`), а собранные данные машинного обучения используются для исправления погрешностей оценки количества строк для всех запросов, план которых содержит определённый узел плана. Как только для параметра aqo.enable устанавливается значение `on`, aqo начинает обучение. Следует несколько раз выполнить запросы, которые необходимо оптимизировать, пока план не станет достаточно хорошим, и изменить значение параметра aqo.mode на `frozen`. Чтобы применить данные машинного обучения на уровне экземпляра сервера, выполните следующие команды:

```
ALTER SYSTEM SET aqo.mode = frozen;
ALTER SYSTEM SET aqo.enable = on;
SELECT pg_reload_conf();
```

Данные машинного обучения будут применяться не только к запросам, на которых обучался модуль aqo, но также ко всем запросам с планом, содержащим узлы, по которым собиралась статистика. Чтобы данные машинного обучения не влияли на другие запросы, установите для параметра aqo.advanced значение `on`, и статистика будет собираться для отдельных запросов. За подробностями обратитесь к Подразделу F.2.3.2.

F.2.3.2. Выбор режима работы для расширенной оптимизации запросов

Если часто выполняются однотипные запросы, например, в приложении ограничено число возможных классов запросов, можно включить параметр aqo.advanced и использовать интеллектуальный режим (`intelligent`) для улучшения планирования таких запросов. В этом режиме aqo анализирует выполнение каждого запроса и собирает статистику. При этом статистика по разным классам запросов собирается отдельно. Если производительность не увеличивается после 50 итераций, расширение aqo отключается, и планирование выполняется стандартным планировщиком запросов.

> **Примечание**
>
> Можно посмотреть текущий план запроса, воспользовавшись стандартной командой Postgres Pro `EXPLAIN` с указанием `ANALYZE`. За подробностями обратитесь к Разделу 14.1.

Так как в режиме `intelligent` различные классы запросов анализируются отдельно, aqo может не улучшить производительность, если запросы в рабочей нагрузке относятся к нескольким разным классам или постоянно меняются. Для такого профиля нагрузки стоит перевести aqo в режим `controlled` или попробовать отключить параметр `aqo.advanced`.

Когда включён параметр `aqo.advanced`, в режиме `controlled` расширение aqo не собирает статистику для новых классов запросов, поэтому они не будут оптимизированы, но для известных классов запросов aqo продолжит собирать статистику и использовать оптимизированные алгоритмы планирования. Поэтому следует использовать режим `controlled` только после обучения aqo в режиме `learn` или `intelligent`.

После того, как aqo прошло обучение, рекомендуется использовать режим `controlled` для рабочей среды. Чтобы aqo работало в этом режиме на уровне всего производственного кластера, выполните

```
ALTER SYSTEM SET aqo.mode = 'controlled';
SELECT pg_reload_conf();
```

При включённом параметре `aqo.advanced` расширение в режиме `learn` собирает статистику по всем выполненным запросам и обновляет данные для классов запросов. Этот режим аналогичен режиму `intelligent`, за исключением возможности интеллектуальной настройки. Его не рекомендуется использовать постоянно для всего кластера, поскольку он пытается оптимизировать aqo для каждого класса запросов, даже для тех, которым это не нужно, и может привести к ненужным вычислительным издержкам и снижению производительности.

Для рабочей нагрузки с динамически генерируемой структурой запросов используйте режим `learn` с отключённым параметром. Общее улучшение производительности не гарантируется. Поскольку в этом режиме отсутствует интеллектуальная настройка, производительность некоторых запросов может даже снизиться, однако он больше подходит для динамически меняющейся нагрузки и потребляет меньше памяти, чем режим `intelligent`.

Когда параметр `aqo.advanced` включён, aqo в режиме `auto` работает как в режиме `learn`, но режим `auto` предотвращает переполнение памяти, реализуя механизм вытеснения из кеша давно неиспользуемых (LRU, least recently used) данных. Этот механизм гарантирует, что ограничения aqo.dsm_size_max и aqo.fss_max_items не будут превышены при сохранении входящих запросов. При вызове функции `aqo_query_texts_update` в этом режиме aqo не сохраняет тексты запросов и не освобождает используемую динамическую разделяемую память.

Параметр конфигурации `aqo.delta_rows` может быть полезен в случае с данными, которые могут существенно изменяться между запросами. Этот параметр позволяет aqo делать предсказания на основе оценки количества строк планировщика. Например, планировщик оценивает меньшее число строк при удалении данных из таблицы. Расширение aqo может использовать эту обновлённую информацию, чтобы предсказать меньшее число строк без необходимости дополнительного обучения.

Если вы хотите уменьшить влияние aqo на время планирования и исполнения запросов, вы можете использовать режим `frozen`. В этом режиме aqo только читает собранную статистику, но новые данные не собираются.

F.2.3.3. Тонкая настройка aqo

Для обращения к представлениям aqo и изменения расширенных свойств запросов необходимо иметь права суперпользователя.

Все обработанные классы запросов и соответствующие хеш-значения можно увидеть в представлении aqo_query_texts:

```
SELECT * FROM aqo_query_texts;
```

Чтобы узнать класс (то есть хеш) запроса и режим aqo, включите переменные среды aqo.show_hash (boolean) и aqo.show_details (boolean) и выполните запрос. Вывод будет примерно следующим:

```
...
Planning Time: 23.538 ms
...
Execution Time: 249813.875 ms
...
Using aqo: true
AQO mode: LEARN
AQO advanced: OFF
...
Query hash: -2439501042637610315
```

Разные классы запросов имеют собственные свойства оптимизации. Эти свойства отображаются в представлении aqo_queries:

```
SELECT * FROM aqo_queries;
```

Можно вручную изменять эти свойства, чтобы скорректировать оптимизацию для определённого класса запросов. Например:

```
 -- Добавление нового класса запросов в представление aqo_queries:

SET aqo.enable='on';
SET aqo.advanced='on';
SET aqo.mode='intelligent';
SELECT * FROM a, b WHERE a.id=b.id;
SET aqo.mode='controlled';

 -- Отключение автонастройки, включение learn_aqo и use_aqo
 -- для данного класса запросов:

SELECT count(*) FROM aqo_queries,
  LATERAL aqo_queries_update(fs, NULL, true, true, false)
  WHERE fs = (SELECT fs FROM aqo_query_texts
```

```
  WHERE query_text LIKE 'SELECT * FROM a, b WHERE a.id=b.id;');


 -- Запуск EXPLAIN ANALYZE и наблюдение изменённого плана:

EXPLAIN ANALYZE SELECT * FROM a, b WHERE a.id=b.id;
EXPLAIN ANALYZE SELECT * FROM a, b WHERE a.id=b.id;

 -- Отключение обучения для прекращения сбора статистики и
 -- начала использования оптимизированного плана:

SELECT count(*) FROM aqo_queries,
  LATERAL aqo_queries_update(fs, NULL, false, true, false)
  WHERE fs = (SELECT fs FROM aqo_query_texts
  WHERE query_text LIKE 'SELECT * FROM a, b WHERE a.id=b.id;');
```

Чтобы предотвратить интеллектуальную настройку для определённого класса запросов, отключите свойство `auto_tuning`:

```
SELECT count(*) FROM aqo_queries,
  LATERAL aqo_queries_update(fs, NULL, NULL, NULL, false)
  WHERE fs = 'hash';
```

Здесь *хеш* — это значение хеша для данного класса запросов. В результате aqo не будет автоматически менять свойства `learn_aqo` и `use_aqo`.

Чтобы отключить дальнейшее обучение для некоторого класса запросов, выполните следующую команду:

```
SELECT count(*) FROM aqo_queries,
  LATERAL aqo_queries_update(fs, NULL, false, NULL, false)
  WHERE fs = 'hash';
```

Здесь *хеш* — это значение хеша для данного класса запросов.

Чтобы полностью отключить aqo для всех запросов и использовать стандартный планировщик Postgres Pro, выполните:

```
SELECT count(*) FROM aqo_queries,
  LATERAL aqo_disable_class(fs, NULL)
  WHERE fs <> 0;
```

Чтобы временно отключить aqo для всех запросов в текущем сеансе или на уровне всего кластера, но не удалять и не изменять собранную статистику и параметры, отключите параметр aqo.enable, как указано ниже:

```
SET aqo.enable = 'off';
```

или

```
ALTER SYSTEM SET aqo.enable = 'off'
```

F.2.3.4. Режим «песочницы»

Вы можете экспериментировать с aqo, не затрагивая его основную базу знаний. Для этого выполните команду

```
SET aqo.sandbox = ON;
```

Она включит *режим «песочницы»*, в котором aqo будет работать в изолированной среде. Однако если включить aqo.sandbox в разных сеансах SQL, они будут использовать одни и те же данные.

Данные, полученные в режиме «песочницы», не реплицируются. Но режим «песочницы» можно использовать на ведомом сервере. Более того, единственный способ обучить aqo на ведомом сервере — включить режим «песочницы» при включённой репликации, то есть при значении true для `aqo.wal_rw`. Без режима «песочницы» aqo будет работать на ведомом сервере так, как будто `aqo.mode` = `FROZEN`, то есть сможет использовать существующую базу знаний, но не сможет её обновлять или расширять.

F.2.4. Справка

F.2.4.1. Параметры конфигурации

`aqo.enable` (`boolean`)

Определяет состояние расширения aqo. Если установлено значение `off`, aqo не работает, за исключением случаев, когда параметр aqo.force_collect_stat = `on`.

По умолчанию: `off` (выкл.).

`aqo.mode` (`text`)

Устанавливает режим работы aqo и определяет, как расширение будет обрабатывать новые запросы. Возможные значения:

- `intelligent` — расширение сохраняет новые запросы с включённым `auto_tuning`. За дополнительными сведениями обратитесь к описанию представления aqo_queries. В данном режиме aqo может отключиться для запроса в случае снижения средней производительности. Этот режим работает таким образом, только если параметр aqo.advanced = `on`, в противном случае работает аналогично режиму `learn`.
- `learn` — расширение собирает статистику по всем выполненным запросам, обучается и делает предсказания на основе этой статистики.
- `controlled` — расширение только обучается и делает предсказания для известных запросов.
- `frozen` — расширение делает предсказания для известных запросов, но не обучается ни на каких запросах.
- `auto` — работает как `learn`, но предотвращает переполнение памяти, реализуя механизм вытеснения из кеша давно неиспользуемых (LRU, least recently used) данных.

По умолчанию: `learn`.

`aqo.advanced` (`boolean`)

Включает расширенную процедуру обучения, которая сохраняет статистику обучения отдельно для каждого класса запроса. Также позволяет настраивать параметры `use_aqo` и `learn_aqo` в представлении aqo_queries. После тонкой настройки параметры запроса в представлении `aqo_query` продолжают работать при выключенном параметре `aqo.advanced`.

По умолчанию: `off` (выкл.).

`aqo.force_collect_stat` (`boolean`)

Собирать статистику выполнения запросов во всех режимах aqo даже при `aqo.enable` = `off`.

По умолчанию: `off` (выкл.).

`aqo.show_details` (`boolean`)

Добавлять некоторые детали в вывод команды `EXPLAIN` запроса, такие как предсказание или хеш подпространства признаков, и отображать некоторую дополнительную информацию, специфичную для aqo.

По умолчанию: `on` (вкл.).

`aqo.show_hash` (`boolean`)

Показывать хеш-значение, однозначно идентифицирующее класс запросов или класс узлов плана. Расширение aqo использует в качестве идентификатора класса запроса идентификатор самого Postgres Pro, чтобы обеспечить согласованность с другими расширениями, такими как pg_stat_statements. Таким образом, идентификатор запроса можно получить из поля `Query hash` в выводе команды `EXPLAIN ANALYZE`.

По умолчанию: `on` (вкл.).

`aqo.join_threshold` (`integer`)

Игнорировать запросы, содержащие количество соединений меньше указанного, то есть статистика для таких запросов не собирается.

По умолчанию: `0` (запросы не игнорируются).

`aqo.learn_statement_timeout` (`boolean`)

Обучаться на планах запросов, прерванных по тайм-ауту оператора.

По умолчанию: `off` (выкл.).

`aqo.statement_timeout` (`integer`)

Определяет начальное значение так называемого *«умного» тайм-аута операторов* в миллисекундах, который необходим для ограничения времени выполнения при ручном обучении aqo на специальных запросах с неудовлетворительным прогнозом количества строк. Расширение aqo может динамически изменять умный тайм-аут операторов во время этого обучения. Когда погрешность оценки количества строк на узлах превышает 0.1, значение `aqo.statement_timeout` автоматически увеличивается экспоненциально, но не превышает statement_timeout.

По умолчанию: `0`.

`aqo.wide_search` (`boolean`)

Включает поиск соседей с одним и тем же подпространством признаков среди разных классов запросов. Работает, только если параметр aqo.advanced = `on`.

По умолчанию: `off` (выкл.).

`aqo.min_neighbors_for_predicting` (`integer`)

Определяет, сколько выборок, собранных при предыдущих выполнениях запроса, будет использоваться для прогнозирования оценки количества строк в следующий раз. Если количество выборок меньше заданного значения, aqo не будет делать предсказаний. Слишком большое значение может повлиять на производительность, а слишком маленькое — снизить качество предсказания.

По умолчанию: `3`.

`aqo.predict_with_few_neighbors` (`boolean`)

Позволяет aqo делать предсказания с меньшим количеством соседей, чем указано в параметре aqo.min_neighbors_for_predicting. Если установлено значение `off`, aqo обучается, но не делает предсказания до тех пор, пока счётчик выполнений запроса с разными константами не достигнет 3 (по умолчанию для `aqo.min_neighbors_for_predicting`).

По умолчанию: `on` (вкл.).

`aqo.fs_max_items` (`integer`)

Определяет максимальное количество пространств признаков, с которыми может работать aqo. При превышении этого количества обучение на новых классах запросов прекратится, и они не будут отображаться в представлениях. Задать этот параметр можно только при запуске сервера.

По умолчанию: `10000`.

`aqo.fss_max_items` (`integer`)

Определяет максимальное количество подпространств признаков, с которыми может работать aqo. При превышении этого количества данные об избирательности и предсказание количества строк для новых узлов плана запроса больше не будут собираться и новые подпространства признаков не будут отображаться в представлении aqo_data. Задать этот параметр можно только при запуске сервера.

По умолчанию: `100000`.

`aqo.querytext_max_size` (`integer`)

Определяет максимальный размер запроса в представлении aqo_query_texts. Задать этот параметр можно только при запуске сервера.

По умолчанию: `1000`.

`aqo.dsm_size_max` (`integer`)

Определяет максимальный размер динамической разделяемой памяти в мегабайтах, которую модуль aqo может выделить для хранения данных обучения и текстов запросов. Если установленное значение меньше объёма сохранённых данных aqo, сервер не запустится. Задать этот параметр можно только при запуске сервера.

По умолчанию: `100`.

`aqo.wal_rw` (`boolean`)

Включает физическую репликацию и обеспечивает полное восстановление данных aqo после сбоя. При значении `off` на ведущем сервере данные на реплику не передаются. При значении `off` на реплике данные, передаваемые с ведущего сервера, игнорируются. В таком случае при сбое сервера данные могут быть восстановлены только до последней контрольной точки. Этот параметр можно задать только при запуске сервера.

По умолчанию: `on` (вкл.).

`aqo.sandbox` (`boolean`)

Позволяет резервировать отдельную область в общей памяти для использования ведущим или ведомым узлом, что позволяет собирать и использовать статистику с данными в этой области памяти. Если включён на ведомом узле, расширение использует отдельную область общей памяти, которая не реплицируется на ведомый сервер. Изменение значения этого параметра сбрасывает кеш aqo. Изменить этот параметр могут только суперпользователи.

По умолчанию: `off` (выкл.).

`aqo.delta_rows` (`boolean`)

Включает механизм обучения, при котором aqo корректирует оценку количества строк планировщика своими предсказаниями. Если параметр выключен, aqo использует собственные предсказания.

По умолчанию: `off` (выкл.).

F.2.4.2. Представления

F.2.4.2.1. `aqo_query_texts`

В представлении `aqo_query_texts` классифицируются все классы запросов, обрабатываемые aqo. Для каждого класса запросов в представлении отображается текст первого проанализированного запроса этого класса.

Таблица F.1. Представление `aqo_query_texts`

Имя столбца	Описание
`queryid`	Уникальный идентификатор запроса.
`dbid`	Идентификатор базы данных.
`fs`	Идентификатор пространства признаков.

Имя столбца	Описание
`query_text`	Текст первого проанализированного запроса данного класса. Длина текста запроса ограничивается параметром aqo.querytext_max_size.

F.2.4.2.2. `aqo_queries`

В представлении `aqo_queries` отображаются свойства оптимизации для разных классов запросов. Один запрос, выполненный в двух разных базах данных, сохраняется дважды с одинаковым идентификатором запроса (`fs`).

Таблица F.2. Представление `aqo_queries`

Свойство	Описание
`fs`	Идентификатор пространства признаков.
`dbid`	Идентификатор базы данных, в которой выполнялся запрос.
`learn_aqo`	Показывает, включён ли сбор статистики для данного класса запросов.
`use_aqo`	Показывает, включено ли предсказание количества строк средствами aqo для следующего выполнения данного класса запросов.
`auto_tuning`	Показывает, может ли aqo динамически изменять параметры `use_aqo` и `learn_aqo` для этого класса запросов. По умолчанию для новых запросов устанавливается значение `true`, если включён параметр aqo.advanced и режим aqo.mode = `intelligent`. При включённом свойстве `auto_tuning` , если для нескольких последовательных выполнений запроса с выключенным `use_aqo` ошибка оценки количества строк остаётся достаточно малой и стабильной, aqo включает `use_aqo` . Запросы с `learn_aqo` =true (все новые запросы) несколько раз выполняются с использованием aqo и без него, со стандартным планировщиком. Чем быстрее запрос выполняется с aqo, тем вероятнее, что модуль будет использоваться для следующих запросов. Если после нескольких выполнений окажется, что запросы с использованием aqo выполняются дольше, чем с использованием стандартного планировщика, aqo больше не будет использоваться для этого класса запросов: для `auto_tuning` , `use_aqo` и `learn_aqo` устанавливается значение `off`.
`smart_timeout`	Значение «умного» тайм-аута операторов для данного класса запросов. Начальное значение такого тайм-аута для любого запроса определяется параметром конфигурации statement_timeout.
`count_increase_timeout`	Показывает, сколько раз увеличивался «умный» тайм-аут операторов для данного класса запросов.

F.2.4.2.3. `aqo_data`

В представлении `aqo_data` отображаются данные машинного обучения для уточнения оценки количества строк. Количество строк ограничено параметром aqo.fss_max_items. Чтобы сбросить всю собранную статистику для определённого класса запросов, можно удалить из представления `aqo_data` все строки с соответствующим `fs`.

Таблица F.3. Представление `aqo_data`

Данные	Описание
`fs`	Идентификатор (хеш) пространства признаков.
`fss`	Идентификатор (хеш) подпространства признаков.
`dbid`	Идентификатор базы данных.
`delta_rows`	Если true, aqo делает предсказания на основе оценки планировщика; в противном случае `false`.
`nfeatures`	Размер подпространства признаков для узла плана запроса.
`features`	Логарифм избирательности, на котором основано предсказание количества строк.
`targets`	Логарифм количества строк для узла плана запроса.
`reliability`	Уровень достоверности статистики обучения: • 1 (по умолчанию) — указывает, что данные получены после нормального выполнения запроса • 0,1 — указывает, что данные получены от частично выполненного узла (они не нужны, так как ненадёжны) • 0,9 — указывает, что данные получены от готового узла, но от частично выполненного оператора
`oids`	Список идентификаторов таблиц, которые участвовали в предсказании для этого узла.
`tmpoids`	Список идентификаторов временных таблиц, которые участвовали в предсказании для этого узла.

F.2.4.2.4. `aqo_query_stat`

В представлении `aqo_query_stat` отображается статистика выполнения запросов, группируемая по классам запросов. Расширение aqo использует эти данные, когда для определённого класса запросов включено свойство `auto_tuning`.

Таблица F.4. Представление `aqo_query_stat`

Данные	Описание
`fs`	Идентификатор пространства признаков.
`dbid`	Идентификатор базы данных.
`execution_time_with_aqo`	Массив значений времени выполнения запросов со включённым aqo.
`execution_time_without_aqo`	Массив значений времени выполнения запросов с отключённым aqo.

Данные	**Описание**
`planning_time_with_aqo`	Массив значений времени планирования запросов со включённым aqo.
`planning_time_without_aqo`	Массив значений времени планирования запросов с отключённым aqo.
`cardinality_error_with_aqo`	Массив ошибок оценки количества строк со включённым aqo в планах выбранных запросов.
`cardinality_error_without_aqo`	Массив ошибок оценки количества строк с отключённым aqo в планах выбранных запросов.
`executions_with_aqo`	Число запросов, выполненных со включённым aqo.
`executions_without_aqo`	Число запросов, выполненных с отключённым aqo.

F.2.4.3. Функции

Модуль aqo добавляет несколько функций в каталог Postgres Pro.

F.2.4.3.1. Функции управления хранилищем

> **Важно**
>
> Функции `aqo_queries_update`, `aqo_query_texts_update`, `aqo_query_stat_update`, `aqo_data_update` и `aqo_data_delete` изменяют файлы данных, на которых основаны соответствующие представления aqo. Поэтому вызывайте эти функции только в том случае, если вы понимаете логику адаптивной оптимизации запросов.

`aqo_cleanup()` → `setof integer`

Удаляет данные, относящиеся к классам запросов, которые связаны (возможно частично) с удалёнными отношениями. Возвращает количество удалённых пространств признаков (классов) и подпространств признаков. Игнорирует удаление других объектов.

`aqo_enable_class` (*`fs`* bigint, *`dbid`* oid) → `void`

Устанавливает для `learn_aqo`, `use_aqo` и `auto_tuning` (только в режиме `intelligent`) значение true для класса запросов с указанными *`fs`* и *`dbid`*. Можно использовать для *`dbid`* значение NULL вместо идентификатора текущей базы данных.

`aqo_disable_class` (*`fs`* bigint, *`dbid`* oid) → `void`

Устанавливает для `learn_aqo`, `use_aqo` и `auto_tuning` значение false для класса запросов с указанными *`fs`* и *`dbid`*. Можно использовать для *`dbid`* значение NULL вместо идентификатора текущей базы данных.

`aqo_drop_class` (*`fs`* bigint, *`dbid`* oid) → `integer`

Удаляет все данные, относящиеся к заданному классу запросов и базе данных, из хранилища aqo. Для параметра *`dbid`* можно задать значение NULL вместо идентификатора текущей базы данных. Возвращает количество записей, удалённых из хранилища aqo.

`aqo_reset` (*`dbid`* oid) → `bigint`

Удаляет записи из указанной базы данных: данные машинного обучения, тексты запросов, статистику и свойства классов запросов. Если параметр *`dbid`* не указан, данные удаляются из текущей базы данных. Если *`dbid`* имеет значение NULL, удаляются все записи из хранилища aqo. Возвращает количество удалённых записей.

`aqo_queries_update` (*fs* `bigint`, *dbid* `oid`, *learn_aqo* `boolean`, *use_aqo* `boolean`, *auto_tuning* `boolean`) → `boolean`

Изменяет или вставляет запись в файл данных, лежащий в основе представления aqo_queries, для указанных *fs* и *dbid*. Вместо идентификатора текущей базы данных в *dbid* можно указать NULL. Значения NULL для остальных параметров означают, что их следует оставить без изменений. Обратите внимание, что записи с нулевым значением *fs* или *dbid* не могут быть изменены. Возвращает `false` в случае ошибки и `true` в противном случае.

`aqo_query_texts_update` (*fs* `bigint`, *dbid* `oid`, *query_text* `text`) → `boolean`

Изменяет или вставляет запись в файл данных, лежащий в основе представления aqo_query_texts, для указанных *fs* и *dbid*. Можно использовать для *dbid* значение NULL вместо идентификатора текущей базы данных. Обратите внимание, что записи с нулевым значением *fs* или *dbid* не могут быть изменены. Возвращает `false` в случае ошибки и `true` в противном случае.

`aqo_query_stat_update` (*fs* `bigint`, *dbid* `oid`, *execution_time_with_aqo* `double precision[]`, *execution_time_without_aqo* `double precision[]`, *planning_time_with_aqo* `double precision[]`, *planning_time_without_aqo* `double precision[]`, *cardinality_error_with_aqo* `double precision[]`, *cardinality_error_without_aqo* `double precision[]`, *executions_with_aqo* `bigint[]`, *executions_without_aqo* `bigint[]`) → `boolean`

Изменяет или вставляет запись в файл данных, лежащий в основе представления aqo_query_stat, для указанных *fs* и *dbid*. Записи для обобщенного пространства признаков, с нулевым значением *fs* или *dbid*, не могут быть изменены. Можно использовать для *dbid* значение NULL вместо идентификатора текущей базы данных. Возвращает `false` в случае ошибки и `true` в противном случае.

`aqo_data_update` (*fs* `bigint`, *fss* `integer`, *dbid* `oid`, *delta_rows* `boolean`, *nfeatures* `integer`, *features* `double precision[][]`, *targets* `double precision[]`, *reliability* `double precision[]`, *oids* `oid[]`, *tmpoids* `oid[]`) → `boolean`

Изменяет или вставляет запись в файл данных, лежащий в основе представления aqo_data, для указанных *fs*, *fss* и *dbid*. Можно использовать для *dbid* значение NULL вместо идентификатора текущей базы данных. Если для *delta_rows* задать NULL, будет использовано значение параметра конфигурации `aqo.delta_rows`. Возвращает `false` в случае ошибки и `true` в противном случае.

`aqo_data_delete` (*fs* `bigint`, *fss* `integer`, *dbid* `oid`, *delta_rows* `boolean`) → `boolean`

Удаляет запись из файла данных, лежащего в основе представления aqo_data, для указанных *fs*, *fss* и *dbid*. Можно использовать для *dbid* значение NULL вместо идентификатора текущей базы данных. Если не указать параметр *delta_rows* или задать для него NULL, будет использовано значение параметра конфигурации `aqo.delta_rows`. Возвращает `false` в случае ошибки и `true` в противном случае.

F.2.4.3.2. Функции управления памятью

`aqo_memory_usage` () → `setof record`

Отображает выделенные и использованные размеры контекстов памяти и хеш-таблиц aqo. Возвращает таблицу со следующими столбцами:

`name`

Краткое описание контекста памяти или хеш-таблицы

`allocated_size`

Общий размер выделенной памяти

`used_size`

Размер текущей используемой памяти

F.2.4.3.3. Функции для аналитики

`aqo_cardinality_error` (*`controlled`* `boolean`) → `setof record`

Показывает ошибку оценки количества строк для последнего выполнения запроса. Если *`controlled`* имеет значение `true`, показывает запросы, выполненные с включённым aqo. Если *`controlled`* имеет значение `false`, показывает запросы, выполненные с отключённым aqo, но имеющие накопленную статистику. Возвращает таблицу со следующими столбцами:

`num`

Порядковый номер

`dbid`

Идентификатор базы данных

`fs`

Идентификатор пространства признаков. Может содержать ноль или основное хеш-значение.

`error`

Погрешность aqo, рассчитываемая на узлах планов запросов

`nexecs`

Количество выполнений запросов, связанных с данным `fs`

`aqo_execution_time` (*`controlled`* `boolean`) → `setof record`

Показывает время выполнения запросов. Если *`controlled`* имеет значение true, показывает время выполнения последнего запроса с включённым aqo. Если *`controlled`* имеет значение false, возвращает среднее время выполнения запроса для всех записанных в журнал выполнений с отключённым aqo. Время выполнения без aqo можно собрать, если параметр aqo.mode = `intelligent` или параметр aqo.force_collect_stat = `on`. Возвращает таблицу со следующими столбцами:

`num`

Порядковый номер

`dbid`

Идентификатор базы данных

`fs`

Идентификатор пространства признаков. Может содержать ноль или основное хеш-значение.

`exec_time`

Если *`controlled`* имеет значение true, показывает время выполнения последнего запроса с включённым aqo, в противном случае — среднее время выполнения запроса для всех записанных в журнал выполнений с отключённым aqo.

`nexecs`

Количество выполнений запросов, связанных с данным `fs`

F.2.5. Примеры

Пример F.1. Обучение на запросе (Основной режим)

Рассмотрим оптимизацию запроса с использованием расширения aqo.

Когда запрос выполняется в первый раз, его нет в таблицах, лежащих в основе представлений aqo. Таким образом, данных для предсказания aqo для каждого узла плана нет, и в выводе `EXPLAIN` появляются строки «AQO not used»:

```
demo=# EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT bp.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
WHERE f.scheduled_departure > '2017-08-15 15:00:00+00';
                                                          QUERY PLAN
--------------------------------------------------------------------------------------------
 Hash Join  (cost=63796.33..140286.73 rows=477728 width=25) (actual rows=2215 loops=1)
   AQO not used, fss=8598194613120045129
   Hash Cond: ((bp.flight_id = f.flight_id) AND (bp.ticket_no = tf.ticket_no))
   ->  Seq Scan on boarding_passes bp  (cost=0.00..32894.95 rows=1894295 width=25)
 (actual rows=1894295 loops=1)
         AQO not used, fss=1362775811343989307
   ->  Hash  (cost=51379.43..51379.43 rows=595260 width=22) (actual rows=468255
 loops=1)
         Buckets: 131072  Batches: 8  Memory Usage: 4223kB
         ->  Hash Join  (cost=1859.80..51379.43 rows=595260 width=22) (actual
 rows=468255 loops=1)
               AQO not used, fss=-7651474063207585780
               Hash Cond: (tf.flight_id = f.flight_id)
               ->  Seq Scan on ticket_flights tf  (cost=0.00..43323.35 rows=2360335
 width=18) (actual rows=2360335 loops=1)
                     AQO not used, fss=-6410966714754547713
               ->  Hash  (cost=1652.80..1652.80 rows=16560 width=4) (actual rows=16340
 loops=1)
                     Buckets: 32768  Batches: 1  Memory Usage: 831kB
                     ->  Seq Scan on flights f  (cost=0.00..1652.80 rows=16560 width=4)
 (actual rows=16340 loops=1)
                           AQO not used, fss=-1289471166524579716
                           Filter: (scheduled_departure > '2017-08-15
 22:00:00+07'::timestamp with time zone)
                           Rows Removed by Filter: 49324
 Using aqo: true
 AQO mode: LEARN
 AQO advanced: OFF
 Query hash: 402936317627943257
 JOINS: 2
(23 rows)
```

Если в представлении aqo_data нет информации об определённом узле, aqo добавит в него соответствующую запись для дальнейшего изучения и предсказания, за исключением узлов с `fss=0` в выводе `EXPLAIN`. Поскольку значения в полях `features` и `targets` в представлении `aqo_data` являются логарифмом по основанию `e`, чтобы получить фактическое значение, возведите `e` в соответствующую степень. Например: exp(7.703007682479236):

```
 demo=# select * from aqo_data;
 fs |         fss          | dbid  | delta_rows | nfeatures |                  features
                 |       targets        | reliability |       oids          | tmpoids
----+----------------------+-------+------------+-----------
+---------------------------------------------+----------------------+-------------
+---------------------+-------
  0 |  1362775811343989307 | 16429 | f          |         0 |
                 | {14.454357295615447} | {1}         | {16452}             |
  0 | -6410966714754547713 | 16429 | f          |         0 |
                 | {14.674314116080508} | {1}         | {16479}             |
```

```
  0 | -1289471166524579716 | 16429 | f          |         1 | {{-1.3775704575284085}}
                | {9.701371368413994}  | {1}         | {16458}            |
  0 |  8598194613120045129 | 16429 | f          |         2 |
 {{-13.492416828684513,-1.3775704575284085}} | {7.703007682479236}  | {1}         |
 {16479,16458,16452} |
  0 | -7651474063207585780 | 16429 | f          |         2 |
 {{-11.092306109090387,-1.3775704575284085}} | {13.056768298305919} | {1}         |
 {16479,16458}       |
(5 rows)
```

При повторном выполнении запроса aqo распознаёт его и делает предсказание. Обратите внимание на оценку количества строк, предсказанную aqo, и значение погрешности aqo («error=0%»).

```
demo=# EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT bp.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
WHERE f.scheduled_departure > '2017-08-15 15:00:00+00';
                                                         QUERY PLAN
-------------------------------------------------------------------------------------------------
 Hash Join  (cost=61144.50..136890.86 rows=2215 width=25) (actual rows=2215 loops=1)
   AQO: rows=2215, error=0%, fss=8598194613120045129
   Hash Cond: ((bp.flight_id = f.flight_id) AND (bp.ticket_no = tf.ticket_no))
   ->  Seq Scan on boarding_passes bp  (cost=0.00..32894.95 rows=1894295 width=25)
 (actual rows=1894295 loops=1)
         AQO: rows=1894295, error=0%, fss=1362775811343989307
   ->  Hash  (cost=51376.68..51376.68 rows=468255 width=22) (actual rows=468255
 loops=1)
         Buckets: 131072  Batches: 4  Memory Usage: 7438kB
         ->  Hash Join  (cost=1857.05..51376.68 rows=468255 width=22) (actual
 rows=468255 loops=1)
               AQO: rows=468255, error=0%, fss=-7651474063207585780
               Hash Cond: (tf.flight_id = f.flight_id)
               ->  Seq Scan on ticket_flights tf  (cost=0.00..43323.35 rows=2360335
 width=18) (actual rows=2360335 loops=1)
                     AQO: rows=2360335, error=0%, fss=-6410966714754547713
               ->  Hash  (cost=1652.80..1652.80 rows=16340 width=4) (actual rows=16340
 loops=1)
                     Buckets: 16384  Batches: 1  Memory Usage: 703kB
                     ->  Seq Scan on flights f  (cost=0.00..1652.80 rows=16340 width=4)
 (actual rows=16340 loops=1)
                           AQO: rows=16340, error=0%, fss=-1289471166524579716
                           Filter: (scheduled_departure > '2017-08-15
 22:00:00+07'::timestamp with time zone)
                           Rows Removed by Filter: 49324
 Using aqo: true
 AQO mode: LEARN
 AQO advanced: OFF
 Query hash: 4029363176279432 57
 JOINS: 2
(23 rows)
```

Изменив константу в запросе, можно заметить, что предсказание сделано с ошибкой:

```
demo=# EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT bp.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
```

```
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
WHERE f.scheduled_departure > '2017-08-10 15:00:00+00';
                                                         QUERY PLAN
---------------------------------------------------------------------------------------------
 Hash Join  (cost=61144.50..136890.86 rows=2215 width=25) (actual rows=111397 loops=1)
   AQO: rows=2215, error=-4929%, fss=8598194613120045129
   Hash Cond: ((bp.flight_id = f.flight_id) AND (bp.ticket_no = tf.ticket_no))
   ->  Seq Scan on boarding_passes bp  (cost=0.00..32894.95 rows=1894295 width=25)
 (actual rows=1894295 loops=1)
         AQO: rows=1894295, error=0%, fss=1362775811343989307
   ->  Hash  (cost=51376.68..51376.68 rows=468255 width=22) (actual rows=577437
 loops=1)
         Buckets: 131072 (originally 131072)  Batches: 8 (originally 4)  Memory Usage:
 7169kB
         ->  Hash Join  (cost=1857.05..51376.68 rows=468255 width=22) (actual
 rows=577437 loops=1)
               AQO: rows=468255, error=-23%, fss=-7651474063207585780
               Hash Cond: (tf.flight_id = f.flight_id)
               ->  Seq Scan on ticket_flights tf  (cost=0.00..43323.35 rows=2360335
 width=18) (actual rows=2360335 loops=1)
                     AQO: rows=2360335, error=0%, fss=-6410966714754547713
               ->  Hash  (cost=1652.80..1652.80 rows=16340 width=4) (actual rows=19040
 loops=1)
                     Buckets: 32768 (originally 16384)  Batches: 1 (originally 1)
 Memory Usage: 926kB
                     ->  Seq Scan on flights f  (cost=0.00..1652.80 rows=16340 width=4)
 (actual rows=19040 loops=1)
                           AQO: rows=16340, error=-17%, fss=-1289471166524579716
                           Filter: (scheduled_departure > '2017-08-10
 22:00:00+07'::timestamp with time zone)
                           Rows Removed by Filter: 46624
 Using aqo: true
 AQO mode: LEARN
 AQO advanced: OFF
 Query hash: 402936317627943257
 JOINS: 2
(23 rows)
```

Однако вместо пересчёта полей `features` и `targets`, aqo добавил новые значения избирательности и оценки количества строк для этого запроса в `aqo_data`:

```
demo=# select * from aqo_data;
 fs |         fss          | dbid  | delta_rows | nfeatures |
             features                                         |                 targets
                | reliability |        oids          | tmpoids
----+----------------------+-------+------------+-----------
+----------------------------------------------------------------------------------------
+-----------------------------------------+-------------+---------------------
+---------
  0 |  1362775811343989307 | 16429 | f          |         0 |
                                                              | {14.454357295615447}
                | {1}         | {16452}              |
  0 | -6410966714754547713 | 16429 | f          |         0 |
                                                              | {14.67431416080508}
                | {1}         | {16479}              |
  0 | -1289471166524579716 | 16429 | f          |         1 | {{-1.3775704575284085},
{-1.2287385463463019}}                                          |
 {9.701371368413994,9.854297308345357}   | {1,1}       | {16458}             |
```

```
  0 |  8598194613120045129 | 16429 | f          |         2 |
 {{-13.492416828684513,-1.3775704575284085},{-13.492416828684513,-1.2287385463463019}}
 | {7.703007682479236,11.620855676130656}  | {1,1}       | {16479,16458,16452} |
  0 | -7651474063207585780 | 16429 | f          |         2 |
 {{-11.092306109090387,-1.3775704575284085},{-11.092306109090387,-1.2287385463463019}}
 | {13.056768298305919,13.266354624518149} | {1,1}       | {16479,16458}       |
(5 rows)
```

Теперь в предсказании есть небольшая погрешность примерно в 3%, которая может объясняться погрешностью вычислений:

```
demo=# EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT bp.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
WHERE f.scheduled_departure > '2017-08-10 15:00:00+00';
                                                         QUERY PLAN
----------------------------------------------------------------------------------------------------
 Hash Join  (cost=63438.72..139820.11 rows=108541 width=25) (actual rows=111397
 loops=1)
   AQO: rows=108541, error=-3%, fss=8598194613120045129
   Hash Cond: ((bp.flight_id = f.flight_id) AND (bp.ticket_no = tf.ticket_no))
   ->  Seq Scan on boarding_passes bp  (cost=0.00..32894.95 rows=1894295 width=25)
 (actual rows=1894295 loops=1)
         AQO: rows=1894295, error=0%, fss=1362775811343989307
   ->  Hash  (cost=51410.19..51410.19 rows=576635 width=22) (actual rows=577437
 loops=1)
         Buckets: 131072  Batches: 8  Memory Usage: 4966kB
         ->  Hash Join  (cost=1890.56..51410.19 rows=576635 width=22) (actual
 rows=577437 loops=1)
               AQO: rows=576635, error=-0%, fss=-7651474063207585780
               Hash Cond: (tf.flight_id = f.flight_id)
               ->  Seq Scan on ticket_flights tf  (cost=0.00..43323.35 rows=2360335
 width=18) (actual rows=2360335 loops=1)
                     AQO: rows=2360335, error=0%, fss=-6410966714754547713
               ->  Hash  (cost=1652.80..1652.80 rows=19021 width=4) (actual rows=19040
 loops=1)
                     Buckets: 32768  Batches: 1  Memory Usage: 926kB
                     ->  Seq Scan on flights f  (cost=0.00..1652.80 rows=19021 width=4)
 (actual rows=19040 loops=1)
                           AQO: rows=19021, error=-0%, fss=-1289471166524579716
                           Filter: (scheduled_departure > '2017-08-10
 22:00:00+07'::timestamp with time zone)
                           Rows Removed by Filter: 46624
 Using aqo: true
 AQO mode: LEARN
 AQO advanced: OFF
 Query hash: 402936317627943257
 JOINS: 2
(23 rows)
```

Можно изменить запрос, добавив некоторую таблицу в список `JOIN`. В этом случае aqo будет прогнозировать оценку количества строк на узлах, использовавшихся для обучения.

```
demo=# EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT t.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
```

```
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
JOIN tickets t ON t.ticket_no = tf.ticket_no
WHERE f.scheduled_departure > '2017-08-15 15:00:00+00';
                                                           QUERY PLAN
----------------------------------------------------------------------------------------------------
 Nested Loop  (cost=61158.91..134296.78 rows=2273 width=104) (actual rows=2215 loops=1)
   AQO not used, fss=-8581941154270057231
   ->  Hash Join  (cost=61158.48..133208.83 rows=2273 width=28) (actual rows=2215
 loops=1)
         AQO: rows=2273, error=3%, fss=8598194613120045129
         Hash Cond: ((bp.flight_id = f.flight_id) AND (bp.ticket_no = tf.ticket_no))
         ->  Seq Scan on boarding_passes bp  (cost=0.00..32894.95 rows=1894295
 width=18) (actual rows=1894295 loops=1)
               AQO: rows=1894295, error=0%, fss=1362775811343989307
         ->  Hash  (cost=51376.89..51376.89 rows=468906 width=22) (actual rows=468255
 loops=1)
               Buckets: 131072  Batches: 4  Memory Usage: 7438kB
               ->  Hash Join  (cost=1857.26..51376.89 rows=468906 width=22) (actual
 rows=468255 loops=1)
                     AQO: rows=468906, error=0%, fss=-7651474063207585780
                     Hash Cond: (tf.flight_id = f.flight_id)
                     ->  Seq Scan on ticket_flights tf  (cost=0.00..43323.35
 rows=2360335 width=18) (actual rows=2360335 loops=1)
                           AQO: rows=2360335, error=0%, fss=-6410966714754547713
                     ->  Hash  (cost=1652.80..1652.80 rows=16357 width=4) (actual
 rows=16340 loops=1)
                           Buckets: 16384  Batches: 1  Memory Usage: 703kB
                           ->  Seq Scan on flights f  (cost=0.00..1652.80 rows=16357
 width=4) (actual rows=16340 loops=1)
                                 AQO: rows=16357, error=0%, fss=-1289471166524579716
                                 Filter: (scheduled_departure > '2017-08-15
 22:00:00+07'::timestamp with time zone)
                                 Rows Removed by Filter: 49324
   ->  Index Scan using tickets_pkey on tickets t  (cost=0.42..0.48 rows=1 width=104)
 (actual rows=1 loops=2215)
         AQO not used, fss=2731022528523952664
         Index Cond: (ticket_no = bp.ticket_no)
 Using aqo: true
 AQO mode: LEARN
 AQO advanced: OFF
 Query hash: 7809046947949890015
 JOINS: 3
(28 rows)
```

Пример F.2. Использование представления aqo_query_stat

В представлении `aqo_query_stat` отображается статистика времени планирования запросов, времени выполнения запросов и ошибок оценки количества строк. На основании этих данных вы можете принимать решения об использовании предсказаний aqo для различных классов запросов.

Обратимся к представлению `aqo_query_stats`:

```
demo=# SELECT * FROM aqo_query_stat \gx
-[ RECORD 1 ]------------------
+---------------------------------------------------------------
fs                             | 7809046947949890015
dbid                           | 16429
execution_time_with_aqo        | {1.039218233,0.925258453,0.831166925,0.779602353}
execution_time_without_aqo     | {1.022052611,0.936486619}
planning_time_with_aqo         | {0.003305339,0.002129048,0.002538877,0.002142972}
```

```
planning_time_without_aqo     | {0.000767553,0.000711208}
cardinality_error_with_aqo    |
 {0.4854215265638894,0,1.1711726076352047,0.007732205169478082}
cardinality_error_without_aqo | {0.4854215265638894,1.571562511977072}
executions_with_aqo           | 4
executions_without_aqo        | 2
```

Полученные данные относятся к запросу, рассмотренному в примере Пример F.1. Этот запрос выполнялся с каждым из параметров `f.scheduled_departure > '2017-08-10 15:00:00+00'` и `f.scheduled_departure > '2017-08-15 15:00:00+00'` по одному разу без aqo и по два раза с aqo. Видно, что с aqo погрешность оценки количества строк уменьшается до 0,0077, а минимальная погрешность оценки количества строк без aqo составляет 0,4854. Кроме того, время выполнения с aqo меньше, чем без него. Таким образом, можно сделать вывод, что aqo хорошо обучается на этом запросе и предсказание можно использовать для этого класса запросов.

Пример F.3. Использование расширенного режима aqo

Расширенный режим позволяет более гибко управлять расширением aqo. При включении данного режима, то есть

```
demo=# SET aqo.advanced = on;
```

aqo будет собирать данные машинного обучения отдельно для каждого выполняемого запроса. Например:

```
demo=# EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT bp.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
WHERE f.scheduled_departure > '2017-08-15 15:00:00+00';
                                                         QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Join  (cost=63796.33..140286.73 rows=477728 width=25) (actual rows=2215 loops=1)
   AQO not used, fss=8598194613120045129
   Hash Cond: ((bp.flight_id = f.flight_id) AND (bp.ticket_no = tf.ticket_no))
   ->  Seq Scan on boarding_passes bp  (cost=0.00..32894.95 rows=1894295 width=25)
 (actual rows=1894295 loops=1)
         AQO not used, fss=1362775811343989307
   ->  Hash  (cost=51379.43..51379.43 rows=595260 width=22) (actual rows=468255
 loops=1)
         Buckets: 131072  Batches: 8  Memory Usage: 4223kB
         ->  Hash Join  (cost=1859.80..51379.43 rows=595260 width=22) (actual
 rows=468255 loops=1)
               AQO not used, fss=-7651474063207585780
               Hash Cond: (tf.flight_id = f.flight_id)
               ->  Seq Scan on ticket_flights tf  (cost=0.00..43323.35 rows=2360335
 width=18) (actual rows=2360335 loops=1)
                     AQO not used, fss=-6410966714754547713
               ->  Hash  (cost=1652.80..1652.80 rows=16560 width=4) (actual rows=16340
 loops=1)
                     Buckets: 32768  Batches: 1  Memory Usage: 831kB
                     ->  Seq Scan on flights f  (cost=0.00..1652.80 rows=16560 width=4)
 (actual rows=16340 loops=1)
                           AQO not used, fss=-1289471166524579716
                           Filter: (scheduled_departure > '2017-08-15
 22:00:00+07'::timestamp with time zone)
                           Rows Removed by Filter: 49324
 Using aqo: true
 AQO mode: LEARN
 AQO advanced: ON
```

```
 Query hash: 402936317627943257
 JOINS: 2
(23 rows)
```

Теперь этот запрос хранится в aqo_data с ненулевым `fs` (по умолчанию `fs` равен хешу запроса):

```
demo=# SELECT * FROM aqo_data;
         fs         |         fss          | dbid  | delta_rows | nfeatures |
        features                   |        targets        | reliability |        oids
       | tmpoids
--------------------+----------------------+-------+------------+-----------
+---------------------------------------------+----------------------+-------------
+---------------------+---------
 402936317627943257 | -6410966714754547713 | 16429 | f          |         0 |
                                   | {14.674314116080508} | {1}         | {16479}
       |
 402936317627943257 |  1362775811343989307 | 16429 | f          |         0 |
                                   | {14.454357295615447} | {1}         | {16452}
       |
 402936317627943257 | -1289471166524579716 | 16429 | f          |         1 |
 {{-1.3775704575284085}}                     | {9.701371368413994}  | {1}         |
 {16458}             |
 402936317627943257 |  8598194613120045129 | 16429 | f          |         2 |
 {{-13.492416828684513,-1.3775704575284085}} | {7.703007682479236}  | {1}         |
 {16479,16458,16452} |
 402936317627943257 | -7651474063207585780 | 16429 | f          |         2 |
 {{-11.092306109090387,-1.3775704575284085}} | {13.056768298305919} | {1}         |
 {16479,16458}       |
(5 rows)
```

Можно настроить несколько параметров только для данного запроса. Это значения `learn_aqo`, `use_aqo` и `auto_tuning` в представлении aqo_queries:

```
demo=# SELECT * FROM aqo_queries;
         fs         | dbid  | learn_aqo | use_aqo | auto_tuning | smart_timeout |
 count_increase_timeout
--------------------+-------+-----------+---------+-------------+---------------
+------------------------
 402936317627943257 | 16429 | t         | t       | f           |             0 |
                0
                  0 |     0 | f         | f       | f           |             0 |
                0
(2 rows)
```

Зададим для `use_aqo` значение `false`:

```
demo=# SELECT aqo_queries_update(402936317627943257, NULL, NULL, false, NULL);
 aqo_queries_update
--------------------
 t
(1 row)
```

Теперь изменим константу в запросе:

```
demo=# EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT bp.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
WHERE f.scheduled_departure > '2017-08-10 15:00:00+00';
                                                         QUERY PLAN
```

```
----------------------------------------------------------------------------------------------------
 Hash Join  (cost=65822.16..142872.60 rows=554378 width=25) (actual rows=111397
 loops=1)
   AQO not used, fss=0
   Hash Cond: ((bp.flight_id = f.flight_id) AND (bp.ticket_no = tf.ticket_no))
   ->  Seq Scan on boarding_passes bp  (cost=0.00..32894.95 rows=1894295 width=25)
 (actual rows=1894295 loops=1)
         AQO not used, fss=0
   ->  Hash  (cost=51412.64..51412.64 rows=690768 width=22) (actual rows=577437
 loops=1)
         Buckets: 131072  Batches: 8  Memory Usage: 4966kB
         ->  Hash Join  (cost=1893.01..51412.64 rows=690768 width=22) (actual
 rows=577437 loops=1)
               AQO not used, fss=0
               Hash Cond: (tf.flight_id = f.flight_id)
               ->  Seq Scan on ticket_flights tf  (cost=0.00..43323.35 rows=2360335
 width=18) (actual rows=2360335 loops=1)
                     AQO not used, fss=0
               ->  Hash  (cost=1652.80..1652.80 rows=19217 width=4) (actual rows=19040
 loops=1)
                     Buckets: 32768  Batches: 1  Memory Usage: 926kB
                     ->  Seq Scan on flights f  (cost=0.00..1652.80 rows=19217 width=4)
 (actual rows=19040 loops=1)
                           AQO not used, fss=0
                           Filter: (scheduled_departure > '2017-08-10
 22:00:00+07'::timestamp with time zone)
                           Rows Removed by Filter: 46624
 Using aqo: false
 AQO mode: LEARN
 AQO advanced: ON
 Query hash: 402936317627943257
 JOINS: 2
(23 rows)
```

aqo не использовался для этого запроса, но в представлении `aqo_data` появились новые данные:

```
demo=# SELECT * FROM aqo_data;
         fs         |         fss          | dbid  | delta_rows | nfeatures |
                             features                                      |
        targets                 | reliability |        oids         | tmpoids
--------------------+----------------------+-------+------------+-----------
+----------------------------------------------------------------------------------------
+-----------------------------------------+-------------+---------------------
+---------
 402936317627943257 | -6410966714754547713 | 16429 | f          |         0 |
                                                                                     |
 {14.674314116080508}                    | {1}         | {16479}             |
 402936317627943257 |  1362775811343989307 | 16429 | f          |         0 |
                                                                                     |
 {14.454357295615447}                    | {1}         | {16452}             |
 402936317627943257 | -1289471166524579716 | 16429 | f          |         1 |
 {{-1.3775704575284085},{-1.2287385463463019}}
 | {9.701371368413994,9.854297308345357}   | {1,1}       | {16458}             |
 402936317627943257 |  8598194613120045129 | 16429 | f          |         2 |
 {{-13.492416828684513,-1.3775704575284085},{-13.492416828684513,-1.2287385463463019}}
 | {7.703007682479236,11.620855676130656}  | {1,1}       | {16479,16458,16452} |
 402936317627943257 | -7651474063207585780 | 16429 | f          |         2 |
 {{-11.092306109090387,-1.3775704575284085},{-11.092306109090387,-1.2287385463463019}}
 | {13.056768298305919,13.266354624518149} | {1,1}       | {16479,16458}       |
```

```
(5 rows)
```

Установленное значение параметра `use_aqo` не относится к другим запросам. После выполнения другого запроса дважды видно, что aqo обучается на нём и делает для него предсказание:

```
EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT t.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
JOIN tickets t ON t.ticket_no = tf.ticket_no
WHERE f.scheduled_departure > '2017-08-15 15:00:00+00';
                                                            QUERY PLAN
---------------------------------------------------------------------------------------------
 Nested Loop  (cost=61144.93..134251.05 rows=2215 width=104) (actual rows=2215 loops=1)
   AQO: rows=2215, error=0%, fss=-8581941154270057231
   ->  Hash Join  (cost=61144.50..133190.86 rows=2215 width=28) (actual rows=2215
 loops=1)
         AQO: rows=2215, error=0%, fss=8598194613120045129
         Hash Cond: ((bp.flight_id = f.flight_id) AND (bp.ticket_no = tf.ticket_no))
         ->  Seq Scan on boarding_passes bp  (cost=0.00..32894.95 rows=1894295
 width=18) (actual rows=1894295 loops=1)
               AQO: rows=1894295, error=0%, fss=1362775811343989307
         ->  Hash  (cost=51376.68..51376.68 rows=468255 width=22) (actual rows=468255
 loops=1)
               Buckets: 131072  Batches: 4  Memory Usage: 7438kB
               ->  Hash Join  (cost=1857.05..51376.68 rows=468255 width=22) (actual
 rows=468255 loops=1)
                     AQO: rows=468255, error=0%, fss=-7651474063207585780
                     Hash Cond: (tf.flight_id = f.flight_id)
                     ->  Seq Scan on ticket_flights tf  (cost=0.00..43323.35
 rows=2360335 width=18) (actual rows=2360335 loops=1)
                           AQO: rows=2360335, error=0%, fss=-6410966714754547713
                     ->  Hash  (cost=1652.80..1652.80 rows=16340 width=4) (actual
 rows=16340 loops=1)
                           Buckets: 16384  Batches: 1  Memory Usage: 703kB
                           ->  Seq Scan on flights f  (cost=0.00..1652.80 rows=16340
 width=4) (actual rows=16340 loops=1)
                                 AQO: rows=16340, error=0%, fss=-1289471166524579716
                                 Filter: (scheduled_departure > '2017-08-15
 22:00:00+07'::timestamp with time zone)
                                 Rows Removed by Filter: 49324
   ->  Index Scan using tickets_pkey on tickets t  (cost=0.42..0.48 rows=1 width=104)
 (actual rows=1 loops=2215)
         AQO not used, fss=2731022528523952664
         Index Cond: (ticket_no = bp.ticket_no)
 Using aqo: true
 AQO mode: LEARN
 AQO advanced: ON
 Query hash: 7809046947949890015
 JOINS: 3
(28 rows)
```

Пример F.4. Использование режима «песочницы»

```
SET aqo.sandbox = ON;
SET aqo.enable = ON;
SET aqo.advanced = OFF;
-- Очистка базы знаний песочницы, не затрагивающая основные данные
SELECT aqo_reset();
```

```
EXPLAIN (ANALYZE, SUMMARY OFF, TIMING OFF)
SELECT t.*
FROM ticket_flights tf
JOIN flights f ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no AND bp.flight_id = tf.flight_id
JOIN tickets t ON t.ticket_no = tf.ticket_no
WHERE f.scheduled_departure > '2017-08-15 15:00:00+00';

-- Выполнение предыдущего запроса, пока планы не стабилизируются
...

-- Копирование данных, полученных из песочницы с aqo.advanced = OFF
CREATE TABLE aqo_data_sandbox AS SELECT * FROM aqo_data;
SET aqo.sandbox = OFF;
SELECT aqo_data_update (fs, fss, dbid, delta_rows, nfeatures, features, targets,
 reliability, oids, tmpoids)
FROM aqo_data_sandbox WHERE fs = 0;
DROP TABLE aqo_data_sandbox;
```

F.2.6. Автор

Олег Иванов

F.3. auth_delay — задержка при ошибке аутентификации

Модуль `auth_delay` добавляет небольшую задержку в процессе проверки подлинности перед тем, как выдаётся сообщение об ошибке, чтобы усложнить подбор паролей к базам данных. Заметьте, что это никоим образом не препятствует атакам типа «отказ в обслуживании», а даже наоборот, может помочь их осуществить, так как процессы, ожидающие сообщения об ошибке, всё равно занимают слоты подключения.

Чтобы эта функция работала, данный модуль нужно загрузить посредством параметра конфигурации shared_preload_libraries в `postgresql.conf`.

F.3.1. Параметры конфигурации

`auth_delay.milliseconds` (`integer`)

Число миллисекунд, которое нужно подождать, прежде чем сообщать об ошибке аутентификации. По умолчанию 0.

Эти параметры должны задаваться в `postgresql.conf`. Обычное использование выглядит так:

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.3.2. Автор

КайГай Кохэй `<kaigai@ak.jp.nec.com>`

F.4. auto_explain — протоколирование планов выполнения медленных запросов

Модуль `auto_explain` предоставляет возможность автоматического протоколирования планов выполнения медленных операторов, что позволяет обойтись без выполнения EXPLAIN вручную. Это особенно полезно для выявления неоптимизированных запросов в больших приложениях.

Этот модуль не предоставляет функций, доступных из SQL. Чтобы использовать его, просто загрузите его в процесс сервера. Это можно сделать в отдельном сеансе:

```
LOAD 'auto_explain';
```

(Для этого нужно быть суперпользователем.) Более типична конфигурация, когда он загружается в некоторые или все сеансы в результате включения `auto_explain` в переменную session_preload_libraries или в shared_preload_libraries в файле `postgresql.conf`. Загрузив этот модуль, вы можете отслеживать исключительно медленные запросы, вне зависимости от того, когда они происходят. Конечно, это имеет свою цену.

F.4.1. Параметры конфигурации

Есть несколько параметров конфигурации, которые управляют поведением `auto_explain`. Заметьте, что поведение по умолчанию сводится к бездействию, так что необходимо установить как минимум переменную `auto_explain.log_min_duration`, если вы хотите получить какие-либо результаты.

`auto_explain.log_min_duration` (`integer`)

Переменная `auto_explain.log_min_duration` задаёт время выполнения оператора, в миллисекундах, при превышении которого план оператора будет протоколироваться. При значении, равном `0`, протоколируются все планы, а при `-1` (по умолчанию) протоколирование планов отключается. Например, если вы установите значение `250ms`, протоколироваться будут все запросы, выполняющиеся 250 мс и дольше. Изменить этот параметр могут только суперпользователи.

`auto_explain.log_parameter_max_length` (`integer`)

`auto_explain.log_parameter_max_length` управляет протоколированием значений параметров запроса. Значение `-1` (по умолчанию) означает запись значений параметров полностью, а `0` отключает запись. Значение больше нуля усекает каждое значение параметра до указанного количества байтов. Изменить этот параметр могут только суперпользователи.

`auto_explain.log_analyze` (`boolean`)

При включении параметра `auto_explain.log_analyze` в протокол будет записываться вывод команды `EXPLAIN ANALYZE`, а не простой `EXPLAIN`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

> **Примечание**
>
> Когда этот параметр включён, замер времени на уровне узлов плана производится для всех операторов, даже если они выполняются недостаточно долго для протоколирования. Это может оказать крайне негативное влияние на производительность. Отключение `auto_explain.log_timing` исключает это влияние, но при этом собирается меньше информации.

`auto_explain.log_buffers` (`boolean`)

Параметр `auto_explain.log_buffers` определяет, будет ли при протоколировании плана выполнения выводиться статистика об использовании буферов; он равносилен указанию `BUFFERS` команды `EXPLAIN`. Этот параметр действует, только если включён параметр `auto_explain.log_analyze`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_wal` (boolean)

Параметр `auto_explain.log_wal` определяет, будет ли при протоколировании плана выполнения выводиться статистика об использовании WAL; он равносилен указанию `WAL` команды `EXPLAIN`. Этот параметр действует, только если включён параметр `auto_explain.log_analyze`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_timing` (boolean)

Параметр `auto_explain.log_timing` определяет, будет ли при протоколировании плана выполнения выводиться длительность на уровне узлов: он равнозначен указанию `TIMING` команды `EXPLAIN`. Издержки от постоянного чтения системных часов могут значительно замедлить запросы в некоторых системах, так что может иметь смысл отключать этот параметр, когда нужно знать только знать количество строк, но не точную длительность каждого узла. Этот параметр действует, только если включён `auto_explain.log_analyze`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_triggers` (boolean)

При включении параметра `auto_explain.log_triggers` в протокол будет записываться статистика выполнения триггеров. Этот параметр действует, только если включён параметр `auto_explain.log_analyze`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_verbose` (boolean)

Параметр `auto_explain.log_verbose` определяет, будут ли при протоколировании плана выполнения выводиться подробные сведения; он равнозначен указанию `VERBOSE` команды `EXPLAIN`. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_settings` (boolean)

Параметр `auto_explain.log_settings` определяет, будут ли вместе с планами выполнения выводиться изменённые параметры конфигурации. При его включении выводятся только те параметры, которые влияют на планирование запросов и имеют значения, отличающиеся от встроенных. По умолчанию этот параметр отключён. Изменить его могут только суперпользователи.

`auto_explain.log_format` (enum)

Параметр `auto_explain.log_format` выбирает формат вывода для `EXPLAIN`. Он может принимать значение `text`, `xml`, `json` и `yaml`. Значение по умолчанию — text. Изменить этот параметр могут только суперпользователи.

`auto_explain.log_level` (enum)

Параметр `auto_explain.log_level` выбирает уровень, с которым auto_explain будет выводить в протокол планы запросов. Допустимые значения: `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING` и `LOG`. По умолчанию подразумевается `LOG`. Изменить этот параметр могут только суперпользователи.

`auto_explain.log_nested_statements` (boolean)

При включении параметра `auto_explain.log_nested_statements` протоколированию могут подлежать и вложенные операторы (операторы, выполняемые внутри функции). Когда он отключён, протоколируются планы запросов только верхнего уровня. Изменить этот параметр могут только суперпользователи.

`auto_explain.sample_rate` (real)

Параметр `auto_explain.sample_rate` задаёт для auto_explain процент операторов, которые будут отслеживаться в каждом сеансе. Значение по умолчанию — 1, то есть отслеживаются все запросы. Вложенные операторы отслеживаются совместно — либо все, либо никакой из них. Изменить этот параметр могут только суперпользователи.

В обычной ситуации эти параметры устанавливаются в `postgresql.conf`, хотя суперпользователи могут изменить их «на лету» в рамках своих сеансов. Типичное их использование может выглядеть так:

```
# postgresql.conf
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```

F.4.2. Пример

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SET auto_explain.log_analyze = true;
postgres=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
```

В результате этих команд может быть получен такой вывод:

```
LOG:  duration: 0.196 ms plan:
  Query Text: SELECT count(*)
              FROM pg_class, pg_index
              WHERE oid = indrelid AND indisunique;
  Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1
 loops=1)
    ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594 rows=92
 loops=1)
          Hash Cond: (pg_class.oid = pg_index.indrelid)
          ->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual
 time=0.016..0.140 rows=255 loops=1)
          ->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238 rows=92
 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 4kB
                ->  Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4) (actual
 time=0.008..3.187 rows=92 loops=1)
                      Filter: indisunique
  Planning Time: 0.548 ms
```

Заметьте, что в Postgres Pro было добавлено новое поле для отображения длительности планирования.

F.4.3. Авторы

Такахиро Итагаки `<itagaki.takahiro@oss.ntt.co.jp>`

Postgres Professional, Москва, Россия

F.5. basebackup_to_shell — пример создания получателей резервной копии для модуля pg_basebackup

В модуле `basebackup_to_shell` реализован дополнительный получатель базовых копий, названный `shell`. Он позволяет выполнить `pg_basebackup --target=shell` или, в зависимости от конфигурации этого модуля, `pg_basebackup --target=shell:`*`ДОП_ИНФОРМАЦИЯ`* и вызвать серверную команду, выбранную администратором сервера для каждого tar-архива, созданного в процессе резервного копирования. Команда получит содержимое архива через стандартный ввод.

Этот модуль разрабатывался в первую очередь как пример создания новых получателей резервной копии посредством модуля расширения, но в некоторых сценариях он может быть полезен сам по себе. Для использования модуля его нужно загрузить через shared_preload_libraries или local_preload_libraries.

F.5.1. Параметры конфигурации

`basebackup_to_shell.command` (`string`)

Команда, которую сервер должен выполнять для каждого архива, созданного в процессе резервного копирования. Если строка команды содержит символы `%f`, они заменяются именем архива (например, `base.tar`). Символы `%d` в строке команды заменяются дополнительной информацией, указанной пользователем, — она должна задаваться обязательно, если строка команды содержит символы `%d`; в противном случае задать её нельзя. Из соображений безопасности описание может содержать только буквенные и цифровые символы. Символы `%%` в командной строке заменяются одним символом `%`. Если за символом `%` в командной строке следует любой другой символ или он стоит в конце строки, возникает ошибка.

`basebackup_to_shell.required_role` (`string`)

Роль, необходимая для использования получателя резервных копий `shell`. Если роль не задана, этого получателя может использовать любой пользователь репликации.

F.5.2. Автор

Роберт Хаас `<rhaas@postgresql.org>`

F.6. basic_archive — пример модуля архивирования WAL

Модуль `basic_archive` является примером модуля архивирования. Он копирует файлы завершённых сегментов WAL в указанный каталог. Он не очень полезен сам по себе, но на его основе можно разработать собственный модуля архивирования. За дополнительными сведениями о модулях архивирования обратитесь к разделу Глава 49.

Чтобы этот модуль работал, его нужно добавить в shared_preload_libraries, а также включить режим archive_mode.

F.6.1. Параметры конфигурации

`basic_archive.archive_directory` (`string`)

Каталог, в который сервер должен копировать файлы сегментов WAL. Этот каталог уже должен существовать. Значение по умолчанию — пустая строка, то есть архивирование WAL остановлено, но если archive_mode включён, сервер будет накапливать файлы сегментов WAL, ожидая, что значение будет вскоре задано.

Эти параметры должны задаваться в `postgresql.conf`. Обычное использование выглядит так:

```
# postgresql.conf
archive_mode = 'on'
archive_library = 'basic_archive'
basic_archive.archive_directory = '/path/to/archive/directory'
```

F.6.2. Примечания

В случаях сбоя сервера в каталоге архива могут оставаться временные файлы с префиксом `archtemp`. Такие файлы рекомендуется удалять перед перезапуском сервера после сбоя, но их можно удалять и во время работы сервера, убедившись в том, что они не относятся к выполняющимся процессам архивирования.

F.6.3. Автор

Натан Боссарт (Nathan Bossart)

F.7. bloom — индексный метод доступа, основанный на фильтрах Блума

Модуль `bloom` предоставляет индексный метод доступа, основанный на *фильтрах Блума*.

Фильтр Блума представляет собой компактную структуру данных, позволяющую проверить, является ли элемент членом множества. В виде метода доступа индекса он позволяет быстро исключать неподходящие кортежи по сигнатурам, размер которых определяется при создании индекса.

Сигнатура — это неточное представление проиндексированных атрибутов, вследствие чего оно допускает ложные положительные срабатывания; то есть оно может показывать, что элемент содержится в множестве, хотя это не так. Поэтому результаты поиска по такому индексу должны всегда перепроверяться по фактическим значениям атрибутов записи в таблице. Чем больше размер сигнатуры, тем меньше вероятность ложного срабатывания и число напрасных обращений к таблице, но это, разумеется, влечёт увеличение индекса и замедление сканирования.

Этот тип индекса наиболее полезен, когда в таблице много атрибутов и в запросах проверяются их произвольные сочетания. Традиционный индекс-B-дерево быстрее индекса Блума, но для поддержки всевозможных запросов может потребоваться множество индексов типа B-дерево, при том что индекс Блума нужен всего один. Заметьте, однако, что индексы Блума поддерживают только проверки на равенство, тогда как индексы-B-деревья также полезны при проверке неравенств и поиске в диапазоне.

F.7.1. Параметры

Индекс `bloom` принимает в своём предложении `WITH` следующие параметры:

`length`

Длина каждой сигнатуры (элемента индекса) в битах, округлённая вверх до ближайшего числа, кратного `16`. Значение по умолчанию — `80`, а максимальное значение — `4096`.

`col1 — col32`

Число битов, генерируемых для каждого столбца индекса. В имени параметра отражается номер столбца индекса, для которого это число задаётся. Значение по умолчанию — `2` бита, а максимум — `4095`. Параметры для неиспользуемых столбцов индекса игнорируются.

F.7.2. Примеры

Пример создания индекса bloom:

```
CREATE INDEX bloomidx ON tbloom USING bloom (i1,i2,i3)
       WITH (length=80, col1=2, col2=2, col3=4);
```

Эта команда создаёт индекс с длиной сигнатуры 80 бит, в которой атрибуты i1 и i2 отображаются в 2 бита, а атрибут i3 — в 4. Мы могли бы опустить указания `length`, `col1` и `col2`, так как в них задаются значения по умолчанию.

Ниже представлен более полный пример определения и использования индекса Блума, а также приводится сравнение его с равнозначным индексом-B-деревом. Видно, что индекс Блума значительно меньше индекса-B-дерева, и при этом он может работать быстрее.

```
=# CREATE TABLE tbloom AS
   SELECT
     (random() * 1000000)::int as i1,
     (random() * 1000000)::int as i2,
     (random() * 1000000)::int as i3,
     (random() * 1000000)::int as i4,
     (random() * 1000000)::int as i5,
     (random() * 1000000)::int as i6
```

```
   FROM
  generate_series(1,10000000);
SELECT 10000000
```

Последовательное сканирование по этой большой таблице выполняется долго:

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                              QUERY PLAN
-------------------------------------------------------------------
-----------------------------------
 Seq Scan on tbloom  (cost=0.00..213744.00 rows=250 width=24) (actual
 time=357.059..357.059 rows=0 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 10000000
 Planning Time: 0.346 ms
 Execution Time: 357.076 ms
(5 rows)
```

Даже при наличии индекса btree сканирование остаётся последовательным:

```
=# CREATE INDEX btreeidx ON tbloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('btreeidx'));
 pg_size_pretty
----------------
 386 MB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                              QUERY PLAN
-------------------------------------------------------------------
-----------------------------------
 Seq Scan on tbloom  (cost=0.00..213744.00 rows=2 width=24) (actual
 time=351.016..351.017 rows=0 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 10000000
 Planning Time: 0.138 ms
 Execution Time: 351.035 ms
(5 rows)
```

Если же для таблицы создан индекс bloom, поиск такого рода выполняется эффективнее, чем с индексом btree:

```
=# CREATE INDEX bloomidx ON tbloom USING bloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('bloomidx'));
 pg_size_pretty
----------------
 153 MB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                     QUERY PLAN
-------------------------------------------------------------------
---------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=1792.00..1799.69 rows=2 width=24) (actual
 time=22.605..22.606 rows=0 loops=1)
   Recheck Cond: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Index Recheck: 2300
   Heap Blocks: exact=2256
   ->  Bitmap Index Scan on bloomidx  (cost=0.00..178436.00 rows=1 width=0) (actual
 time=20.005..20.005 rows=2300 loops=1)
         Index Cond: ((i2 = 898732) AND (i5 = 123451))
```

```
 Planning Time: 0.099 ms
 Execution Time: 22.632 ms
(8 rows)
```

При таком подходе основная проблема поиска по B-дереву состоит в том, что B-дерево неэффективно, когда условия поиска не ограничивают ведущие столбцы индекса. Поэтому, применяя индексы типа B-дерево, лучше создавать отдельные индексы для каждого столбца. В этом случае планировщик построит примерно такой план:

```
=# CREATE INDEX btreeidx1 ON tbloom (i1);
CREATE INDEX
=# CREATE INDEX btreeidx2 ON tbloom (i2);
CREATE INDEX
=# CREATE INDEX btreeidx3 ON tbloom (i3);
CREATE INDEX
=# CREATE INDEX btreeidx4 ON tbloom (i4);
CREATE INDEX
=# CREATE INDEX btreeidx5 ON tbloom (i5);
CREATE INDEX
=# CREATE INDEX btreeidx6 ON tbloom (i6);
CREATE INDEX
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
                                                          QUERY PLAN
-------------------------------------------------------------------
--------------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=9.29..13.30 rows=1 width=24) (actual
 time=0.032..0.033 rows=0 loops=1)
   Recheck Cond: ((i5 = 123451) AND (i2 = 898732))
   ->  BitmapAnd  (cost=9.29..9.29 rows=1 width=0) (actual time=0.047..0.047 rows=0
 loops=1)
         ->  Bitmap Index Scan on btreeidx5  (cost=0.00..4.52 rows=11 width=0) (actual
 time=0.026..0.026 rows=7 loops=1)
               Index Cond: (i5 = 123451)
         ->  Bitmap Index Scan on btreeidx2  (cost=0.00..4.52 rows=11 width=0) (actual
 time=0.007..0.007 rows=8 loops=1)
               Index Cond: (i2 = 898732)
 Planning Time: 0.264 ms
 Execution Time: 0.047 ms
(9 rows)
```

Хотя этот запрос выполняется гораздо быстрее, чем с каким-либо одиночным индексом, мы платим за это увеличением размера индекса. Каждый индекс-B-дерево занимает 88.5 Мбайта, так что общий объём индексов составляет 531 Мбайт, что в более 3 раз больше размера индекса Блума.

F.7.3. Интерфейс класса операторов

Класс операторов для индексов Блума требует наличия только хеш-функции для индексируемого типа данных и оператора равенства для поиска. Этот пример демонстрирует соответствующее определение класса операторов для типа `text`:

```
CREATE OPERATOR CLASS text_ops
DEFAULT FOR TYPE text USING bloom AS
    OPERATOR    1   =(text, text),
    FUNCTION    1   hashtext(text);
```

F.7.4. Ограничения

- В этот модуль включены только классы операторов для `int4` и `text`.
- При поиске поддерживается только оператор `=`. Но в будущем возможно добавление поддержки для массивов с операциями объединения и пересечения.

- Метод доступа `bloom` не поддерживает уникальные индексы (`UNIQUE`).
- Метод доступа `bloom` не поддерживает поиск значений `NULL`.

F.7.5. Авторы

Фёдор Сигаев `<teodor@postgrespro.ru>`, Postgres Professional, Москва, Россия

Александр Коротков `<a.korotkov@postgrespro.ru>`, Postgres Professional, Москва, Россия

Олег Бартунов `<obartunov@postgrespro.ru>`, Postgres Professional, Москва, Россия

F.8. btree_gin — классы операторов GIN с поведением B-дерева

Модуль `btree_gin` предоставляет классы операторов GIN, реализующие поведение, подобное тому, что реализуют обычные классы B-дерева, для типов данных `int2`, `int4`, `int8`, `float4`, `float8`, `timestamp with time zone`, `timestamp without time zone`, `time with time zone`, `time without time zone`, `date`, `interval`, `oid`, `money`, `"char"`, `varchar`, `text`, `bytea`, `bit`, `varbit`, `macaddr`, `macaddr8`, `inet`, `cidr`, `uuid`, `name`, `bool`, `bpchar` и всех типов-перечислений (`enum`).

Вообще говоря, эти классы операторов не будут работать быстрее аналогичных стандартных методов индекса-B-дерева, и им не хватает одной важной возможности стандартной реализации B-дерева: возможности ограничивать уникальность. Тем не менее их можно применять для тестирования GIN или взять за основу для разработки других классов операторов GIN. Также, для запросов, где проверяется и столбец с индексом GIN, и столбец с индексом-B-деревом, может быть более эффективным создать составной индекс GIN, который использует один из этих классов операторов, чем использовать два отдельных индекса, выборку из которых придётся объединять, вычисляя AND битовых карт.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.8.1. Пример использования

```
CREATE TABLE test (a int4);
-- создать индекс
CREATE INDEX testidx ON test USING GIN (a);
-- запрос
SELECT * FROM test WHERE a < 10;
```

F.8.2. Авторы

Фёдор Сигаев (`<teodor@stack.net>`) и Олег Бартунов (`<oleg@sai.msu.su>`). Подробности можно найти на странице *http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin*.

F.9. btree_gist — классы операторов GiST с поведением B-дерева

Модуль `btree_gist` предоставляет классы операторов GiST, реализующие поведение, подобное тому, что реализуют обычные классы B-дерева, для типов данных `int2`, `int4`, `int8`, `float4`, `float8`, `numeric`, `timestamp with time zone`, `timestamp without time zone`, `time with time zone`, `time without time zone`, `date`, `interval`, `oid`, `money`, `char`, `varchar`, `text`, `bytea`, `bit`, `varbit`, `macaddr`, `macaddr8`, `inet`, `cidr`, `uuid`, `bool` и всех типов `enum`.

Вообще говоря, эти классы операторов не будут работать быстрее аналогичных стандартных методов индекса-B-дерева, и им не хватает одной важной возможности стандартной реализации B-дерева: возможности ограничивать уникальность. Однако они предлагают несколько других возможностей, описанных ниже. Также эти классы операторов полезны, когда требуется составной индекс GiST, в котором некоторые столбцы имеют типы данных, индексируемые только с GiST, а другие — простые типы. Наконец, эти классы операторов можно применять для тестирования GiST или взять за основу для разработки других классов операторов GiST.

Помимо типичных операторов поиска по B-дереву, `btree_gist` также поддерживает использование индекса для операции `<>` («не равно»). Это может быть полезно в сочетании с ограничением-исключением, как описано ниже.

Также, для типов данных, имеющих естественную метрику расстояния, `btree_gist` определяет оператор расстояния `<->` и поддерживает использование индексов GiST для поиска ближайших соседей с применением этого оператора. Операторы расстояния определены для типов `int2`, `int4`, `int8`, `float4`, `float8`, `timestamp with time zone`, `timestamp without time zone`, `time without time zone`, `date`, `interval`, `oid` и `money`.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.9.1. Пример использования

Простой пример использования `btree_gist` вместо `btree`:

```
CREATE TABLE test (a int4);
-- создать индекс
CREATE INDEX testidx ON test USING GIST (a);
-- запрос
SELECT * FROM test WHERE a < 10;
-- поиск ближайших соседей: найти десять записей, ближайших к "42"
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

Так можно использовать ограничение-исключение, состоящее в том, что в клетке в зоопарке могут содержаться животные только одного типа:

```
=> CREATE TABLE zoo (
  cage   INTEGER,
  animal TEXT,
  EXCLUDE USING GIST (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage,
 animal)=(123, zebra).
```

```
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.9.2. Авторы

Фёдор Сигаев (<`teodor@stack.net`>), Олег Бартунов (<`oleg@sai.msu.su`>), Янко Рихтер (<`jankorichter@yahoo.de`>) и Пол Юнгвирт (<`pj@illuminatedcomputing.com`>). Подробности можно найти на странице *http://www.sai.msu.su/~megera/postgres/gist/*.

F.10. citext — тип данных для строк, нечувствительных к регистру

Модуль `citext` предоставляет тип данных для строк, нечувствительных к регистру, `citext`. По сути он сравнивает значения, вызывая внутри себя функцию `lower`. В остальном он почти не отличается от типа `text`.

> **Подсказка**
>
> Вместо этого модуля имеет смысл использовать *недетерминированные правила сортировки* (см. Подраздел 22.2.2.4). Они позволяют осуществлять сравнение без учёта регистра, без учёта ударения и другие варианты сравнений, при этом более корректно обрабатывая особые случаи Unicode.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.10.1. Обоснование

Стандартный способ выполнить сравнение строк без учёта регистра в Postgres Pro заключается в использовании функции `lower` при сравнении значений, например

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

Этот подход работает довольно хорошо, но имеет ряд недостатков:

- Операторы SQL становятся громоздкими, и нужно не забывать всегда обрабатывать функцией `lower` и столбец, и значение.
- Индекс не будет использоваться, если только дополнительно не создать функциональный индекс с функцией `lower`.
- Если вы объявляете столбец как `UNIQUE` или `PRIMARY KEY`, неявно создаваемый индекс будет чувствительным к регистру. Поэтому он бесполезен для регистронезависимого поиска, так же как он не будет обеспечивать уникальность без учёта регистра.

Тип данных `citext` позволяет исключить вызовы `lower` в SQL-запросах и позволяет сделать первичный ключ регистронезависимым. Тип `citext` учитывает локаль, так же, как и тип `text`, что означает, что сравнение символов в верхнем и нижнем регистре зависит от правил `LC_CTYPE` для базы данных. Это поведение, опять же, не отличается от вызовов `lower` в запросах. Но так как оно реализуется прозрачно типом данных, в самих запросах дополнительно не нужно ничего делать.

F.10.2. Как его использовать

Простой пример использования:

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
);

INSERT INTO users VALUES ( 'larry',  sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Tom',    sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Damian', sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'NEAL',   sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Bjørn',  sha256(random()::text::bytea) );

SELECT * FROM users WHERE nick = 'Larry';
```

Оператор `SELECT` вернёт один кортеж, несмотря на то, что в столбец `nick` записано значение `larry`, а в запросе фигурирует `Larry`.

F.10.3. Поведение при сравнении строк

Модуль `citext` выполняет сравнения, приводя каждую строку к нижнему регистру (как если бы вызывалась функция `lower`) и затем производя сравнения как обычно. Так, например, две строки будут считаться равными, если функция `lower`, обработав их, выдаст одинаковые результаты.

Чтобы имитировать правило сортировки без учёта регистра в максимально возможной степени, этот модуль предоставляет специальные, ориентированные на `citext`, операторы и функции для обработки строки. Так, например, операторы регулярных выражений `~` и `~*` действуют в том же ключе, когда применяются к типу `citext`: оба они не учитывают регистр. Это же распространяется на операторы `!~` и `!~*`, а также операторы `LIKE` `~~`, `~~*`, `!~~` и `!~~*`. Если же вы хотите, чтобы эти операторы учитывали регистр, вы можете привести их аргументы к типу `text`.

Подобным образом, все следующие функции выполняют сопоставления без учёта регистра, если их аргументы имеют тип `citext`:

- `regexp_match()`
- `regexp_matches()`
- `regexp_replace()`
- `regexp_split_to_array()`
- `regexp_split_to_table()`
- `replace()`
- `split_part()`
- `strpos()`
- `translate()`

Для функций с регулярными выражениями, если вам нужно регистрозависимое сопоставление, вы можете добавить флаг «c», чтобы принудительно включить этот режим. Чтобы получить регистрозависимое поведение без этого флага, вы должны привести аргумент к типу `text`, прежде чем вызывать эту функцию.

F.10.4. Ограничения

- Смена регистра символов в `citext` зависит от параметра `LC_CTYPE` вашей базы данных. Таким образом, как будут сравниваться значения, определяется при создании базы данных. На самом деле, по определениям стандарта Unicode, это сравнение не будет истинно регистронезависимым. По сути это означает, что если вас устраивает установленное правило сортировки, вас должны устраивать и сравнения `citext`. Но если в вашей базе данных хранятся строки на разных языках, пользователи одного языка могут получать неожиданные результаты запросов, если правило сортировки предназначено для другого языка.
- Начиная с PostgreSQL версии 9.1, вы можете добавлять указание `COLLATE` к значениям данных или столбцам `citext`. В настоящее время операторы `citext` принимают во внимание такое явное указание `COLLATE`, сравнивая строки в нижнем регистре, но изначальное приведение в нижний регистр всегда выполняется согласно параметру `LC_CTYPE` базы данных (как если бы указывалось `COLLATE "default"`). Это может быть изменено в будущем, чтобы на обоих этапах учитывалось указание `COLLATE` во входных данных.
- Тип `citext` не так эффективен, как `text`, так как функции операторов и функции сравнения для B-дерева должны делать копии данных и переводить их в нижний регистр для сравнения. Кроме того, с типом `text` возможно исключение дубликатов в B-дереве. Тем не менее с `citext` регистронезависимое сравнение реализуется эффективнее, чем с применением `lower`.

- Тип `citext` малополезен в ситуациях, когда вам нужно сравнивать данные без учёта регистра в одних контекстах, и с учётом регистра — в других. Обычно в таких случаях используют `text` и вручную применяют функцию `lower`, когда нужно выполнить сравнение без учёта регистра; это прекрасно работает, если регистронезависимое сравнение требуется выполнять относительно редко. Если же почти всегда сравнение должно быть регистронезависимым и только иногда регистрозависимым, имеет смысл сохранить данные в столбце типа `citext`, и явно приводить их к типу `text` для регистрозависимого сравнения. В любом случае, чтобы оба варианта поиска были быстрыми, вам потребуются два индекса.
- Схема, содержащая операторы `citext`, должна находиться в текущем пути `search_path` (обычно это схема `public`); в противном случае будут вызываться регистрозависимые операторы для типа `text`.
- Подход с переводом строк в нижний регистр для сравнения не работает в некоторых особых случаях Unicode, например когда одной букве в верхнем регистре соответствуют две буквы в нижнем регистре. Поэтому в Unicode различаются понятия *преобразование регистра* (case mapping) и *выравнивание регистра* (case folding). Чтобы сравнение производилось корректно и в этих случаях, используйте вместо `citext` недетерминированные правила сортировки.

F.10.5. Автор

Дэвид Е. Уилер `<david@kineticode.com>`

Разработку вдохновил оригинальный модуль `citext` Дональда Фрейзера.

F.11. cube — тип данных для представления многомерных кубов

Этот модуль реализует тип данных `cube` для представления многомерных кубов.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.11.1. Синтаксис

В Таблице F.5 показаны внешние представления типа `cube`. Буквы *x*, *y* и т. д. обозначают числа с плавающей точкой.

Таблица F.5. Внешние представления кубов

Внешний синтаксис	Значение
x	Одномерная точка (или одномерный интервал нулевой длины)
`(x)`	То же, что и выше
`x1, x2,..., xn`	Точка в n-мерном пространстве, представленная внутри как куб нулевого объёма
`(x1, x2,..., xn)`	То же, что и выше
`(x),(y)`	Одномерный интервал, начинающийся в точке *x* и заканчивающийся в *y*, либо наоборот; порядок значения не имеет
`[(x),(y)]`	То же, что и выше
`(x1,..., xn),(y1,..., yn)`	N-мерный куб, представленный парой диагонально противоположных углов
`[(x1,..., xn),(y1,..., yn)]`	То же, что и выше

В каком порядке вводятся противоположные углы куба, не имеет значения. Функции, принимающие тип `cube`, автоматически меняют углы местами, чтобы получить единое внутреннее представление «левый нижний — правый верхний». Когда эти углы совмещаются, в `cube` для экономии пространства хранится только один угол с флагом «является точкой».

Пробельные символы игнорируются, так что `[(x),(y)]` не отличается от `[ ( x ), ( y ) ]`.

F.11.2. Точность

Значения хранятся внутри как 64-битные числа с плавающей точкой. Это значит, что числа с более чем 16 значащими цифрами будут усекаться.

F.11.3. Использование

В Таблице F.6 показаны операторы, предназначенные специально для работы с типом `cube`.

Таблица F.6. Операторы для кубов

Оператор **Описание**
`cube && cube → boolean` Кубы пересекаются?
`cube @> cube → boolean` Первый куб содержит второй?
`cube <@ cube → boolean` Первый куб содержится во втором?

Оператор	**Описание**
`cube -> integer` → `float8`	Выдаёт *n*-ю координату куба (считая с 1).
`cube ~> integer` → `float8`	Выдаёт *n*-ю координату куба, применяя следующую нумерацию: *n* = 2 * *k* - 1 обозначает нижнюю границу *k*-й размерности, а *n* = 2 * *k* обозначает верхнюю границу *k*-й размерности. Отрицательные *n* обозначают обратное значение соответствующей положительной координаты. Этот оператор предназначен для поддержки KNN-GiST.
`cube <-> cube` → `float8`	Вычисляет евклидово расстояние между двумя кубами.
`cube <#> cube` → `float8`	Вычисляет расстояние городских кварталов (метрику L-1) между двумя кубами.
`cube <=> cube` → `float8`	Вычисляет расстояние Чебышева (метрику L-бесконечность) между двумя кубами.

Помимо показанных выше операторов, для типа `cube` имеются обычные операторы сравнения, показанные в Таблице 9.1. Эти операторы сначала сравнивают первые координаты и если они равны, сравнивают вторые и т. д. Они предназначены в основном для поддержки класса операторов индекса-B-дерева для типа `cube`, который может быть полезен, например, если вы хотите создать ограничение UNIQUE для столбца типа `cube`. Для других практических применений реализуемая ими сортировка не очень полезна.

Модуль `cube` также предоставляет класс операторов индекса GiST для значений `cube`. Индекс GiST для `cube` может применяться для поиска значений в выражениях с операторами `=`, `&&`, `@>` и `<@` в предложениях `WHERE`.

GiST-индекс для `cube` может быть полезен и для поиска ближайших соседей с использованием операторов метрики `<->`, `<#>` и `<=>` в предложениях `ORDER BY`. Например, ближайшего соседа точки в трёхмерном пространстве (0.5, 0.5, 0.5) можно эффективно найти так:

```
SELECT c FROM test ORDER BY c <-> cube(array[0.5,0.5,0.5]) LIMIT 1;
```

Оператор `~>` может также использоваться таким образом, чтобы эффективно выдавать первые несколько значений, отсортированных по выбранной координате. Например, чтобы получить первые несколько кубов, упорядоченных по возрастанию первой координаты (левого нижнего угла), можно использовать следующий запрос:

```
SELECT c FROM test ORDER BY c ~> 1 LIMIT 5;
```

А чтобы получить двумерные кубы, отсортированные по убыванию первой координаты правого верхнего угла:

```
SELECT c FROM test ORDER BY c ~> 3 DESC LIMIT 5;
```

В Таблице F.7 перечислены все доступные функции.

Таблица F.7. Функции для работы с кубами

Функция	**Описание**	**Примеры**
`cube` (`float8`) → `cube`	Создаёт одномерный куб, у которого обе координаты равны.	`cube(1)` → `(1)`
`cube` (`float8`, `float8`) → `cube`	Создаёт одномерный куб.	

Функция **Описание** **Примеры**
`cube(1, 2)` → `(1),(2)`

`cube ( float8[] ) → cube`
Создаёт куб нулевого объёма по координатам, определяемым массивом.
`cube(ARRAY[1,2,3])` → `(1, 2, 3)`

`cube ( float8[], float8[] ) → cube`
Создаёт куб с координатами правого верхнего и левого нижнего углов, определяемыми двумя массивами, которые должны быть одинаковой длины.
`cube(ARRAY[1,2], ARRAY[3,4])` → `(1, 2),(3, 4)`

`cube ( cube, float8 ) → cube`
Создаёт новый куб, добавляя размерность к существующему кубу с одинаковым значением новой координаты для обеих углов. Это бывает полезно, когда нужно построить кубы поэтапно из вычисляемых значений.
`cube('(1,2),(3,4)'::cube, 5)` → `(1, 2, 5),(3, 4, 5)`

`cube ( cube, float8, float8 ) → cube`
Создаёт новый куб, добавляя размерность к существующему кубу. Это бывает полезно, когда нужно построить кубы поэтапно из вычисляемых значений.
`cube('(1,2),(3,4)'::cube, 5, 6)` → `(1, 2, 5),(3, 4, 6)`

`cube_dim ( cube ) → integer`
Возвращает число размерностей куба.
`cube_dim('(1,2),(3,4)')` → `2`

`cube_ll_coord ( cube, integer ) → float8`
Выдаёт значение *n*-й координаты левого нижнего угла куба.
`cube_ll_coord('(1,2),(3,4)', 2)` → `2`

`cube_ur_coord ( cube, integer ) → float8`
Выдаёт значение *n*-й координаты правого верхнего угла куба.
`cube_ur_coord('(1,2),(3,4)', 2)` → `4`

`cube_is_point ( cube ) → boolean`
Возвращает true, если куб является точкой, то есть если два определяющих его угла совпадают.
`cube_is_point(cube(1,1))` → `t`

`cube_distance ( cube, cube ) → float8`
Возвращает расстояние между двумя кубами. Если оба куба являются точками, вычисляется обычная функция расстояния.
`cube_distance('(1,2)', '(3,4)')` → `2.8284271247461903`

`cube_subset ( cube, integer[] ) → cube`
Создаёт новый куб из существующего, используя список размерностей из массива. Может применяться для получения координат углов в одном измерении, для удаления измерений и изменения их порядка.
`cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[2])` → `(3),(7)`
`cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[3,2,1,1])` → `(5, 3, 1, 1),(8, 7, 6, 6)`

`cube_union ( cube, cube ) → cube`
Создаёт объединение двух кубов.
`cube_union('(1,2)', '(3,4)')` → `(1, 2),(3, 4)`

Функция **Описание** **Примеры**

`cube_inter` (`cube`, `cube`) → `cube`
Создаёт пересечение двух кубов.
`cube_inter('(1,2)', '(3,4)')` → `(3, 4),(1, 2)`

`cube_enlarge` (*`c`* `cube`, *`r`* `double`, *`n`* `integer`) → `cube`
Увеличивает размер куба на заданный радиус *`r`* как минимум в *`n`* измерениях. Если радиус отрицательный, куб, наоборот, уменьшается. Все определённые измерения изменяются на величину радиуса *`r`*. Координаты левого нижнего угла уменьшаются на *`r`*, а координаты правого верхнего увеличиваются на *`r`*. Если координата левого нижнего угла становится больше соответствующей координаты правого верхнего (это возможно, только когда *`r`* < 0), обоим координатам присваивается их среднее значение. Если *`n`* превышает число определённых измерений и куб увеличивается (*`r`* > 0), добавляются дополнительные размерности, недостающие до *`n`*; начальным значением для дополнительных координат считается ноль. Эта функция полезна для создания окружающих точку прямоугольников для поиска ближайших точек.
`cube_enlarge('(1,2),(3,4)', 0.5, 3)` → `(0.5, 1.5, -0.5),(3.5, 4.5, 0.5)`

F.11.4. Поведение по умолчанию

Это объединение:

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-------------------
(0, 0, 0),(2, 5, 2)
(1 row)
```

не противоречит здравому смыслу, как и это пересечение:

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

Во всех бинарных операциях с кубами разных размерностей предполагается, что куб с меньшей размерностью является декартовой проекцией; то есть в опущенных в строковом представлении координатах предполагаются нули. Таким образом, показанные выше вызовы равнозначны следующим:

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

В следующем предикате включения применяется синтаксис точек, хотя фактически второй аргумент представляется внутри кубом. Этот синтаксис избавляет от необходимости определять отдельный тип точек и функции для предикатов (cube,point).

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.11.5. Примечания

Примеры использования можно увидеть в регрессионном тесте `sql/cube.sql`.

Во избежание некорректного применения этого типа, число размерностей кубов искусственно ограничено значением 100. Если это ограничение вас не устраивает, его можно изменить в `cubedata.h`.

F.11.6. Благодарности

Первый автор: Джин Селков мл. `<selkovjr@mcs.anl.gov>`, Аргоннская национальная лаборатория, Отдел математики и компьютерных наук

Я очень благодарен в первую очередь профессору Джо Геллерштейну (*https://dsf.berkeley.edu/jmh/*) за пояснение сути GiST (*http://gist.cs.berkeley.edu/*) и его бывшему студенту, Энди Донгу, за пример, написанный для Illustra. Я также признателен всем разработчикам Postgres в настоящем и прошлом за возможность создать свой собственный мир и спокойно жить в нём. Ещё я хотел бы выразить признательность Аргоннской лаборатории и Министерству энергетики США за годы постоянной поддержки моих исследований в области баз данных.

Небольшие изменения в этот пакет внёс Бруно Вольф III `<bruno@wolff.to>` в августе/сентябре 2002 г. В том числе он перешёл от одинарной к двойной точности и добавил несколько новых функций.

Дополнительные изменения внёс Джошуа Рейх `<josh@root.net>` в июле 2006 г. В частности, он добавил `cube(float8[], float8[])`, подчистил код и перевёл его на протокол вызовов версии V1 с устаревшего протокола V0.

F.12. daterange_inclusive — `daterange`, включающий верхнюю границу диапазона

`daterange_inclusive` — это расширение встроенного диапазонного типа `daterange`. По умолчанию `daterange` исключает верхнюю границу диапазона `[)`, а `daterange_inclusive` — нет `[]`. За исключением границы покрываемого диапазона между `daterange` и `daterange_inclusive` нет разницы.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.12.1. Обоснование

Иногда бывает удобно выводить временные диапазоны со включённой верхней границей. Например, это применимо к отпускам сотрудников. Если сотрудник в отпуске с 1 января 2025 года по 10 января 2025 года, база данных не сможет полностью вывести этот временной диапазон при использовании диапазонного типа `daterange`. В этом случае необходимо вручную вычитать один день из вывода. Расширение `daterange_inclusive` решает эту проблему.

F.12.2. Пример использования

```
CREATE EXTENSION daterange_inclusive;
SELECT daterange('[2024-01-01,2024-06-01]');
      daterange
-------------------------
[2024-01-01,2024-06-02)

SELECT daterange_inclusive('[2024-01-01,2024-06-01]');
   daterange_inclusive
-------------------------
[2024-01-01,2024-06-01]
```

F.13. dbcopies_decoding — модуль 1С для обновления копий баз данных

`dbcopies_decoding` — модуль 1С для обновления копий БД. Он реализован как расширение с механизмом логической репликации и поставляется в рамках `postgrespro-std-17-contrib`. Для корректной работы модуля параметру `wal_level` необходимо задать значение `logical`. За более подробной информацией обратитесь в поддержку 1С.

F.14. dblink — подключение к другим базам данных Postgres Pro

Модуль `dblink` обеспечивает подключения к другим базам данных Postgres Pro из сеанса базы данных.

Модуль `dblink` может сообщать о следующих событиях ожидания с типом `Extension`.

`DblinkConnect`

Ожидание установления подключения к удалённому серверу.

`DblinkGetConnect`

Ожидание установления подключения к удалённому серверу, если его не удалось найти в списке уже открытых подключений.

`DblinkGetResult`

Ожидание получения результатов запроса от удалённого сервера.

См. также описание модуля postgres_fdw, который предоставляет примерно ту же функциональность, но через более современную и стандартизированную инфраструктуру.

dblink_connect

dblink_connect — открывает постоянное подключение к удалённой базе данных

Синтаксис

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

Описание

Функция `dblink_connect()` устанавливает подключение к удалённой базе данных Postgres Pro. Целевой сервер и база данных указываются в стандартной строке подключения libpq. Если требуется, этому подключению можно назначить имя. В один момент времени могут быть открытыми несколько именованных подключений, но только одно подключение без имени. Подключение будет сохраняться, пока не будет закрыто или до завершения сеанса базы данных.

В строке подключения также может задаваться имя существующего стороннего сервера. Для определения стороннего сервера рекомендуется использовать обёртку сторонних данных `dblink_fdw`. См. пример ниже, а также CREATE SERVER и CREATE USER MAPPING.

Аргументы

`connname`

Имя, назначаемое этому подключению; если опускается, открывается безымянное подключение, заменяющее ранее существующее безымянное подключение.

`connstr`

Строка подключения в стиле libpq, например `hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd options=-csearch_path=`. За подробностями обратитесь к Подразделу 32.1.1. В ней также может задаваться имя стороннего сервера.

Возвращаемое значение

Возвращает состояние (это всегда строка `OK`, так как в случае любой ошибки функция прерывается, выдавая исключение).

Примечания

Если к базе данных, которая не приведена в соответствие шаблону безопасного использования схем, имеют доступ недоверенные пользователи, начинайте сеанс с удаления доступных им для записи схем из пути поиска (`search_path`). Например, для этого можно добавить `options=-csearch_path=` в *`connstr`*. Это касается не только `dblink`, но и любых других интерфейсов для выполнения произвольных SQL-команд.

Создавать подключения, не требующие аутентификации по паролю и по GSSAPI, с помощью `dblink_connect` разрешено только суперпользователям. Если эта возможность нужна обычным пользователям, следует воспользоваться функцией `dblink_connect_u`.

Использовать в именах подключений знаки «равно» не рекомендуется, так как при этом возможна путаница со строками подключений в других функциях `dblink`.

Примеры

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres options=-csearch_path=');
```

```
 dblink_connect
----------------
 OK
(1 row)

-- Функциональность обёртки сторонних данных (FOREIGN DATA WRAPPER)
-- Замечание: чтобы это работало, для локальных подключений требуется аутентификация по
 паролю
--       В противном случае, вызвав dblink_connect(), вы получите:
--       ERROR:  password is required
--       DETAIL:  Non-superuser cannot connect if the server does not request a
 password.
--       HINT:  Target server's authentication method must be changed.
--
--       ОШИБКА:  требуется пароль
--       ПОДРОБНОСТИ:  Обычный пользователь не может подключиться, если сервер не
 требует пароль.
--       ПОДСКАЗКА:  Необходимо изменить метод аутентификации целевого сервера.

CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr '127.0.0.1',
 dbname 'contrib_regression');

CREATE USER regress_dblink_user WITH PASSWORD 'secret';
CREATE USER MAPPING FOR regress_dblink_user SERVER fdtest OPTIONS (user
 'regress_dblink_user', password 'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO regress_dblink_user;
GRANT SELECT ON TABLE foo TO regress_dblink_user;

\set ORIGINAL_USER :USER
\c - regress_dblink_user
SELECT dblink_connect('myconn', 'fdtest');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'SELECT * FROM foo') AS t(a int, b text, c text[]);
 a  | b |       c
----+---+---------------
  0 | a | {a0,b0,c0}
  1 | b | {a1,b1,c1}
  2 | c | {a2,b2,c2}
  3 | d | {a3,b3,c3}
  4 | e | {a4,b4,c4}
  5 | f | {a5,b5,c5}
  6 | g | {a6,b6,c6}
  7 | h | {a7,b7,c7}
  8 | i | {a8,b8,c8}
  9 | j | {a9,b9,c9}
 10 | k | {a10,b10,c10}
(11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM regress_dblink_user;
REVOKE SELECT ON TABLE foo FROM regress_dblink_user;
DROP USER MAPPING FOR regress_dblink_user SERVER fdtest;
DROP USER regress_dblink_user;
DROP SERVER fdtest;
```

dblink_connect_u

dblink_connect_u — открывает постоянное подключение к удалённой базе данных, небезопасно

Синтаксис

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

Описание

Функция `dblink_connect_u()` не отличается от `dblink_connect()`, за исключением того, что она позволяет подключаться с любым методом аутентификации обычным пользователям.

Если удалённый сервер выбирает режим аутентификации без пароля, возможно олицетворение и последующее повышение привилегий, так как сеанс будет установлен от имени пользователя, который исполняет локальный процесс Postgres Pro. Кроме того, даже если удалённый сервер запрашивает пароль, этот пароль можно получить из среды сервера, например, из файла `~/.pgpass`, принадлежащего пользователю сервера. Это чревато не только олицетворением, но и выдачей пароля не заслуживающему доверия удалённому серверу. Поэтому `dblink_connect_u()` изначально устанавливается так, что роль `PUBLIC` лишена всех прав на её использование, то есть вызывать её могут только суперпользователи. В некоторых ситуациях допустимо дать право `EXECUTE` для `dblink_connect_u()` определённым пользователям, которым можно доверять, но это нужно делать осторожно. Также рекомендуется убедиться в том, что файл `~/.pgpass`, принадлежащий пользователю сервера, *не* содержит никаких записей со звёздочкой в качестве имени узла.

За дополнительными подробностями обратитесь к описанию `dblink_connect()`.

dblink_disconnect

dblink_disconnect — закрывает постоянное подключение к удалённой базе данных

Синтаксис

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

Описание

`dblink_disconnect()` закрывает подключение, ранее открытое функцией `dblink_connect()`. Форма без аргументов закрывает безымянное подключение.

Аргументы

`connname`

Имя закрываемого именованного подключения.

Возвращаемое значение

Возвращает состояние (это всегда строка `OK`, так как в случае любой ошибки функция прерывается, выдавая исключение).

Примеры

```
SELECT dblink_disconnect();
 dblink_disconnect
-------------------
 OK
(1 row)

SELECT dblink_disconnect('myconn');
 dblink_disconnect
-------------------
 OK
(1 row)
```

dblink

dblink — выполняет запрос в удалённой базе данных

Синтаксис

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

Описание

`dblink` выполняет запрос (обычно `SELECT`, но это может быть и любой другой оператор SQL, возвращающий строки) в удалённой базе данных.

Когда этой функции передаются два аргумента типа `text`, первый сначала рассматривается как имя постоянного подключения; если такое подключение находится, команда выполняется для него. Если не находится, первый аргумент воспринимается как строка подключения, как для функции `dblink_connect`, и заданное подключение устанавливается только на время выполнения этой команды.

Аргументы

connname

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

connstr

Строка подключения, описанная ранее для `dblink_connect`

sql

SQL-запрос, который вы хотите выполнить в удалённой базе данных, например `select * from foo`.

fail_on_error

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и функция не возвращает строки.

Возвращаемое значение

Эта функция возвращает строки, выдаваемые в результате запроса. Так как `dblink` может выполнять произвольные запросы, она объявлена как возвращающая тип `record`, а не некоторый определённый набор столбцов. Это означает, что вы должны указать ожидаемый набор столбцов в вызывающем запросе — в противном случае Postgres Pro не будет знать, чего ожидать. Например:

```
SELECT *
    FROM dblink('dbname=mydb options=-csearch_path=',
                'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

В части «псевдонима» предложения `FROM` должны указываться имена столбцов и типы, которые будет возвращать функция. (Указание имён столбцов в псевдониме таблицы предусмотрено стандартом SQL, но определение типов столбцов является расширением Postgres Pro.) Это позволяет системе понять, во что должно разворачиваться обозначение `*`, и на что ссылается `proname` в предложении `WHERE`, прежде чем пытаться выполнять эту функцию. Во время выполнения произойдёт ошибка, если действительный результат запроса из удалённой базы данных не будет содержать столько столбцов, сколько указано в предложении `FROM`. Однако имена столбцов могут не совпадать, так же, как `dblink` не настаивает на точном совпадении типов. Функция завершится успешно, если возвращаемые строки данных будут допустимыми для ввода в тип столбца, объявленный в предложении `FROM`.

Примечания

Использовать `dblink` с предопределёнными запросами будет удобнее, если создать представление. Это позволит скрыть в его определении информацию о типах столбцов и не выписывать её в каждом запросе. Например:

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres options=-csearch_path=',
                'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

Примеры

```
SELECT * FROM dblink('dbname=postgres options=-csearch_path=',
                     'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
```

```
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
 byteaout   | byteaout
(14 rows)
```

dblink_exec

dblink_exec — выполняет команду в удалённой базе данных

Синтаксис

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

Описание

Функция `dblink_exec` выполняет команду (то есть любой SQL-оператор, не возвращающий строки) в удалённой базе данных.

Когда этой функции передаются два аргумента типа `text`, первый сначала рассматривается как имя постоянного подключения; если такое подключение находится, команда выполняется для него. Если не находится, первый аргумент воспринимается как строка подключения, как для функции `dblink_connect`, и заданное подключение устанавливается только на время выполнения этой команды.

Аргументы

`connname`

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

`connstr`

Строка подключения, описанная ранее для `dblink_connect`

`sql`

SQL-запрос, который вы хотите выполнить в удалённой базе данных, например `insert into foo values(0, 'a', '{"a0","b0","c0"}')`.

`fail_on_error`

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и возвращаемым значением функции будет `ERROR`.

Возвращаемое значение

Возвращает состояние (либо строку состояния команды, либо `ERROR`).

Примеры

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21, ''z'', ''{"a0","b0","c0"}'');');
   dblink_exec
-----------------
 INSERT 943366 1
(1 row)
```

```
SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21, ''z'',
 ''{"a0","b0","c0"}'');');
   dblink_exec
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values (''foo'')',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

dblink_open

dblink_open — открывает курсор в удалённой базе данных

Синтаксис

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) returns
 text
```

Описание

Функция `dblink_open()` открывает курсор в удалённой базе данных. Открытым курсором можно будет манипулировать функциями `dblink_fetch()` и `dblink_close()`.

Аргументы

`connname`

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

`cursorname`

Имя, назначаемое курсору.

`sql`

Оператор `SELECT`, который вы хотите выполнять в удалённой базе данных, например `select * from pg_class`.

`fail_on_error`

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и возвращаемым значением функции будет `ERROR`.

Возвращаемое значение

Возвращает состояние, `OK` или `ERROR`.

Примечания

Так как курсор может существовать только в рамках транзакции, функция `dblink_open` начинает явный блок транзакции (командой `BEGIN`) на удалённой стороне, если транзакция там ещё не открыта. Эта транзакция будет снова закрыта при соответствующем вызове `dblink_close`. Заметьте, что если вы с помощью `dblink_exec` изменяете данные между вызовами `dblink_open` и `dblink_close`, а затем происходит ошибка, либо если вы вызываете `dblink_disconnect` перед `dblink_close`, ваши изменения *будут потеряны*, так как транзакция будет прервана.

Примеры

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
```

```
  OK
(1 row)
```

dblink_fetch

dblink_fetch — возвращает строки из открытого курсора в удалённой базе данных

Синтаксис

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error])
 returns setof record
```

Описание

`dblink_fetch` выбирает строки из курсора, ранее открытого функцией `dblink_open`.

Аргументы

connname

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

cursorname

Имя курсора, из которого выбираются данные.

howmany

Максимальное число строк, которое нужно получить. Данная функция выбирает через курсор следующие *howmany* строк, начиная с текущей позиции курсора и двигаясь вперёд. Когда курсор доходит до конца, строки больше не выдаются.

fail_on_error

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и функция не возвращает строки.

Возвращаемое значение

Эта функция возвращает строки, выбираемые через курсор. Для использования этой функции необходимо задать ожидаемый набор столбцов, как ранее говорилось в описании `dblink`.

Примечания

При несовпадении числа возвращаемых столбцов, определённого в предложении `FROM`, с фактическим числом столбцов, возвращённых удалённым курсором, выдаётся ошибка. В этом случае удалённый курсор всё равно продвигается на столько строк, на сколько он продвинулся бы, если бы ошибка не произошла. То же самое верно для любых других ошибок, происходящих при локальной обработке результатов после выполнения удалённой команды `FETCH`.

Примеры

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname like
 ''bytea%''');
 dblink_open
-------------
```

```
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname |  source
----------+----------
 byteacat | byteacat
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname | source
----------+--------
(0 rows)
```

dblink_close

dblink_close — закрывает курсор в текущей базе данных

Синтаксис

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

Описание

`dblink_close` закрывает курсор, ранее открытый функцией `dblink_open`.

Аргументы

`connname`

Имя используемого подключения; опустите этот параметр, чтобы использовать безымянное подключение.

`cursorname`

Имя курсора, который будет закрыт.

`fail_on_error`

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и возвращаемым значением функции будет `ERROR`.

Возвращаемое значение

Возвращает состояние, `OK` или `ERROR`.

Примечания

Если вызов `dblink_open` начал явный блок транзакции и это последний открытый курсор, остававшийся в этом подключении, то `dblink_close` выполнит соответствующую команду `COMMIT`.

Примеры

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
--------------
 OK
(1 row)
```

dblink_get_connections

dblink_get_connections — возвращает имена всех открытых именованных подключений dblink

Синтаксис

```
dblink_get_connections() returns text[]
```

Описание

`dblink_get_connections` возвращает массив имён всех открытых именованных подключений `dblink`.

Возвращаемое значение

Возвращает текстовый массив имён подключений, либо NULL, если они отсутствуют.

Примеры

```
SELECT dblink_get_connections();
```

dblink_error_message

dblink_error_message — выдаёт сообщение последней ошибки для именованного подключения

Синтаксис

```
dblink_error_message(text connname) returns text
```

Описание

`dblink_error_message` извлекает самое последнее сообщение удалённой ошибки для заданного подключения.

Аргументы

`connname`

Имя используемого подключения.

Возвращаемое значение

Возвращает сообщение последней ошибки либо `OK`, если в сеансе этого подключения не было ошибок.

Примечания

Когда функцией `dblink_send_query` передаются асинхронные запросы, сообщение об ошибке, связанное с текущим подключением, может не измениться до получения от сервера ответного сообщения. Поэтому обычно до вызова `dblink_error_message` следует вызывать функцию `dblink_is_busy` или `dblink_get_result`, чтобы ошибка, возникшая при выполнении асинхронного запроса, не осталась незамеченной.

Примеры

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

dblink_send_query — передаёт асинхронный запрос в удалённую базу данных

Синтаксис

```
dblink_send_query(text connname, text sql) returns int
```

Описание

`dblink_send_query` передаёт запрос для асинхронного выполнения, то есть не дожидается получения результата. С этим подключением не должен быть связан уже выполняющийся асинхронный запрос.

После успешной передачи асинхронного запроса состояние его завершения можно проверять, вызывая функцию `dblink_is_busy`, и в итоге получать данные, вызвав `dblink_get_result`. Также можно попытаться отменить активный асинхронный запрос, вызвав `dblink_cancel_query`.

Аргументы

`connname`

Имя используемого подключения.

`sql`

Оператор SQL, который вы хотите выполнить в удалённой базе данных, например `select * from pg_class`.

Возвращаемое значение

Возвращает 1, если запрос был успешно отправлен на обработку, или 0 в противном случае.

Примеры

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

dblink_is_busy

dblink_is_busy — проверяет, не выполняется ли через подключение асинхронный запрос

Синтаксис

```
dblink_is_busy(text connname) returns int
```

Описание

`dblink_is_busy` проверяет, не выполняется ли асинхронный запрос.

Аргументы

`connname`

Имя проверяемого подключения.

Возвращаемое значение

Возвращает 1, если подключение занято, или 0 в противном случае. Если эта функция возвращает 0, гарантируется, что вызов `dblink_get_result` не будет заблокирован.

Примеры

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

dblink_get_notify — выдаёт асинхронные уведомления подключения

Синтаксис

```
dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)
dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra
 text)
```

Описание

`dblink_get_notify` выдаёт уведомления либо безымянного подключения, либо подключения с заданным именем. Чтобы получать уведомления через dblink, необходимо сначала выполнить `LISTEN`, воспользовавшись функцией `dblink_exec`. За подробностями обратитесь к LISTEN и NOTIFY.

Аргументы

`connname`

Имя именованного подключения, уведомления которого нужно получить.

Возвращаемое значение

Возвращает `setof (notify_name text, be_pid int, extra text)` или пустой набор, если уведомлений нет.

Примеры

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

dblink_get_result

dblink_get_result — получает результат асинхронного запроса

Синтаксис

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

Описание

`dblink_get_result` получает результаты асинхронного запроса, запущенного ранее вызовом `dblink_send_query`. Если запрос ещё выполняется, `dblink_get_result` будет ждать его завершения.

Аргументы

`connname`

Имя используемого подключения.

`fail_on_error`

Если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как ЗАМЕЧАНИЕ, и функция не возвращает строки.

Возвращаемое значение

Для асинхронного запроса (то есть, SQL-оператора, возвращающего строки) эта функция выдаёт строки, полученные в результате запроса. Чтобы использовать эту функцию, вы должны задать ожидаемый набор столбцов, как ранее говорилось в описании `dblink`.

Для асинхронной команды (то есть, SQL-оператора, не возвращающего строки), эта функция возвращает одну строку с одним текстовым столбцом, содержащим строку состояния команды. Для такого вызова в предложении `FROM` так же необходимо определить, что результат будет содержать один текстовый столбец.

Примечания

Эта функция *должна* вызываться, если `dblink_send_query` возвращает 1. Её нужно вызывать по одному разу для каждого отправленного запроса, а затем ещё раз для получения пустого набора данных, прежде чем подключением можно будет пользоваться снова.

Когда используются `dblink_send_query` и `dblink_get_result`, подсистема dblink получает весь набор удалённых результатов, прежде чем передавать его для локальной обработки. Если запрос возвращает большое количество строк, это может занимать много памяти в локальном сеансе. Поэтому может быть лучше открыть такой запрос как курсор, вызвав `dblink_open`, а затем выбирать результаты удобоваримыми порциями. Кроме того, можно воспользоваться простой функцией `dblink()`, которая не допускает заполнения памяти, выгружая большие наборы результатов на диск.

Примеры

```
contrib_regression=# SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
 dblink_connect
----------------
 OK
(1 row)

contrib_regression=# SELECT * FROM
```

```
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3') AS
 t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3;
 select * from foo where f1 > 6') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
 f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)
```

dblink_cancel_query

dblink_cancel_query — отменяет любой активный запрос в заданном подключении

Синтаксис

```
dblink_cancel_query(text connname) returns text
```

Описание

Функция `dblink_cancel_query` пытается отменить любой запрос, выполняющийся через заданное подключение. Заметьте, что её вызов не обязательно будет успешным (например, потому что удалённый запрос уже завершился). Запрос отмены просто увеличивает шансы того, что выполняющийся запрос будет вскоре прерван. При этом всё равно нужно завершить обычную процедуру обработки запроса, например, вызвать `dblink_get_result`.

Аргументы

`connname`

Имя используемого подключения.

Возвращаемое значение

Возвращает `OK`, если запрос отмены был отправлен, либо текст сообщения об ошибке в случае неудачи.

Примеры

```
SELECT dblink_cancel_query('dtest1');
```

dblink_get_pkey

dblink_get_pkey — возвращает позиции и имена полей первичного ключа отношения

Синтаксис

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

Описание

Функция `dblink_get_pkey` выдаёт информацию о первичном ключе отношения в локальной базе данных. Иногда это полезно при формировании запросов, отправляемых в удалённые базы данных.

Аргументы

`relname`

Имя локального отношения, например `foo` или `myschema.mytab`. Заключите его в двойные кавычки, если это имя в смешанном регистре или содержит специальные символы, например `"FooBar"`; без кавычек эта строка приводится к нижнему регистру.

Возвращаемое значение

Возвращает одну строку для каждого поля первичного ключа, либо не возвращает строк, если в отношении нет первичного ключа. Тип результирующей строки определён как

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

В столбце `position` содержится число от 1 до *`N`*; это номер поля в первичном ключе, а не номер столбца в списке столбцов таблицы.

Примеры

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE

SELECT * FROM dblink_get_pkey('foobar');
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

dblink_build_sql_insert

dblink_build_sql_insert — формирует оператор INSERT из локального кортежа, заменяя значения полей первичного ключа переданными альтернативными значениями

Синтаксис

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

Описание

Функция `dblink_build_sql_insert` может быть полезна при избирательной репликации локальной таблицы с удалённой базой данных. Она выбирает строку из локальной таблицы по заданному первичному ключу, а затем формирует SQL-команду `INSERT`, дублирующую эту строку, но заменяет в ней значения первичного ключа данными из последнего аргумента. (Чтобы получить точную копию строки, просто укажите одинаковые значения в двух последних аргументах.)

Аргументы

`relname`

Имя локального отношения, например `foo` или `myschema.mytab`. Заключите его в двойные кавычки, если это имя в смешанном регистре или содержит специальные символы, например `"FooBar"`; без кавычек эта строка приводится к нижнему регистру.

`primary_key_attnums`

Номера атрибутов (начиная с 1) полей первичного ключа, например `1 2`.

`num_primary_key_atts`

Число полей первичного ключа.

`src_pk_att_vals_array`

Значения полей первичного ключа, по которым будет выполняться поиск локального кортежа. Каждое поле здесь представляется в текстовом виде. Если локальной строки с этими значениями первичного ключа нет, выдаётся ошибка.

`tgt_pk_att_vals_array`

Значения полей первичного ключа, которые будут помещены в результирующую команду `INSERT`. Каждое поле представляется в текстовом виде.

Возвращаемое значение

Возвращает запрошенный SQL-оператор в текстовом виде.

Примечания

Начиная с PostgreSQL 9.0, номера атрибутов в *`primary_key_attnums`* воспринимаются как логические номера столбцов, соответствующие позициям столбцов в `SELECT * FROM relname`. Предыдущие версии воспринимали эти номера как физические позиции столбцов. Отличие этих подходов проявляется, когда на протяжении жизни таблицы из неё удаляются столбцы левее указанных.

Примеры

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
```

```
              dblink_build_sql_insert
--------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

dblink_build_sql_delete

dblink_build_sql_delete — формирует оператор DELETE со значениями, передаваемыми для полей первичного ключа

Синтаксис

```
dblink_build_sql_delete(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] tgt_pk_att_vals_array) returns text
```

Описание

Функция `dblink_build_sql_delete` может быть полезна при избирательной репликации локальной таблицы с удалённой базой данных. Она формирует SQL-команду `DELETE`, которая удалит строку с заданными значениями первичного ключа.

Аргументы

`relname`

Имя локального отношения, например `foo` или `myschema.mytab`. Заключите его в двойные кавычки, если это имя в смешанном регистре или содержит специальные символы, например `"FooBar"`; без кавычек эта строка приводится к нижнему регистру.

`primary_key_attnums`

Номера атрибутов (начиная с 1) полей первичного ключа, например `1 2`.

`num_primary_key_atts`

Число полей первичного ключа.

`tgt_pk_att_vals_array`

Значения полей первичного ключа, которые будут использоваться в результирующей команде `DELETE`. Каждое поле представляется в текстовом виде.

Возвращаемое значение

Возвращает запрошенный SQL-оператор в текстовом виде.

Примечания

Начиная с PostgreSQL 9.0, номера атрибутов в *`primary_key_attnums`* воспринимаются как логические номера столбцов, соответствующие позициям столбцов в `SELECT * FROM relname`. Предыдущие версии воспринимали эти номера как физические позиции столбцов. Отличие этих подходов проявляется, когда на протяжении жизни таблицы из неё удаляются столбцы левее указанных.

Примеры

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
---------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

dblink_build_sql_update

dblink_build_sql_update — формирует оператор UPDATE из локального кортежа, заменяя значения первичного ключа переданными альтернативными значениями

Синтаксис

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

Описание

Функция `dblink_build_sql_update` может быть полезна при избирательной репликации локальной таблицы с удалённой базой данных. Она выбирает строку из локальной таблицы по заданному первичному ключу, а затем формирует SQL-команду `UPDATE`, дублирующую эту строку, но заменяющую в ней значения первичного ключа данными из последнего аргумента. (Чтобы получить точную копию строки, просто укажите одинаковые значения в двух последних аргументах.) Команда `UPDATE` всегда присваивает значения всем полям строки — основное отличие этой функции от `dblink_build_sql_insert` в том, что она предполагает, что целевая строка уже существует в удалённой таблице.

Аргументы

`relname`

Имя локального отношения, например `foo` или `myschema.mytab`. Заключите его в двойные кавычки, если это имя в смешанном регистре или содержит специальные символы, например `"FooBar"`; без кавычек эта строка приводится к нижнему регистру.

`primary_key_attnums`

Номера атрибутов (начиная с 1) полей первичного ключа, например `1 2`.

`num_primary_key_atts`

Число полей первичного ключа.

`src_pk_att_vals_array`

Значения полей первичного ключа, по которым будет выполняться поиск локального кортежа. Каждое поле здесь представляется в текстовом виде. Если локальной строки с этими значениями первичного ключа нет, выдаётся ошибка.

`tgt_pk_att_vals_array`

Значения полей первичного ключа, которые будут помещены в результирующую команду `UPDATE`. Каждое поле представляется в текстовом виде.

Возвращаемое значение

Возвращает запрошенный SQL-оператор в текстовом виде.

Примечания

Начиная с PostgreSQL 9.0, номера атрибутов в *`primary_key_attnums`* воспринимаются как логические номера столбцов, соответствующие позициям столбцов в `SELECT * FROM relname`. Предыдущие версии воспринимали эти номера как физические позиции столбцов. Отличие этих подходов проявляется, когда на протяжении жизни таблицы из неё удаляются столбцы левее указанных.

Примеры

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
                   dblink_build_sql_update
-------------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.15. dict_int — пример словаря текстового поиска для целых чисел

Модуль `dict_int` представляет собой пример дополнительного шаблона словаря для полнотекстового поиска. Этот словарь был создан для управляемой индексации целых чисел (со знаком и без); он позволяет индексировать такие числа и при этом избежать чрезмерного разрастания списка уникальных слов, что значительно увеличивает скорость поиска.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.15.1. Конфигурирование

Этот словарь принимает три параметра:

- Параметр `maxlen` задаёт максимальное число цифр, из которого может состоять целое число. Значение по умолчанию — 6.
- Параметр `rejectlong` определяет, должны ли чрезмерно длинные числа усекаться или игнорироваться. Если `rejectlong` имеет значение `false` (по умолчанию), этот словарь возвращает первые `maxlen` цифр целого числа. Если `rejectlong` равен `true`, чрезмерное длинное целое воспринимается как стоп-слово, и в результате не индексируется. Заметьте, это означает, что такое целое нельзя будет найти.
- Параметр `absval` определяет, должны ли удаляться знаки «`+`» и «`-`», стоящие перед целыми числами. Значение по умолчанию — `false`. Если этот параметр имеет значение `true`, знак удаляется до того, как рассматривается ограничение `maxlen`.

F.15.2. Использование

При установке расширения `dict_int` в базе создаётся шаблон текстового поиска `intdict_template` и словарь `intdict` на его базе, с параметрами по умолчанию. Вы можете изменить параметры словаря, например так:

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

или создать новые словари на базе этого шаблона.

Протестировать этот словарь можно так:

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

Но для практического применения его нужно включить в конфигурацию текстового поиска, как описано в Главе 12. Это может выглядеть примерно так:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.16. dict_xsyn — пример словаря текстового поиска синонимов

Модуль `dict_xsyn` (Extended Synonym Dictionary, расширенный словарь синонимов) представляет собой пример дополнительного шаблона словаря для полнотекстового поиска. Этот словарь заменяет слова группами их синонимов, что позволяет находить слово по одному из его синонимов.

F.16.1. Конфигурирование

Словарь `dict_xsyn` принимает следующие параметры:

- Параметр `matchorig` определяет, будет ли словарь принимать изначальное слово. По умолчанию он включён (имеет значение `true`).
- Параметр `matchsynonyms` определяет, будет ли словарь принимать синонимы. По умолчанию он отключён (имеет значение `false`).
- Параметр `keeporig` определяет, будет ли исходное слово включаться в вывод словаря. По умолчанию он включён (имеет значение `true`).
- Параметр `keepsynonyms` определяет, будут ли в вывод словаря включаться синонимы. По умолчанию он включён (имеет значение `true`).
- Параметр `rules` задаёт базовое имя файла со списком синонимов. Этот файл должен находиться в каталоге `$SHAREDIR/tsearch_data/` (где под `$SHAREDIR` понимается каталог с общими данными инсталляции Postgres Pro). Имя файла должно заканчиваться расширением `.rules` (которое не нужно указывать в параметре `rules`).

Файл правил имеет следующий формат:

- Каждая строка представляет группу синонимов для одного слова, которое задаётся первым в этой строке. Символы разделяются пробельными символами, так что строка выглядит так:

  ```
  word syn1 syn2 syn3
  ```
- Символ решётки (`#`) обозначает начало комментария. Он может находиться в любом месте строки. Следующая за ним часть строки игнорируется.

Пример словаря можно найти в файле `xsyn_sample.rules`, устанавливаемом в `$SHAREDIR/tsearch_data/`.

F.16.2. Использование

При установке расширения `dict_xsyn` в базе создаётся шаблон текстового поиска `xsyn_template` и словарь `xsyn` на его базе, с параметрами по умолчанию. Вы можете изменить параметры словаря, например так:

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

или создать новые словари на базе этого шаблона.

Протестировать этот словарь можно так:

```
mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'word');
```

```
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false,
 MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true,
 MATCHORIG=false, KEEPSYNONYMS=false);
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {word}
```

Но для практического применения его нужно включить в конфигурацию текстового поиска, как описано в Главе 12. Это может выглядеть примерно так:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.17. dump_stat — функции выгрузки и восстановления содержимого таблицы `pg_statistic`

Модуль `dump_stat` предоставляет функции, позволяющие выгрузить и восстановить содержимое таблицы `pg_statistic`. При выполнении выгрузки/восстановления данных вы можете воспользоваться `dump_stat` для переноса существующей статистики на новый сервер, обойдясь без выполнения команды `ANALYZE` для всего кластера, что может значительно сократить время простоя для больших баз данных. Функция `dump_statistic` выдаёт операторы `INSERT`, которые затем можно применить к совместимой базе данных. Чтобы успешно восстановить статистические данные, вы должны установить это расширение и на исходном, и на целевом сервере, так как эти операторы будут задействовать функции, реализованные в `dump_stat`.

Заметьте, что определение таблицы `pg_statistic` может поменяться, в результате чего сгенерированный скрипт может оказаться несовместимым с будущими выпусками Postgres Pro.

F.17.1. Установка

Расширение `dump_stat` включено в состав Postgres Pro. Установив Postgres Pro, вы должны выполнить команду CREATE EXTENSION, чтобы подготовить `dump_stat` к работе, следующим образом:

```
CREATE EXTENSION dump_stat;
```

F.17.2. Функции

`anyarray_to_text(array anyarray) returns text`

Возвращает данный массив в виде значения `text`.

`dump_statistic() returns setof text`

Функция `dump_statistic` выгружает содержимое системного каталога `pg_statistic`. Она выдаёт `INSERT` для каждого кортежа в `pg_statistic`, кроме тех, что содержат статистические данные о таблицах в схемах `information_schema` и `pg_catalog`.

Оператор `INSERT` принимает следующую форму:

```
WITH upsert as (
  UPDATE pg_catalog.pg_statistic SET имя_столбца = выражение [, ...]
  WHERE starelid = t_relname::regclass
    AND to_attname(t_relname, staattnum) = t_attname
    AND to_atttype(t_relname, staattnum) = t_atttype
    AND stainherit = t_stainherit
  RETURNING *)
ins as (
  SELECT выражение [, ...]
  WHERE NOT EXISTS (SELECT * FROM upsert)
    AND to_attnum(t_relname, t_attname) IS NOT NULL
    AND to_atttype(t_relname, t_attname) = t_atttype)
INSERT INTO pg_catalog.pg_statistic SELECT * FROM ins;

где выражение может быть следующим:

array_in(текст_массива, имя_типа::regtype::oid, -1)
значение::имя_типа
```

Чтобы сохранить выданные операторы, направьте вывод psql в файл, используя стандартные возможности psql. Аргументы psql подробно рассматриваются в psql. Метакоманды, начинающиеся с обратной косой черты, не поддерживаются.

Например, чтобы сохранить статистику базы данных *имя_базы* в файле `dump_stat.sql`, выполните:

```
$ psql -XAtq -c "SELECT dump_statistic()" имя_базы > dump_stat.sql
```

`dump_statistic(schema_name text) returns setof text`

`dump_statistic` выгружает содержимое системного каталога `pg_statistic`. Она выдаёт оператор `INSERT` для каждого кортежа в `pg_statistic`, относящегося к какой-либо таблице в схеме `schema_name`.

`dump_statistic(schema_name text, table_name text) returns setof text`

`dump_statistic` выгружает содержимое системного каталога `pg_statistic`. Она формирует оператор `INSERT` для каждого кортежа в `pg_statistic`, относящегося к указанной таблице `schema_name.table_name`.

`dump_statistic(relation regclass) returns setof text`

`dump_statistic` выгружает содержимое системного каталога `pg_statistic`. Она формирует оператор `INSERT` для каждого кортежа в `pg_statistic`, содержащего статистические данные отношения `relation`.

`to_schema_qualified_operator(opid oid) returns text`

Выдаёт дополненное схемой имя оператора по его идентификатору `opid`. Например:

```
test=# SELECT to_schema_qualified_operator('+(int,int)'::regoperator);
          to_schema_qualified_operator
------------------------------------------------
 pg_catalog.+(pg_catalog.int4, pg_catalog.int4)
(1 row)
```

`to_schema_qualified_type(typid oid) returns text`

Выдаёт дополненное схемой имя типа по его идентификатору `typid`.

`to_schema_qualified_relation(relid oid) returns text`

Выдаёт дополненное схемой имя отношения по его идентификатору `relid`.

`anyarray_elemtype(arr anyarray) returns oid`

Возвращает тип элемента массива, заданного по `oid`. Например:

```
test=# SELECT anyarray_elemtype(array_in('{1,2,3}', 'int'::regtype, -1));
 anyarray_elemtype
-------------------
                23
(1 row)
```

`to_attname(relation regclass, colnum int2) returns text`

Для отношения с заданным именем `relation` и номером столбца `colnum` возвращает имя столбца в виде значения `text`.

`to_attnum(relation regclass, col text) returns int2`

Для отношения с заданным именем `relation` и столбца с именем `col` возвращает номер столбца в виде значения `int2`.

`to_atttype(relation regclass, col text) returns text`

Для отношения с заданным именем `relation` и столбца с именем `col` возвращает дополненное схемой имя типа столбца в виде значения `text`.

`to_atttype(relation regclass, colnum int2) returns text`

Для отношения с заданным именем `relation` и столбца с номером `colnum` возвращает дополненное схемой имя типа столбца в виде значения `text`.

`to_namespace(nsp text) returns oid`

`to_namespace` повторяет поведение приведения к типу `regnamespace`, которое отсутствует в PostgreSQL версии 9.4 (и предыдущих). Эта функция возвращает `oid` данной схемы.

`get_namespace(relation oid) returns oid`

`get_namespace` возвращает схему заданного отношения в виде `oid`.

F.18. earthdistance — вычисление расстояний между точками на Земле

Модуль `earthdistance` реализует два разных варианта вычисления ортодромии (расстояния между точками на поверхности Земли). Первый описанный вариант зависит от модуля `cube`. Второй вариант основан на встроенном типе данных `point`, в котором в качестве координат задаются широта и долгота.

В этом модуле Земля считается идеальной сферой. (Если для вас это слишком грубо, обратите внимание на проект *PostGIS*.)

Прежде чем устанавливать `earthdistance`, вы должны установить модуль `cube` (хотя можно воспользоваться указанием `CASCADE` команды `CREATE EXTENSION` и установить сразу оба расширения).

> **Внимание**
>
> Расширения `earthdistance` и `cube` настоятельно рекомендуется устанавливать в одну схему, и при этом в данной схеме недоверенные пользователи не должны в настоящем и будущем иметь право CREATE. В противном случае, если в схеме `earthdistance` окажутся объекты, созданные злонамеренным пользователем, возможна угроза безопасности. Более того, используя функции `earthdistance` после установки расширения, следует ограничивать путь поиска только доверенными схемами.

F.18.1. Земные расстояния по кубам

Данные хранятся в кубах, представляющих точки (оба угла куба совпадают) по 3 координатам, выражающим смещения x, y и z от центра Земли. Этот модуль предоставляет *домен* `earth` на базе `cube`, включающий проверки того, что значение соответствует этим ограничениям и представляет точку, достаточно близкую к сферической поверхности Земли.

Радиус Земли выдаёт функция `earth()` (в метрах). Изменив одну эту функцию, вы можете сделать так, чтобы модуль работал с другими единицами, либо выдать другое значение радиуса, которое кажется вам более подходящим.

Этот пакет может также применяться и для астрономических расчётов. Астрономы обычно меняют функцию `earth()`, чтобы она возвращала радиус, равный `180/pi()`, и расстояния в результате выдавались в градусах.

В этом модуле реализованы функции для ввода данных, выражающих широту и долготу (в градусах), для вывода ширины и долготы, для вычисления ортодромии между двумя точками и простого указания окружающего прямоугольника, что полезно для поиска по индексу.

Предоставляемые этим модулем функции показаны в Таблица F.8.

Таблица F.8. Функции земных расстояний по кубам

Функция Описание
`earth` () → `float8` Возвращает предполагаемый радиус Земли.
`sec_to_gc` (`float8`) → `float8` Переводит расстояние по обычной прямой (по секущей) между двумя точками на поверхности Земли в расстояние между ними по сфере.
`gc_to_sec` (`float8`) → `float8` Переводит расстояние по сфере между двумя точками на поверхности Земли в расстояние по обычной прямой (по секущей) между ними.

Функция Описание
`ll_to_earth ( float8, float8 ) → earth` Возвращает положение точки на поверхности Земли по заданной широте (аргумент 1) и долготе (аргумент 2) в градусах.
`latitude ( earth ) → float8` Возвращает широту (в градусах) точки на поверхности Земли.
`longitude ( earth ) → float8` Возвращает долготу (в градусах) точки на поверхности Земли.
`earth_distance ( earth, earth ) → float8` Возвращает расстояние по сфере между двумя точками на поверхности Земли.
`earth_box ( earth, float8 ) → cube` Выдаёт охватывающий куб, подходящий для поиска по индексу с применением реализованного для типа `cube` оператора `@>` для точек в пределах заданной ортодромии от цели. Некоторые точки в этом кубе будут отстоять от цели дальше, чем на заданную ортодромию, поэтому в запрос нужно включить вторую проверку с функцией `earth_distance` .

F.18.2. Земные расстояния по точкам

Вторая часть этого модуля основана на представлении точек на Земле в виде значений типа `point`, в которых первый компонент представляет долготу в градусах, а второй — широту. Точки воспринимаются как (долгота, широта), а не наоборот, так как долгота ближе к интуитивному представлению как оси X, а широта — оси Y.

В модуле реализован один оператор, показанный в Таблице F.9.

Таблица F.9. Операторы земных расстояний по точкам

Оператор Описание
`point <@> point → float8` Вычисляет расстояние в сухопутных милях между точками на поверхности Земли.

Заметьте, что в этой части модуля, в отличие от части, построенной на `cube`, единицы зашиты жёстко: изменение функции `earth()` не повлияет на результат этого оператора.

Представление в виде долготы/широты плохо тем, что вам придётся учитывать граничные условия возле полюсов и в районе +/- 180 градусов долготы. Представление на базе `cube` лишено таких нарушений непрерывности.

F.19. fasttrun — транзакционно-небезопасная функция для усечения временных таблиц

Модуль `fasttrun` предоставляет транзакционно-небезопасную функцию для усечения временных таблиц, предотвращающую разрастание каталога pg_class.

Этот модуль требуется для поддержки системы 1С:Предприятие.

Операция быстрого усечения не является транзакционной, так что её действие нельзя отменить и оно немедленно становится видимым во всех сеансах независимо от уровня изоляции.

F.19.1. Функция

Пример вызова функции:

```
select fasttruncate('TABLE_NAME');
```

F.19.2. Тестовый пример

Для тестирования вы можете использовать этот пример:

```
create or replace function f() returns void as $$
begin
  for i in 1..1000
    loop
      PERFORM fasttruncate('tt1');
    end loop;
  end;
$$ language plpgsql;
```

F.19.3. Авторы

Фёдор Сигаев `<teodor@sigaev.ru>`

F.20. file_fdw — обращение к файлам данных в файловой системе сервера

Модуль `file_fdw` реализует обёртку сторонних данных `file_fdw`, с помощью которой можно обращаться к файлам данных в файловой системе сервера или выполнять программы на сервере и читать их вывод. Файлы и вывод программ должны быть в формате, который понимает команда `COPY FROM`; он рассматривается в описании COPY. В настоящее время файлы доступны только для чтения.

Для сторонней таблицы, создаваемой через эту обёртку, можно задать следующие параметры:

`filename`

Определяет имя файла, который нужно прочитать. При указании относительного пути он рассматривается от каталога данных. Вы должны определить либо параметр `filename`, либо `program`, но не оба сразу.

`program`

Определяет команду, которая будет выполнена. Поток стандартного вывода этой команды будет прочитан так же, как и с `COPY FROM PROGRAM`. Необходимо определить либо параметр `program`, либо `filename`, но не оба сразу.

`format`

Определяет формат файла (аналогично указанию `FORMAT` в команде `COPY`).

`header`

Указывает, что данные содержат строку заголовка с именами столбцов (аналогично указанию `HEADER` в команде `COPY`).

`delimiter`

Задаёт символ, разделяющий столбцы в данных (аналогично указанию `DELIMITER` в команде `COPY`).

`quote`

Задаёт символ, используемый для заключения данных в кавычки (аналогично указанию `QUOTE` в команде `COPY`).

`escape`

Задаёт символ, используемый для экранирования данных (аналогично указанию `ESCAPE` в команде `COPY`).

`null`

Определяет строку, задающую значение `NULL` в данных (аналогично указанию `NULL` в команде `COPY`).

`encoding`

Задаёт кодировку данных (аналогично указанию `ENCODING` в команде `COPY`).

Заметьте, что хотя `COPY` принимает указания, такие как `HEADER`, без соответствующего значения, синтаксис обёртки сторонних данных требует, чтобы значение присутствовало во всех случаях. Чтобы активировать указания `COPY`, которым значение обычно не передаётся, им можно просто передать значение TRUE, так как все они являются логическими.

Для столбцов сторонней таблицы, создаваемой через эту обёртку, можно задать следующие параметры:

`force_not_null`

Логическое значение. Если true, то значение столбца не должно сверяться со значением NULL (заданным в параметре `null` на уровне таблицы). Аналогично включению столбца в список указания `FORCE_NOT_NULL` команды `COPY`.

`force_null`

Логическое значение. Если true, значения столбцов нужно сверять со значением NULL (заданным в параметре `NULL`), даже если они заключены в кавычки. Без этого параметра только значения без кавычек, соответствующие значению `null`, будут возвращаться как NULL. Аналогично включению столбца в список указания `FORCE_NULL` команды `COPY`.

В настоящее время `file_fdw` не поддерживает указание `FORCE_QUOTE` команды `COPY`.

Перечисленные параметры применимы только для сторонних таблиц или их столбцов. Их нельзя указать для обёртки сторонних данных `file_fdw`, серверов или сопоставлений пользователей, использующих эту обёртку.

Для изменения параметров, определяемых для таблицы, требуется быть суперпользователем или иметь права роли `pg_read_server_files` (для указания имени файла) или роли `pg_execute_server_program` (для указания программы). Это сделано в целях безопасности: только избранные пользователи должны выбирать, какой файл читать или какую программу запускать. В принципе право изменения остальных параметров можно предоставить и обычным пользователям, но в настоящий момент это не реализовано.

Задавая параметр `program`, помните, что эта строка выполняется оболочкой ОС. Если вы хотите передавать заданной команде параметры из недоверенного источника, позаботьтесь об исключении или экранировании всех символов, которые могут иметь особое назначение в оболочке. По соображениям безопасности лучше, чтобы эта командная строка была фиксированной или как минимум в ней не передавались данные, поступающие от пользователя.

Для сторонних таблиц, работающих через `file_fdw`, команда `EXPLAIN` показывает имя используемого файла или запускаемой программы. Если не указывать `COSTS OFF`, то выводится и размер файла (в байтах).

Пример F.5. Создание сторонней таблицы для журнала сервера Postgres Pro

Одно из очевидных применений `file_fdw` — это предоставление доступа к журналу сообщений Postgres Pro как к таблице. Для этого необходимо предварительно настроить вывод сообщений в файл CSV (дальше мы будем считать, что это файл `pglog.csv`). Сначала установите расширение `file_fdw`:

```
CREATE EXTENSION file_fdw;
```

Затем создайте сторонний сервер:

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

Всё готово для создания сторонней таблицы. В команде `CREATE FOREIGN TABLE` нужно перечислить столбцы таблицы, указать файл CSV и его формат:

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
```

```
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text,
  leader_pid integer,
  query_id bigint
) SERVER pglog
OPTIONS ( filename 'log/pglog.csv', format 'csv' );
```

Вот и всё. Теперь для просмотра журнала сервера можно просто выполнять запросы к таблице. В производственной среде, разумеется, ещё потребуется как-то учесть ротацию файлов журнала.

Пример F.6. Создать внешнюю таблицу с параметром для столбца

Чтобы задать параметр `force_null` для столбца, используйте ключевое слово `OPTIONS`.

```
CREATE FOREIGN TABLE films (
 code char(5) NOT NULL,
 title text NOT NULL,
 rating text OPTIONS (force_null 'true')
) SERVER film_server
OPTIONS ( filename 'films/db.csv', format 'csv' );
```

F.21. fulleq — дополнительный оператор равенства для совместимости с Microsoft SQL Server

Модуль `fulleq` предоставляет дополнительный оператор равенства для совместимости с Microsoft SQL Server.

Этот модуль требуется для поддержки системы 1С:Предприятие.

F.21.1. Обзор

Оператор равенства в Postgres Pro определён так, что результатом сравнения двух NULL оказывается NULL. Однако в СУБД семейства Microsoft SQL исторически оператор равенства ведёт себя по-другому и возвращает TRUE, когда оба его операнда равны NULL. Данный модуль предоставляет такой оператор с поведением, принятым в MS SQL.

F.21.2. Оператор fulleq

Оператор == определён для следующих типов данных:

- bool
- bytea
- char
- name
- int2
- int4
- int8
- int2vector
- text
- oid
- xid
- cid
- oidvector
- float4
- float8
- abstime
- reltime
- macaddr
- inet
- cidr
- varchar
- date
- time
- timestamp
- timestamptz
- interval
- timetz

F.21.3. Авторы

Фёдор Сигаев `<teodor@sigaev.ru>`

F.22. fuzzystrmatch — вычисление схожести и расстояния между строками

Модуль `fuzzystrmatch` содержит несколько функций для вычисления схожести и расстояния между строками.

> **Внимание**
>
> В настоящее время функции `soundex`, `metaphone`, `dmetaphone` и `dmetaphone_alt` плохо работают с многобайтными кодировками (в частности, с UTF-8). Используйте функции `daitch_mokotoff` или `levenshtein` в работе с такими данными.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.22.1. Soundex

Система Soundex позволяет вычислить похожие по звучанию имена, приводя их к одинаковым кодам. Изначально она использовалась для обработки данных переписи населения США в 1880, 1900 и 1910 г. Заметьте, что эта система не очень полезна для неангloязычных имён.

Модуль `fuzzystrmatch` предоставляет две функции для работы с кодами Soundex:

```
soundex(text) returns text
difference(text, text) returns int
```

Функция `soundex` преобразует строку в код Soundex. Функция `difference` преобразует две строки в их коды Soundex и затем сообщает количество совпадающих позиций в этих кодах. Так как коды Soundex состоят из четырёх символов, результатом может быть число от нуля до четырёх (0 обозначает полное несоответствие, а 4 — точное совпадение). (Таким образом, имя этой функции не вполне корректное — лучшим именем для неё было бы `similarity`.)

Несколько примеров использования:

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.22.2. Daitch-Mokotoff Soundex

Как и исходная система Soundex, система Daitch-Mokotoff Soundex сопоставляет похожие по звучанию имена и преобразует их в один и тот же код. Однако Daitch-Mokotoff Soundex значительно полезнее для неанглийских имён, чем исходная система. Основные улучшения по сравнению с исходной системой перечислены ниже:

- Код основан на первых шести значимых буквах, а не на четырёх.
- Буква или комбинация букв сопоставляется с десятью возможными кодами, а не с семью.
- Если две последовательные буквы имеют один звук, они заменяются цифрой.
- Когда буква или комбинация букв могут иметь разные звуки, генерируется несколько кодов, чтобы охватить все возможные варианты.

Эта функция генерирует коды Daitch-Mokotoff Soundex для входных данных:

```
daitch_mokotoff(source text) returns text[]
```

Результат может содержать один или несколько кодов в зависимости от количества возможных вариантов произношения, поэтому он представляется в виде массива.

Поскольку код Daitch-Mokotoff Soundex состоит всего из 6 цифр, рекомендуется передавать в *source* одно слово или имя.

Несколько примеров:

```
SELECT daitch_mokotoff('George');
 daitch_mokotoff
-----------------
 {595000}

SELECT daitch_mokotoff('John');
 daitch_mokotoff
-----------------
 {160000,460000}

SELECT daitch_mokotoff('Bierschbach');
                      daitch_mokotoff
-----------------------------------------------------------
 {794575,794574,794750,794740,745750,745740,747500,747400}

SELECT daitch_mokotoff('Schwartzenegger');
 daitch_mokotoff
-----------------
 {479465}
```

Для сопоставления отдельных имён возвращаемые текстовые массивы могут сравниваться напрямую с использованием оператора `&&`: любое пересечение можно считать совпадением. Для повышения эффективности можно использовать индекс GIN, см. Раздел 62.4 и следующий пример:

```
CREATE TABLE s (nm text);
CREATE INDEX ix_s_dm ON s USING gin (daitch_mokotoff(nm)) WITH (fastupdate = off);

INSERT INTO s (nm) VALUES
  ('Schwartzenegger'),
  ('John'),
  ('James'),
  ('Steinman'),
  ('Steinmetz');

SELECT * FROM s WHERE daitch_mokotoff(nm) && daitch_mokotoff('Swartzenegger');
SELECT * FROM s WHERE daitch_mokotoff(nm) && daitch_mokotoff('Jane');
SELECT * FROM s WHERE daitch_mokotoff(nm) && daitch_mokotoff('Jens');
```

Для индексации и сопоставления любого количества имён в любом порядке можно использовать функции полнотекстового поиска. См. Главу 12 и следующий пример:

```
CREATE FUNCTION soundex_tsvector(v_name text) RETURNS tsvector
```

```
BEGIN ATOMIC
  SELECT to_tsvector('simple',
                     string_agg(array_to_string(daitch_mokotoff(n), ' '), ' '))
  FROM regexp_split_to_table(v_name, '\s+') AS n;
END;

CREATE FUNCTION soundex_tsquery(v_name text) RETURNS tsquery
BEGIN ATOMIC
  SELECT string_agg('(' || array_to_string(daitch_mokotoff(n), '|') || ')',
 '&')::tsquery
  FROM regexp_split_to_table(v_name, '\s+') AS n;
END;

CREATE TABLE s (nm text);
CREATE INDEX ix_s_txt ON s USING gin (soundex_tsvector(nm)) WITH (fastupdate = off);

INSERT INTO s (nm) VALUES
  ('John Doe'),
  ('Jane Roe'),
  ('Public John Q.'),
  ('George Best'),
  ('John Yamson');

SELECT * FROM s WHERE soundex_tsvector(nm) @@ soundex_tsquery('john');
SELECT * FROM s WHERE soundex_tsvector(nm) @@ soundex_tsquery('jane doe');
SELECT * FROM s WHERE soundex_tsvector(nm) @@ soundex_tsquery('john public');
SELECT * FROM s WHERE soundex_tsvector(nm) @@ soundex_tsquery('besst, giorgio');
SELECT * FROM s WHERE soundex_tsvector(nm) @@ soundex_tsquery('Jameson John');
```

Если желательно избежать пересчёта кодов Soundex при перепроверке индекса, вместо индекса по выражению можно использовать индекс по отдельному столбцу. Для этого можно использовать хранимый генерируемый столбец; см. Раздел 5.4.

F.22.3. Левенштейн

Эта функция вычисляет расстояние Левенштейна между двумя строками:

```
levenshtein(source text, target text, ins_cost int, del_cost int, sub_cost int) returns
 int
levenshtein(source text, target text) returns int
levenshtein_less_equal(source text, target text, ins_cost int, del_cost int, sub_cost
 int, max_d int) returns int
levenshtein_less_equal(source text, target text, max_d int) returns int
```

И в `source`, и в `target` может быть передана любая строка, отличная от NULL, не длиннее 255 символов. Параметры стоимости (ins_cost, del_cost, sub_cost) определяют цену добавления, удаления или замены символов, соответственно. Эти параметры можно опустить, как во второй версии функции; в этом случае все они по умолчанию равны 1.

Функция `levenshtein_less_equal` является ускоренной версией функции Левенштейна, предназначенной для использования, только когда интерес представляют небольшие расстояния. Если фактическое расстояние меньше или равно `max_d`, то `levenshtein_less_equal` возвращает точное его значение; в противном случае она возвращает значение, большее чем `max_d`. Если значение `max_d` отрицательное, она работает так же, как функция `levenshtein`.

Примеры:

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
```

```
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2, 1, 1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive', 2);
 levenshtein_less_equal
------------------------
                      3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive', 4);
 levenshtein_less_equal
------------------------
                      4
(1 row)
```

F.22.4. Metaphone

Metaphone, как и Soundex, построен на идее составления кода, представляющего входную строку. Две строки признаются похожими, если их коды совпадают.

Эта функция вычисляет код метафона входной строки:

```
metaphone(source text, max_output_length int) returns text
```

В качестве `source` должна передаваться строка, отличная от NULL, не длиннее 255 символов. Параметр `max_output_length` задаёт максимальную длину выходного кода метафона; если код оказывается длиннее, он обрезается до этой длины.

Пример:

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
```

F.22.5. Double Metaphone

Алгоритм Double Metaphone (Двойной метафон) вычисляет две строки «похожего звучания» для заданной строки — «первичную» и «альтернативную». В большинстве случаев они совпадают, но для неанглоязычных имён в особенности они могут быть весьма различными, в зависимости от произношения. Эти функции вычисляют первичный и альтернативный коды:

```
dmetaphone(source text) returns text
dmetaphone_alt(source text) returns text
```

Длина входных строк может быть любой.

Пример:

```
test=# SELECT dmetaphone('gumbo');
 dmetaphone
------------
```

```
 KMP
(1 row)
```

F.23. hstore — тип данных `hstore` для хранения пар ключ-значение

Этот модуль реализует тип данных `hstore` для хранения пар ключ-значение внутри одного значения Postgres Pro. Это может быть полезно в самых разных сценариях, например для хранения строк со множеством редко анализируемых атрибутов или частично структурированных данных. Ключи и значения задаются простыми текстовыми строками.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.23.1. Внешнее представление `hstore`

Текстовое представление типа `hstore`, применяемое для ввода и вывода, включает ноль или более пар *`ключ`* `=>` *`значение`*, разделённых запятыми. Несколько примеров:

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

Порядок пар не имеет значения (и может не воспроизводиться при выводе). Пробелы между парами и вокруг знака `=>` игнорируются. Ключи и значения, содержащие пробелы, запятые и знаки `=` или `>`, нужно заключать в двойные кавычки. Если в ключ или значение нужно вставить символ кавычек или обратную косую черту, добавьте перед ним обратную косую черту.

Все ключи в `hstore` уникальны. Если вы объявите тип `hstore` с дублирующимися ключами, в `hstore` будет сохранён только один ключ без гарантии определённого выбора:

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
----------
 "a"=>"1"
```

В качестве значения (но не ключа) может задаваться SQL `NULL`. Например:

```
key => NULL
```

В ключевом слове `NULL` регистр не имеет значения. Если требуется, чтобы текст `NULL` воспринимался как обычная строка «NULL», заключите его в кавычки.

> **Примечание**
>
> Учтите, что когда текстовый формат `hstore` используется для ввода данных, он применяется *до* обработки кавычек или спецсимволов. Таким образом, если значение `hstore` передаётся в параметре, дополнительная обработка не требуется. Но если вы передаёте его в виде строковой константы, то все символы апострофов и (в зависимости от параметра конфигурации `standard_conforming_strings`) обратной косой черты нужно корректно экранировать. Подробнее о записи строковых констант можно узнать в Подразделе 4.1.2.1.

При выводе значения и ключи всегда заключаются в кавычки, даже когда без этого можно обойтись.

F.23.2. Операторы и функции `hstore`

Реализованные в модуле `hstore` операторы перечислены в Таблице F.10, функции — в Таблице F.11.

Таблица F.10. Операторы `hstore`

Оператор Описание Примеры
`hstore -> text → text`

Оператор **Описание** **Примеры**
Выдаёт значение для заданного ключа, или `NULL`, если он отсутствует. `'a=>x, b=>y'::hstore -> 'a'` → `x`

`hstore -> text[]` → `text[]`
Выдаёт значения, связанные с заданными ключами, или `NULL`, если таких ключей нет.
`'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a']` → `{"z","x"}`

`hstore || hstore` → `hstore`
Соединяет два набора `hstore`.
`'a=>b, c=>d'::hstore || 'c=>x, d=>q'::hstore` → `"a"=>"b", "c"=>"x", "d"=>"q"`

`hstore ? text` → `boolean`
Набор `hstore` включает ключ?
`'a=>1'::hstore ? 'a'` → `t`

`hstore ?& text[]` → `boolean`
Набор `hstore` содержит все указанные ключи?
`'a=>1,b=>2'::hstore ?& ARRAY['a','b']` → `t`

`hstore ?| text[]` → `boolean`
Набор `hstore` содержит какой-либо из указанных ключей?
`'a=>1,b=>2'::hstore ?| ARRAY['b','c']` → `t`

`hstore @> hstore` → `boolean`
Левый операнд содержит правый?
`'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1'` → `t`

`hstore <@ hstore` → `boolean`
Левый операнд содержится в правом?
`'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL'` → `f`

`hstore - text` → `hstore`
Удаляет ключ из левого операнда.
`'a=>1, b=>2, c=>3'::hstore - 'b'::text` → `"a"=>"1", "c"=>"3"`

`hstore - text[]` → `hstore`
Удаляет ключи из левого операнда.
`'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b']` → `"c"=>"3"`

`hstore - hstore` → `hstore`
Удаляет из левого операнда пары ключ-значение, совпадающие с парами в правом.
`'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore` → `"a"=>"1", "c"=>"3"`

`anyelement #= hstore` → `anyelement`
Заменяет поля в левом операнде, имеющем составной тип, соответствующими значениями из `hstore`.
`ROW(1,3) #= 'f1=>11'::hstore` → `(11,3)`

`%% hstore` → `text[]`
Преобразует `hstore` в массив перемежающихся ключей и значений.
`%% 'a=>foo, b=>bar'::hstore` → `{a,foo,b,bar}`

`%# hstore` → `text[]`
Преобразует `hstore` в двумерный массив ключей-значений.
`%# 'a=>foo, b=>bar'::hstore` → `{{a,foo},{b,bar}}`

Таблица F.11. Функции `hstore`

Функция **Описание** **Примеры**

`hstore` (`record`) → `hstore`
Формирует `hstore` из записи или кортежа.
`hstore(ROW(1,2))` → `"f1"=>"1", "f2"=>"2"`

`hstore` (`text[]`) → `hstore`
Формирует `hstore` из двумерного массива или из массива, содержащего перемежающиеся ключи-значения.
`hstore(ARRAY['a','1','b','2'])` → `"a"=>"1", "b"=>"2"`
`hstore(ARRAY[['c','3'],['d','4']])` → `"c"=>"3", "d"=>"4"`

`hstore` (`text[]`, `text[]`) → `hstore`
Формирует `hstore` из отдельных массивов ключей и значений.
`hstore(ARRAY['a','b'], ARRAY['1','2'])` → `"a"=>"1", "b"=>"2"`

`hstore` (`text`, `text`) → `hstore`
Формирует `hstore` с одним элементом.
`hstore('a', 'b')` → `"a"=>"b"`

`akeys` (`hstore`) → `text[]`
Извлекает ключи из `hstore` в виде массива.
`akeys('a=>1,b=>2')` → `{a,b}`

`skeys` (`hstore`) → `setof text`
Извлекает ключи из `hstore` в виде множества.
`skeys('a=>1,b=>2')` →
`a`
`b`

`avals` (`hstore`) → `text[]`
Извлекает значения из `hstore` в виде массива.
`avals('a=>1,b=>2')` → `{1,2}`

`svals` (`hstore`) → `setof text`
Извлекает значения `hstore` в виде множества.
`svals('a=>1,b=>2')` →
`1`
`2`

`hstore_to_array` (`hstore`) → `text[]`
Извлекает ключи и значения из `hstore` в виде массива перемежающихся ключей и значений.
`hstore_to_array('a=>1,b=>2')` → `{a,1,b,2}`

`hstore_to_matrix` (`hstore`) → `text[]`
Извлекает ключи и значения из `hstore` в виде двумерного массива.
`hstore_to_matrix('a=>1,b=>2')` → `{{a,1},{b,2}}`

`hstore_to_json` (`hstore`) → `json` Преобразует `hstore` в `json`, переводя все отличные от NULL значения в строки JSON. Эта функция применяется неявно, когда значение `hstore` приводится к типу `json`.

Функция **Описание** **Примеры**
`hstore_to_json('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')` → `{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}`

`hstore_to_jsonb` (`hstore`) → `jsonb`
Преобразует `hstore` в `jsonb`, переводя все отличные от NULL значения в строки JSON. Эта функция применяется неявно, когда значение `hstore` приводится к типу `jsonb`.
`hstore_to_jsonb('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')` → `{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}`

`hstore_to_json_loose` (`hstore`) → `json`
Преобразует `hstore` в `json`, по возможности распознавая числовые и логические значения и передавая их в JSON без кавычек.
`hstore_to_json_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')` → `{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}`

`hstore_to_jsonb_loose` (`hstore`) → `jsonb`
Преобразует `hstore` в `jsonb`, по возможности распознавая числовые и логические значения и передавая их в JSON без кавычек.
`hstore_to_jsonb_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')` → `{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}`

`slice` (`hstore`, `text[]`) → `hstore`
Извлекает из `hstore` подмножество, содержащее только заданные ключи.
`slice('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x'])` → `"b"=>"2", "c"=>"3"`

`each` (`hstore`) → `setof record` (*`key`* `text`, *`value`* `text`) Извлекает ключи и значения из `hstore` в виде набора записей. `select * from each('a=>1,b=>2')` →

```
 key | value
-----+-------
 a   | 1
 b   | 2
```

`exist` (`hstore`, `text`) → `boolean`
Набор `hstore` включает ключ?
`exist('a=>1', 'a')` → `t`

`defined` (`hstore`, `text`) → `boolean`
Значение `hstore` содержит отличное от `NULL` значение для заданного ключа?
`defined('a=>NULL', 'a')` → `f`

`delete` (`hstore`, `text`) → `hstore`
Удаляет пару с соответствующим ключом.
`delete('a=>1,b=>2', 'b')` → `"a"=>"1"`

`delete` (`hstore`, `text[]`) → `hstore`
Удаляет пары с соответствующими ключами.
`delete('a=>1,b=>2,c=>3', ARRAY['a','b'])` → `"c"=>"3"`

`delete` (`hstore`, `hstore`) → `hstore`

Функция Описание Примеры
Удаляет пары, соответствующие парам во втором аргументе. `delete('a=>1,b=>2', 'a=>4,b=>2'::hstore)` → `"a"=>"1"`

`populate_record ( anyelement, hstore ) → anyelement`
Заменяет поля в левом операнде, имеющем составной тип, соответствующими значениями из `hstore`.
`populate_record(ROW(1,2), 'f1=>42'::hstore)` → `(42,2)`

Помимо этих операторов и функций, со значениями типа `hstore` работает операция обращения по индексу, наделяющая их качествами ассоциативных массивов. Можно указать только один индекс типа `text`; он воспринимается как ключ, для которого сохраняется или извлекается соответствующее значение. Например:

```
CREATE TABLE mytable (h hstore);
INSERT INTO mytable VALUES ('a=>b, c=>d');
SELECT h['a'] FROM mytable;
 h
---
 b
(1 row)

UPDATE mytable SET h['c'] = 'new';
SELECT h FROM mytable;
          h
----------------------
 "a"=>"b", "c"=>"new"
(1 row)
```

При обращении по индексу возвращается `NULL`, если индекс — `NULL` или указанный ключ не существует в `hstore`. (Таким образом, обращение по индексу не сильно отличается от оператора `->`.) Изменение по индексу не выполняется, если индекс — `NULL`; в остальных случаях заменяется значение указанного ключа (если такой ключ не существует, он добавляется в `hstore`).

F.23.3. Индексы

Тип `hstore` поддерживает индексы GiST и GIN для операторов `@>`, `?`, `?&` и `?|`. Например:

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

Класс операторов GiST `gist_hstore_ops` аппроксимирует набор пар ключ-значение в виде сигнатуры битовой карты. В его необязательном целочисленном параметре `siglen` можно задать размер сигнатуры в байтах. Параметр может принимать значения от 1 до 2024, по умолчанию он равен 16. При увеличении размера сигнатуры поиск работает точнее (сканируется меньшая область в индексе и меньше страниц кучи), но сам индекс становится больше.

Пример создания такого индекса с длиной сигнатуры 32 байта:

```
CREATE INDEX hidx ON testhstore USING GIST (h gist_hstore_ops(siglen=32));
```

Тип `hstore` также поддерживает индексы `btree` и `hash` для оператора `=`. Это позволяет объявлять столбцы `hstore` как уникальные (`UNIQUE`) и использовать их в выражениях `GROUP BY`, `ORDER BY` или `DISTINCT`. Порядок сортировки значений `hstore` не имеет практического смысла, но эти индексы могут быть полезны для поиска по равенству. Индексы для сравнений (с помощью `=`) можно создать так:

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.23.4. Примеры

Присваивание значения существующему или новому ключу:

```
UPDATE tab SET h['c'] = '3';
```

Другой способ сделать то же самое:

```
UPDATE tab SET h = h || hstore('c', '3');
```

Если в одной операции нужно добавить или изменить несколько ключей, эффективнее использовать конкатенацию, а не обращение по индексу:

```
UPDATE tab SET h = h || hstore(array['q', 'w'], array['11', '12']);
```

Удаление ключа:

```
UPDATE tab SET h = delete(h, 'k1');
```

Приведение типа `record` к типу `hstore`:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                   hstore
---------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

Приведение типа `hstore` к предопределённому типу `record`:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

Изменение существующей записи по данным из `hstore`:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.23.5. Статистика

Тип `hstore`, вследствие присущей ему либеральности, может содержать множество самых разных ключей. Контроль допустимости ключей является задачей приложения. Следующие примеры демонстрируют несколько приёмов проверки ключей и получения статистики.

Простой пример:

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

С таблицей:

```
CREATE TABLE stat AS SELECT (each(h)).key, (each(h)).value FROM testhstore;
```

Актуальная статистика:

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
...................
```

F.23.6. Совместимость

Начиная с PostgreSQL 9.0, `hstore` использует внутреннее представление, отличающееся от предыдущих версий. Это не проблема при обновлении путём выгрузки/перезагрузки данных, так как текстовое представление (используемое при выгрузке) не меняется.

В случае двоичного обновления обратная совместимость поддерживается благодаря тому, что новый код понимает данные в старом формате. При таком обновлении возможно небольшое снижение производительности при обработке данных, которые ещё не были изменены новым кодом. Все значения в столбце таблицы можно обновить принудительно, выполнив следующий оператор `UPDATE`:

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

Это можно сделать и так:

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

Вариант с командой `ALTER TABLE` требует блокировки таблицы в режиме `ACCESS EXCLUSIVE`, но не приводит к раздуванию таблицы из-за старых версий строк.

F.23.7. Трансформации

Также имеются дополнительные расширения, реализующие трансформации типа `hstore` для языков PL/Perl и PL/Python. Расширения для PL/Perl называются `hstore_plperl` и `hstore_plperlu` для доверенного и недоверенного PL/Perl, соответственно. Если вы установите эти трансформации и укажете их при создании функции, значения `hstore` будут отображаться в хеши Perl. Расширение для PL/Python называется `hstore_plpython3u`. Если вы воспользуетесь им, значения `hstore` будут отображаться в словари Python.

Внимание

Расширения, реализующие трансформации, настоятельно рекомендуется устанавливать в одну схему с `hstore`. Выбор какой-либо другой схемы, которая может содержать объекты, созданные злонамеренным пользователем, чреват угрозами безопасности во время установки расширения.

F.23.8. Авторы

Олег Бартунов `<oleg@sai.msu.su>`, Москва, Московский Государственный Университет, Россия

Фёдор Сигаев `<teodor@sigaev.ru>`, Москва, ООО «Дельта-Софт», Россия

Дополнительные улучшения внёс Эндрю Гирт `<andrew@tao11.riddles.org.uk>`, Великобритания

F.24. Модули словарей Hunspell

Эти модули предоставляют словари Hunspell для разных языков. После установки модуля в базу командой `CREATE EXTENSION` в схеме public появляются объекты конфигурации и словаря текстового поиска.

Таблица F.12. Модули

Язык	Название расширения	Имя словаря	Имя конфигурации
Американский английский	`hunspell_en_us`	`english_hunspell`	`english_hunspell`
Нидерландский	`hunspell_nl_nl`	`dutch_hunspell`	`dutch_hunspell`
Французский	`hunspell_fr`	`french_hunspell`	`french_hunspell`
Русский	`hunspell_ru_ru`	`russian_hunspell`	`russian_hunspell`

F.24.1. Примеры

Объекты текстового поиска будут созданы после установки модуля словаря. Мы можем проверить созданную конфигурацию:

```
SELECT * FROM ts_debug('english_hunspell', 'abilities');
   alias   |   description   |   token   |         dictionaries          |
 dictionary    |  lexemes
-----------+-----------------+-----------+-------------------------------
+------------------+-----------
 asciiword | Word, all ASCII | abilities | {english_hunspell,english_stem} |
 english_hunspell | {ability}
(1 row)
```

Либо вы можете создать собственную конфигурацию текстового поиска. Например, с созданными словарями и словарём `Snowball` вы можете получить смешанную русско-английскую конфигурацию:

```
CREATE TEXT SEARCH CONFIGURATION russian_en (
  COPY = simple
);

ALTER TEXT SEARCH CONFIGURATION russian_en
  ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
  WITH english_hunspell, english_stem;

ALTER TEXT SEARCH CONFIGURATION russian_en
  ALTER MAPPING FOR word, hword, hword_part
  WITH russian_hunspell, russian_stem;
```

Создавать смешанные словари можно только для языков с различными алфавитами. Если у двух языков похожие алфавиты, Postgres Pro не сможет определить, какой словарь нужно использовать.

Конфигурация текстового поиска, созданная модулем словаря, непосредственно готова к использованию. Например, так можно искать определённые слова в этом тексте:

```
SELECT to_tsvector('english_hunspell', 'The blue whale is the largest animal');
              to_tsvector
-----------------------------------------
 'animal':7 'blue':2 'large':6 'whale':3
(1 row)
```

Поисковый запрос может выглядеть так:

```
SELECT to_tsvector('english_hunspell', 'The blue whale is the largest animal')
  @@ to_tsquery('english_hunspell', 'large & whale');
 ?column?
----------
 t
(1 row)
```

С такими конфигурациями можно искать текст, применяя индексы GIN или GIST. Например, если существует таблица с индексом GIN:

```
CREATE TABLE table1 (t varchar);
INSERT INTO table1 VALUES ('The blue whale is the largest animal');
CREATE INDEX t_idx ON table1 USING GIN (to_tsvector('english_hunspell', "t"));
```

Для неё можно выполнить следующий запрос:

```
SELECT * FROM table1 where to_tsvector('english_hunspell', t)
  @@ to_tsquery('english_hunspell', 'blue & animal');
                  t
--------------------------------------
 The blue whale is the largest animal
(1 row)
```

F.25. hypopg — поддержка гипотетических индексов

F.25.1. Описание

hypopg — это расширение Postgres Pro, добавляющее поддержку *гипотетических индексов*.

Гипотетический или виртуальный индекс — это индекс, который на самом деле не существует, поэтому на его создание не расходуются ресурсы процессора, диска и прочие. Такие индексы помогают без затрат ресурсов выяснить, можно ли повысить производительность проблемных запросов с помощью каких-либо индексов, а также узнать, будет ли Postgres Pro использовать эти индексы или нет.

F.25.2. Установка

Расширение hypopg поставляется вместе с Postgres Pro в виде отдельного пакета `hypopg-ent-16` (подробные инструкции по установке приведены в Главе 16). После установки Postgres Pro создайте расширение hypopg:

```
CREATE EXTENSION hypopg;
```

Расширение hypopg теперь доступно. Это можно проверить с помощью psql:

```
\dx
                     List of installed extensions
  Name    | Version |   Schema   |              Description
---------+---------+------------+-------------------------------------
 hypopg   | 1.4.1   | public     | Hypothetical indexes for Postgres Pro
 plpgsql  | 1.0     | pg_catalog | PL/pgSQL procedural language
(2 rows)
```

Видно, что расширение hypopg установлено.

F.25.3. Функции

Расширение hypopg можно использовать, чтобы проверить, поможет ли какой-либо индекс выполнению одного или нескольких запросов. Поэтому вам должно быть уже известно, какие запросы нужно оптимизировать и какие индексы попробовать.

Гипотетические индексы, которые создаёт hypopg, хранятся не в каталоге, а в собственной памяти вашего соединения. Таким образом, это не раздувает таблицы и не влияет на параллельные соединения.

Кроме того, поскольку гипотетические индексы фактически не существуют, hypopg гарантирует, что они будут использоваться только в обычном операторе `EXPLAIN` (без параметра `ANALYZE`).

Поддерживаются следующие методы доступа:

- `btree`
- `brin`
- `hash`
- `bloom` (требуется наличие установленного расширения bloom)

> **Примечание**
>
> Использование hypopg требует некоторых знаний о команде `EXPLAIN`.

F.25.3.1. Создание гипотетического индекса

`hypopg_create_index()`

Рассмотрим простой пример:

```
CREATE TABLE hypo (id integer, val text) ;
INSERT INTO hypo SELECT i, 'line ' || i FROM generate_series(1, 100000) i ;
VACUUM ANALYZE hypo ;
```

У этой таблицы нет индекса. Предположим, мы хотим проверить, поможет ли индекс простому запросу. Сначала посмотрим, как он себя ведёт:

```
EXPLAIN SELECT val FROM hypo WHERE id = 1;
                       QUERY PLAN
--------------------------------------------------------
 Seq Scan on hypo  (cost=0.00..1791.00 rows=1 width=14)
   Filter: (id = 1)
(2 rows)
```

Используется простое последовательное сканирование, поскольку индекса в таблице нет. Построение индекса B-дерева по столбцу `id` должно помочь этому запросу. Проверим это с помощью hypopg. Функция `hypopg_create_index()` примет любые стандартные операторы `CREATE INDEX` (любые другие операторы, переданные этой функции, будут проигнорированы) и создаст гипотетический индекс для каждого:

```
SELECT * FROM hypopg_create_index('CREATE INDEX ON hypo (id)') ;
 indexrelid |      indexname
------------+----------------------
      18284 | <18284>btree_hypo_id
(1 row)
```

Функция возвращает два столбца:

- идентификатор объекта гипотетического индекса
- сгенерированное имя гипотетического индекса

Можно снова запустить `EXPLAIN`, чтобы увидеть, будет ли Postgres Pro использовать этот индекс:

```
EXPLAIN SELECT val FROM hypo WHERE id = 1;
                                    QUERY PLAN
----------------------------------------------------------------------------------
 Index Scan using <18284>btree_hypo_id on hypo  (cost=0.04..8.06 rows=1 width=10)
   Index Cond: (id = 1)
(2 rows)
```

Да, Postgres Pro будет использовать такой индекс. Теперь нужно убедиться, что гипотетический индекс не будет использоваться для фактического выполнения запроса:

```
EXPLAIN ANALYZE SELECT val FROM hypo WHERE id = 1;
                                            QUERY PLAN
---------------------------------------------------------------------------------------------------
 Seq Scan on hypo  (cost=0.00..1791.00 rows=1 width=10) (actual time=0.046..46.390
 rows=1 loops=1)
   Filter: (id = 1)
   Rows Removed by Filter: 99999
 Planning time: 0.160 ms
 Execution time: 46.460 ms
(5 rows)
```

Это всё, что необходимо для создания гипотетических индексов и проверки, будет ли Postgres Pro использовать такие индексы.

F.25.3.2. Управление гипотетическими индексами

В представлении `hypopg_list_indexes` перечислены все созданные гипотетические индексы.

```
SELECT * FROM hypopg_list_indexes ;
```

```
 indexrelid |      index_name       | schema_name | table_name | am_name
------------+-----------------------+-------------+------------+---------
      18284 | <18284>btree_hypo_id  | public      | hypo       | btree
(1 row)
```

`hypopg()`

Функция `hypopg()` выводит список всех гипотетических индексов, созданных в том же формате, что и `pg_index`.

```
SELECT * FROM hypopg() ;
      indexname       | indexrelid | indrelid | innatts | indisunique | indkey |
 indcollation | indclass | indoption | indexprs | indpred | amid
----------------------+------------+----------+---------+-------------+--------
+--------------+----------+-----------+----------+---------+------
 <18284>btree_hypo_id |      13543 |    18122 |       1 | f           | 1      | 0
          | 1978     | <NULL>    | <NULL>    | <NULL>  |  403
(1 row)
```

`hypopg_get_indexdef(oid)`

Функция `hypopg_get_indexdef(oid)` перечисляет операторы `CREATE INDEX`, которые воссоздадут сохранённый гипотетический индекс.

```
SELECT index_name, hypopg_get_indexdef(indexrelid) FROM hypopg_list_indexes ;
      index_name       |             hypopg_get_indexdef
-----------------------+----------------------------------------------
 <18284>btree_hypo_id  | CREATE INDEX ON public.hypo USING btree (id)
(1 row)
```

`hypopg_relation_size(oid)`

Функция `hypopg_relation_size(oid)` вычисляет размер гипотетического индекса:

```
SELECT index_name, pg_size_pretty(hypopg_relation_size(indexrelid))
  FROM hypopg_list_indexes ;
      index_name       | pg_size_pretty
-----------------------+----------------
 <18284>btree_hypo_id  | 2544 kB
(1 row)
```

`hypopg_drop_index(oid)`

Функция `hypopg_drop_index(oid)` удаляет указанный гипотетический индекс.

`hypopg_reset()`

Функция `hypopg_reset()` удаляет все гипотетические индексы.

F.25.3.3. Гипотетическое скрытие существующих индексов

`hypopg_hide_index(oid)`

Можно гипотетически скрывать как существующие, так и гипотетические индексы. Если вы хотите протестировать эту возможность согласно описанию в документации, сначала вызовите функцию `hypopg_reset()`, чтобы очистить систему от других гипотетических индексов.

В качестве примера рассмотрим два индекса:

```
SELECT hypopg_reset();
CREATE INDEX ON hypo(id);
CREATE INDEX ON hypo(id, val);
```

```
EXPLAIN SELECT * FROM hypo WHERE id = 1;
                                  QUERY PLAN
```

```
---------------------------------------------------------------------------------------
Index Only Scan using hypo_id_val_idx on hypo  (cost=0.29..8.30 rows=1 width=13)
Index Cond: (id = 1)
(2 rows)
```

План запроса теперь использует индекс `hypo_id_val_idx`.

Функция `hypopg_hide_index(oid)` позволяет скрыть индекс в выводе `EXPLAIN`, используя его OID. Возвращает true, если индекс был успешно скрыт, и false в противном случае.

```
SELECT hypopg_hide_index('hypo_id_val_idx'::REGCLASS);
 hypopg_hide_index
-------------------
t
(1 row)

EXPLAIN SELECT * FROM hypo WHERE id = 1;
                            QUERY PLAN
-------------------------------------------------------------------------
Index Scan using hypo_id_idx on hypo  (cost=0.29..8.30 rows=1 width=13)
Index Cond: (id = 1)
(2 rows)
```

В качестве примера предположим, что план запроса в данный момент использует индекс `hypo_id_val_idx`. Для продолжения тестирования используйте функцию `hypopg_hide_index(oid)`, чтобы скрыть другой индекс.

```
SELECT hypopg_hide_index('hypo_id_idx'::REGCLASS);
 hypopg_hide_index
-------------------
t
(1 row)

EXPLAIN SELECT * FROM hypo WHERE id = 1;
                    QUERY PLAN
-------------------------------------------------------
Seq Scan on hypo  (cost=0.00..180.00 rows=1 width=13)
Filter: (id = 1)
(2 rows)
```

`hypopg_unhide_index(oid)`

Функция `hypopg_unhide_index(oid)` восстанавливает ранее скрытый индекс в выводе `EXPLAIN`, используя его OID. Возвращает true, если индекс был успешно восстановлен, и false в противном случае.

```
SELECT hypopg_unhide_index('hypo_id_idx'::regclass);
 hypopg_unhide_index
-------------------
t
(1 row)

EXPLAIN SELECT * FROM hypo WHERE id = 1;
                            QUERY PLAN
-------------------------------------------------------------------------
Index Scan using hypo_id_idx on hypo  (cost=0.29..8.30 rows=1 width=13)
Index Cond: (id = 1)
(2 rows)
```

`hypopg_unhide_all_index(oid)`

Функция `hypopg_unhide_all_index()` восстанавливает все скрытые индексы и возвращает `void`.

`hypopg_hidden_indexes()`

Функция `hypopg_hidden_indexes()` возвращает список OID для всех скрытых индексов.

```
SELECT * FROM hypopg_hidden_indexes();
 indexid
---------
526604
(1 rows)
```

В представлении `hypopg_hidden_indexes` содержится форматированный список всех скрытых индексов.

```
SELECT * FROM hypopg_hidden_indexes;
  indexrelid |      index_name      | schema_name | table_name | am_name | is_hypo
-------------+----------------------+-------------+------------+---------+---------
      526604 | hypo_id_val_idx      | public      | hypo       | btree   | f
(1 rows)
```

Примечание

Гипотетические индексы также можно скрывать.

```
SELECT hypopg_create_index('CREATE INDEX ON hypo(id)');
    hypopg_create_index
------------------------------
(12659,<12659>btree_hypo_id)
(1 row)

EXPLAIN SELECT * FROM hypo WHERE id = 1;
                                    QUERY PLAN
------------------------------------------------------------------------------------
Index Scan using "<12659>btree_hypo_id" on hypo  (cost=0.04..8.05 rows=1 width=13)
Index Cond: (id = 1)
(2 rows)
```

Теперь, когда используется гипотетический индекс, можно попробовать скрыть его, чтобы увидеть изменение:

```
SELECT hypopg_hide_index(12659);
 hypopg_hide_index
-------------------
t
(1 row)

EXPLAIN SELECT * FROM hypo WHERE id = 1;
                            QUERY PLAN
-------------------------------------------------------------------------
Index Scan using hypo_id_idx on hypo  (cost=0.29..8.30 rows=1 width=13)
Index Cond: (id = 1)
(2 rows)

SELECT * FROM hypopg_hidden_indexes;
  indexrelid |      index_name      | schema_name | table_name | am_name | is_hypo
-------------+----------------------+-------------+------------+---------+---------
       12659 | <12659>btree_hypo_id | public      | hypo       | btree   | t
      526604 | hypo_id_val_idx      | public      | hypo       | btree   | f
(2 rows)
```

Примечание

Если гипотетический индекс был скрыт, он будет автоматически раскрыт при удалении функцией `hypopg_drop_index(oid)` или `hypopg_reset()`.

```
SELECT hypopg_drop_index(12659);

SELECT * FROM hypopg_hidden_indexes;
  indexrelid |      index_name      | schema_name | table_name | am_name | is_hypo
-------------+----------------------+-------------+------------+---------+---------
      526604 | hypo_id_val_idx      | public      | hypo       | btree   | f
(2 rows)
```

F.25.4. Параметры GUC

Доступны следующие параметры конфигурации (GUC), которые можно изменять интерактивно:

`hypopg.enabled`

По умолчанию включён (`on`). Используйте этот параметр для глобального включения или отключения hypopg. Если расширение отключено, гипотетические индексы не будут использоваться, но уже определённые гипотетические индексы не будут удалены.

`hypopg.use_real_oids`

По умолчанию выключен (`off`). При таком значении hypopg не будет использовать «реальные» идентификаторы объектов, а вместо этого позаимствует их из диапазона ~ 14000 / 16384 (соответственно наименьший неиспользуемый OID меньше `FirstNormalObjectId` и `FirstNormalObjectId`), которые зарезервированы Postgres Pro для использования в будущих выпусках. Это не вызывает никаких проблем, так как свободный диапазон динамически вычисляется при первом использовании hypopg подключением и расширение может работать на резервных серверах. Однако одновременно может существовать не более 2500 гипотетических индексов, и после превышения максимального количества объектов создание нового гипотетического индекса станет очень медленным, пока не будет вызвана функция `hypopg_reset()`.

Если эти недостатки вызывают проблемы, можно включить этот параметр. Тогда hypopg будет запрашивать реальный идентификатор объекта, для чего потребуется получать больше блокировок, и не будет работать на резервных серверах, но позволит использовать полный диапазон идентификаторов объектов.

Обратите внимание, что переключение этого параметра не требует сброса записей.

F.26. intagg — агрегатор и нумератор целых чисел

Модуль `intagg` предоставляет агрегатор и нумератор целых чисел. На данный момент имеются встроенные функции, предлагающие более широкие возможности, поэтому `intagg` считается устаревшим. Однако этот модуль продолжает существовать для обратной совместимости, теперь как набор обёрток встроенных функций.

F.26.1. Функции

Агрегатор реализуется функцией `int_array_aggregate(integer)`, которая выдаёт массив целых чисел, содержащий в точности те числа, что переданы ей. Это обёртка встроенной функции `array_agg`, которая делает то же самое для массива любого типа.

Нумератор реализуется функцией `int_array_enum(integer[])`, которая возвращает набор целых (`setof integer`). По сути его действие обратно действие агрегатора: получив массив целых, он разворачивает его в набор строк. Это оболочка функции `unnest`, которая делает то же самое для массива любого типа.

F.26.2. Примеры использования

Во многих СУБД есть понятие таблицы соотношений «многие ко многим». Такая таблица обычно находится между двумя индексированными таблицами, например:

```
CREATE TABLE left_table  (id INT PRIMARY KEY, ...);
CREATE TABLE right_table (id INT PRIMARY KEY, ...);
CREATE TABLE many_to_many(id_left  INT REFERENCES left_table,
                          id_right INT REFERENCES right_table);
```

Как правило, она используется так:

```
SELECT right_table.*
FROM right_table JOIN many_to_many ON (right_table.id = many_to_many.id_right)
WHERE many_to_many.id_left = элемент;
```

Этот запрос вернёт все элементы из таблицы справа для записи в таблице слева. Это очень распространённая конструкция в SQL.

Однако этот подход может вызывать затруднения с очень большим количеством записей в таблице `many_to_many`. Часто такое соединение влечёт сканирование индекса и выборку каждой записи в таблице справа для конкретного элемента слева. Если у вас динамическая система, с этим ничего не поделать. Но если какое-то множество данных довольно статическое, вы можете создать сводную таблицу, применив агрегатор.

```
CREATE TABLE summary AS
  SELECT id_left, int_array_aggregate(id_right) AS rights
  FROM many_to_many
  GROUP BY id_left;
```

Эта команда создаст таблицу, содержащую одну строку для каждого элемента слева с массивом элементов справа. Она малополезна, пока не найден подходящий способ использования этого массива; именно для этого и нужен нумератор массива. Вы можете выполнить:

```
SELECT id_left, int_array_enum(rights) FROM summary WHERE id_left = элемент;
```

Приведённый выше запрос с вызовом `int_array_enum` выдаёт те же результаты, что и

```
SELECT id_left, id_right FROM many_to_many WHERE id_left = элемент;
```

Отличие состоит в том, что запрос к сводной таблице должен выдать только одну строку таблицы, тогда как непосредственный запрос к `many_to_many` потребует сканирования индекса и выборки строки для каждой записи.

На тестовом компьютере команда `EXPLAIN` показала, что стоимость запроса снизилась с 8488 до 329. Исходный запрос выполнял соединение с таблицей `many_to_many` и был заменён на:

```
SELECT id_right, count(id_right) FROM
  ( SELECT id_left, int_array_enum(rights) AS id_right
    FROM summary
    JOIN (SELECT id FROM left_table
          WHERE id = элемент) AS lefts
    ON (summary.id_left = lefts.id)
  ) AS list
  GROUP BY id_right
  ORDER BY count DESC;
```

F.27. intarray — работа с массивами целых чисел

Модуль `intarray` предоставляет ряд полезных функций и операторов для работы с массивами целых чисел без NULL. Также он поддерживает поиск по индексу для некоторых из этих операторов.

Все эти операции выдают ошибку, если в передаваемом массиве оказываются значения NULL.

Многие из этих операций имеют смысл только с одномерными массивами. Хотя им можно передать входной массив и большей размерности, значения будут считываться из него как из линейного массива в порядке хранения.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.27.1. Функции и операторы `intarray`

Реализованные в модуле `intarray` функции перечислены в Таблице F.13, а операторы — в Таблице F.14.

Таблица F.13. Функции `intarray`

Функция **Описание** **Примеры**

`icount` (`integer[]`) → `integer`
Выдаёт число элементов в массиве.
`icount('{1,2,3}'::integer[])` → `3`

`sort` (`integer[]`, *`dir`* `text`) → `integer[]`
Сортирует массив в порядке возрастания или убывания. Направление определяется параметром *`dir`*, значением которого должно быть `asc` (по возрастанию) или `desc` (по убыванию).
`sort('{1,3,2}'::integer[], 'desc')` → `{3,2,1}`

`sort` (`integer[]`) → `integer[]`
`sort_asc` (`integer[]`) → `integer[]`
Сортирует в порядке возрастания.
`sort(array[11,77,44])` → `{11,44,77}`

`sort_desc` (`integer[]`) → `integer[]`
Сортирует в порядке убывания.
`sort_desc(array[11,77,44])` → `{77,44,11}`

`uniq` (`integer[]`) → `integer[]`
Удаляет соседние дубликаты. Часто используется вместе с `sort` для удаления всех дубликатов.
`uniq('{1,2,2,3,1,1}'::integer[])` → `{1,2,3,1}`
`uniq(sort('{1,2,3,2,1}'::integer[]))` → `{1,2,3}`

`idx` (`integer[]`, *`item`* `integer`) → `integer`
Выдаёт индекс первого элемента, равного *`item`*, или 0, если такого элемента нет.
`idx(array[11,22,33,22,11], 22)` → `2`

`subarray` (`integer[]`, *`start`* `integer`, *`len`* `integer`) → `integer[]`
Извлекает часть массива, которая начинается с позиции *`start`* и содержит *`len`* элементов.
`subarray('{1,2,3,2,1}'::integer[], 2, 3)` → `{2,3,2}`

`subarray` (`integer[]`, *`start`* `integer`) → `integer[]`

Функция **Описание** **Примеры**
Извлекает часть массива, которая начинается с позиции *start*. `subarray('{1,2,3,2,1}'::integer[], 2)` → `{2,3,2,1}`

`intset` (`integer`) → `integer[]`
Создаёт массив с одним элементом.
`intset(42)` → `{42}`

Таблица F.14. Операторы `intarray`

Оператор **Описание**
`integer[] && integer[]` → `boolean` Массивы пересекаются (у них есть минимум один общий элемент)?
`integer[] @> integer[]` → `boolean` Левый массив содержит правый?
`integer[] <@ integer[]` → `boolean` Левый массив содержится в правом?
`# integer[]` → `integer` Выдаёт число элементов в массиве.
`integer[] # integer` → `integer` Выдаёт индекс первого элемента, равного правому аргументу, или 0, если такого элемента нет. (Аналог функции `idx`.)
`integer[] + integer` → `integer[]` Добавляет элемент в конец массива.
`integer[] + integer[]` → `integer[]` Соединяет два массива.
`integer[] - integer` → `integer[]` Удаляет из массива элементы, равные правому аргументу.
`integer[] - integer[]` → `integer[]` Удаляет из левого массива элементы правого массива.
`integer[] | integer` → `integer[]` Вычисляет объединение аргументов.
`integer[] | integer[]` → `integer[]` Вычисляет объединение аргументов.
`integer[] & integer[]` → `integer[]` Вычисляет пересечение аргументов.
`integer[] @@ query_int` → `boolean` Массив удовлетворяет запросу? (см. ниже)
`query_int ~~ integer[]` → `boolean` Массив удовлетворяет запросу? (коммутирующий оператор к `@@`)

Операторы `&&`, `@>` и `<@` равнозначны встроенным операторам Postgres Pro с теми же именами, за исключением того, что они работают только с целочисленными массивами, не содержащими NULL, тогда как встроенные операторы работают с массивами любых типов. Благодаря этому ограничению, в большинстве случаев они работают быстрее, чем встроенные операторы.

Операторы `@@` и `~~` проверяют, удовлетворяет ли массив *запросу*, представляемому в виде значения специализированного типа данных `query_int`. *Запрос* содержит целочисленные значения, сравниваемые с элементами массива, возможно с использованием операторов `&` (AND), `|` (OR) и `!` (NOT). При необходимости могут использоваться скобки. Например, запросу `1&(2|3)` удовлетворяют запросы, которые содержат 1 и также содержат 2 или 3.

F.27.2. Поддержка индексов

Модуль `intarray` поддерживает индексы для операторов `&&`, `@>`, и `@@`, а также обычную проверку равенства массивов.

Модуль предоставляет два параметризованных класса операторов GiST: `gist__int_ops` (используется по умолчанию), подходящий для маленьких и средних по размеру наборов данных, и `gist__intbig_ops`, применяющий сигнатуру большего размера и подходящий для индексации больших наборов данных (то есть столбцов, содержащих много различных значений массива). В этой реализации используется структура данных RD-дерева со встроенным сжатием с потерями.

Класс `gist__int_ops` аппроксимирует набор целых чисел в виде массива диапазонов. В его необязательном целочисленном параметре `numranges` можно задать максимальное число диапазонов в одном ключе индекса. Параметр может принимать значения от 1 до 253, по умолчанию он равен 100. При увеличении числа массивов, составляющих ключ индекса GiST, поиск работает точнее (сканируется меньшая область в индексе и меньше страниц кучи), но сам индекс становится больше.

Класс `gist__intbig_ops` аппроксимирует набор целых чисел в виде сигнатуры битовой карты. В его необязательном целочисленном параметре `siglen` можно задать размер сигнатуры в байтах. Параметр может принимать значения от 1 до 2024, по умолчанию он равен 16. При увеличении размера сигнатуры поиск работает точнее (сканируется меньшая область в индексе и меньше страниц кучи), но сам индекс становится больше.

Есть также нестандартный класс операторов GIN, `gin__int_ops`, поддерживающий эти операторы наряду с `<@`.

Выбор между индексами GiST и GIN зависит от относительных характеристик производительности GiST и GIN, которые здесь не рассматриваются.

F.27.3. Пример

```
-- сообщение может относиться к одной или нескольким «секциям»
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- создать специализированный индекс с длиной сигнатуры 32 байта
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__intbig_ops
 (siglen = 32));

-- вывести сообщения из секций 1 или 2 — оператор пересечения
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- вывести сообщения из секций 1 и 2 — оператор включения
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- тот же результат, но с оператором запроса
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.27.4. Авторы

Разработку осуществили Фёдор Сигаев (`<teodor@sigaev.ru>`) и Олег Бартунов (`<oleg@sai.msu.su>`). Дополнительные сведения можно найти на странице *http://www.sai.msu.su/~megera/postgres/gist/*. Андрей Октябрьский проделал отличную работу, добавив новые функции и операторы.

F.28. isn — типы данных для международных стандартов нумерации (ISBN, EAN, UPC и т. д.)

Модуль `isn` предоставляет типы данных для следующих международных стандартов нумерации товаров: EAN13, UPC, ISBN (книги), ISMN (музыка) и ISSN (серийные номера). Номера проверяются на входе согласно жёстко заданному списку префиксов: этот список префиксов также применяется для группирования цифр при выводе. Так как время от времени появляются новые префиксы, этот список префиксов может устаревать. Ожидается, что будущая версия этого модуля будет получать список префиксов из одной или нескольких таблиц, благодаря чему его при необходимости смогут легко обновлять пользователи; однако в настоящее время этот список можно изменить, только модифицировав исходный код и перекомпилировав его. Также есть вероятность, что в будущем этот модуль лишится функций проверки префиксов и группирования цифр.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.28.1. Типы данных

В Таблице F.15 перечислены типы данных, реализованные модулем `isn`.

Таблица F.15. Типы данных isn

Тип данных	Описание
`EAN13`	European Article Number (Европейский номер товара), всегда выводится в формате EAN13
`ISBN13`	International Standard Book Number (Международный стандартный книжный номер), выводимый в новом формате EAN13
`ISMN13`	International Standard Music Number (Международный стандартный музыкальный номер), выводимый в новом формате EAN13
`ISSN13`	International Standard Serial Number (Международный стандартный серийный номер), выводимый в новом формате EAN13
`ISBN`	Международный стандартный книжный номер, выводимый в старом коротком формате
`ISMN`	Международный стандартный музыкальный номер, выводимый в старом коротком формате
`ISSN`	Международный стандартный серийный номер, выводимый в старом коротком формате
`UPC`	Universal Product Code (Универсальный код товара)

Замечания:

1. Номера ISBN13, ISMN13, ISSN13 являются номерами EAN13.
2. Не все номера EAN13 (только их подмножество) представляют ISBN13, ISMN13 или ISSN13.
3. Некоторые номера ISBN13 можно вывести в формате ISBN.
4. Некоторые номера ISMN13 можно вывести в формате ISMN.
5. Некоторые номера ISSN13 можно вывести в формате ISSN.
6. Номера UPC являются подмножеством номеров EAN13 (по сути они являются номерами EAN13 без первой цифры `0`).
7. Любые номера UPC, ISBN, ISMN и ISSN можно представить как номера EAN13.

Внутри все эти типы представляются одинаково (64-битными целыми числами) и все они взаимозаменяемы. Различные типы введены для управления форматированием при выводе и для строгой проверки правильности ввода, что и определяет конкретный тип номера.

Типы `ISBN`, `ISMN` и `ISSN` выводят короткую версию числа (ISxN 10), когда это возможно, либо выбирают формат ISxN 13 для чисел, не умещающихся в короткую версию. Типы `EAN13`, `ISBN13`, `ISMN13` и `ISSN13` всегда выводят длинную версию ISxN (EAN13).

F.28.2. Приведения

Модуль `isn` предоставляет следующие пары приведений типов:

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

При приведении `EAN13` к другому типу выполняется проверка времени выполнения, соответствует ли исходное значение целевому типу, и если это не так, выдаётся ошибка. При других приведениях значения просто перемечаются, так что они успешно выполняются всегда.

F.28.3. Функции и операторы

Модуль `isn` предоставляет стандартные операторы сравнения плюс поддержку индексов по хешу и B-деревьев для этих типов данных. Кроме того, он реализует ряд специализированных функций; они перечислены в Таблице F.16. В этой таблице под `isn` понимается один из типов данных модуля.

Таблица F.16. Функции `isn`

Функция Описание
`isn_weak ( boolean ) → boolean` Устанавливает ослабленный режим проверки ввода и возвращает новое состояние.
`isn_weak () → boolean` Выдаёт текущее состояние ослабленного режима.
`make_valid ( isn ) → isn` Делает неверный номер верным (сбрасывает флаг ошибки).
`is_valid ( isn ) → boolean` Проверяет присутствие флага ошибки.

Ослабленный режим позволяет вставлять в таблицу неверные значения. Под неверным значением понимается номер, в котором не сходится проверочная цифра, а не номер отсутствует вовсе.

Зачем может понадобиться ослабленный режим? Ну например, у вас может быть большой набор номеров ISBN, и очень многие из них по каким-то странным причинам содержат неверную контрольную цифру (возможно, номера были отсканированы с печатного листа и были распознаны неправильно, или они вводились вручную, кто знает...). В любом случае суть в том, что вы хотите с этим разобраться, но вам нужно загрузить все эти номера в базу данных, а затем, возможно, использовать дополнительное средство поиска неверных номеров в базе данных, чтобы проверить и исправить данные; так что, например, вам нужно будет выбрать все неверные номера в таблице.

Если попытаться вставить в таблицу неверный номер в ослабленном режиме проверки, номер будет вставлен с исправленной проверочной цифрой, но выводиться будет с восклицательным знаком (`!`) в конце, например `0-11-000322-5!`. Этот маркер ошибки можно проверить с помощью функции `is_valid` и очистить функцией `make_valid`.

Вы также можете принудительно вставить числа даже не в ослабленном режиме, добавив символ `!` в конце номера.

Есть ещё одна особенность — во вводимом номере вместо проверочной цифры можно указать `?` и нужная проверочная цифра будет вставлена автоматически.

F.28.4. Примеры

```
-- Использование типов напрямую:
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

-- Приведение типов:
-- заметьте, что номер ean13 можно привести к другому типу, только когда
-- этот номер будет допустимым для целевого типа;
-- таким образом, это НЕ будет работать: select isbn(ean13('0220356483481'));
-- а это будет:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

-- Создание таблицы с одним столбцом, который будет содержать номера ISBN:
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

-- Автоматическое вычисление проверочных цифр (обратите внимание на '?'):
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

-- Использование ослабленного режима:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;
```

F.28.5. Библиография

Информация, на основе которой реализован этот модуль, была собрана с нескольких сайтов, включая:

- *https://www.isbn-international.org/*
- *https://www.issn.org/*
- *https://www.ismn-international.org/*

- *https://www.wikipedia.org/*

Префиксы, используемые для группирования цифр, были также взяты с:

- *https://www.gs1.org/standards/id-keys*
- *https://en.wikipedia.org/wiki/List_of_ISBN_identifier_groups*
- *https://www.isbn-international.org/content/isbn-users-manual/29*
- *https://en.wikipedia.org/wiki/International_Standard_Music_Number*
- *https://www.ismn-international.org/ranges/tools*

Алгоритмы реализованы со всей тщательностью и скрупулёзно сверены с алгоритмами, предлагаемыми в официальных руководствах ISBN, ISMN, ISSN.

F.28.6. Автор

Герман Мендез Браво (Kronuz), 2004–2006

Этот модуль написан под влиянием кода `isbn_issn` Гаретта А. Уоллмена.

F.29. jsquery — язык запросов к типу данных `jsonb`

JsQuery — это язык запросов к данным `jsonb`. Его основное предназначение — предоставить дополнительную функциональность для `jsonb`, например, простой и эффективный способ поиска во вложенных объектах и массивах, а также дополнительные операторы сравнения с поддержкой индексов.

JsQuery реализован посредством типа данных `jsquery` (подобно `tsquery`) и оператора `@@` для применения запросов к `jsonb`.

F.29.1. Установка

Программный комплект Postgres Pro Standard включает jsquery в качестве дополнительного модуля (contrib). Установив необходимые компоненты Postgres Pro Standard, создайте расширение jsquery следующим образом:

```
CREATE EXTENSION jsquery;
```

F.29.2. Язык запросов JSON

Расширение JsQuery содержит тип `jsquery`, представляющий весь запрос JSON в виде одного значения (как и `tsquery` для полнотекстового поиска). В этом запросе задаётся выражение, обращающееся к документам JSON.

Простое выражение записывается в виде *`путь бинарный_оператор значение`* или *`путь унарный_оператор`*. См. следующие примеры.

- `x = "abc"` — значение ключа "x" равно "abc";
- `$ @> [4, 5, "zzz"]` — документ JSON представляет собой массив, содержащий значения 4, 5 и "zzz";
- `"abc xyz" >= 10` — значение ключа "abc xyz" больше или равно 10;
- `volume IS NUMERIC` — ключ «volume» имеет числовой тип.
- `$ = true` — весь документ JSON представляет собой просто значение true.
- `similar_ids.@# > 5` - `similar_ids` является массивом или объектом с числом элементов больше 5;
- `similar_product_ids.# = "0684824396"` — массив `similar_product_ids` содержит строку "0684824396".
- `*.color = "red"` — внутри документа существует объект, ключ "color" в котором имеет значение "red".
- `foo = *` — в объекте существует ключ «foo».

Путь выбирает набор значений JSON для проверки, используя заданные операторы. В простейшем случае путь — это просто имя ключа. В общем случае путь задают имена ключей и подстановочные знаки, соединённые точками. Подстановочные знаки в пути могут быть следующими:

- `#` — любой элемент в массиве;
- `#N` — N-ый индекс в массиве;
- `%` — любой ключ объекта;
- `*` — любой набор элементов массивов и ключей объектов;
- `@#` — длина массива или объекта, может задаваться только в последнем компоненте пути;
- `$` — весь документ JSON в виде одного значения, может быть только единственным компонентом пути.

Результат выражения — true, когда оператор выдаёт true как минимум для одного значения, выбранного по заданному пути.

Имена ключей могут задаваться в двойных кавычках и без них. Имена ключей без кавычек не могут содержать пробелы, начинаться с цифры или совпадать с ключевым словом jsquery.

Поддерживаются следующие бинарные операторы:

- Оператор равенства: `=`;
- Операторы численного сравнения: `>`, `>=`, `<`, `<=`;
- Оператор `IN` для поиска в списке скалярных значений;
- Операторы сравнения массивов: `&&` (пересекается с), `@>` (содержит), `<@` (содержится в).
- Оператор фильтра: `~~`. Принимая данные `jsonb` в качестве левого операнда и выражение `jsquery` в качестве правого, этот оператор ищет в данных `jsonb` элементы, удовлетворяющие условию, заданному в выражении `jsquery`, и возвращает массив таких элементов, если они находятся.

Поддерживаются следующие унарные операторы:

- Оператор проверки существования: `= *`;
- Операторы проверки типа: `IS ARRAY`, `IS NUMERIC`, `IS OBJECT`, `IS STRING` и `IS BOOLEAN`.

Выражения могут быть сложными. Сложное выражение образуется набором выражений, соединённых логическими операторами (`AND`, `OR` и `NOT`) и сгруппированных скобками.

Примеры сложных выражений приведены ниже.

- `a = 1 AND (b = 2 OR c = 3) AND NOT d = 1`
- `x.% = true OR x.# = true`

Путь может включать в себя префиксные выражения (подвыражения). В этом случае путь выбирает значения JSON для проверки по заданным подвыражениям. Результаты проверки агрегируются так же, как и в простых выражениях.

- `#(a = 1 AND b = 2)` — массив содержит элемент, в котором значение ключа `a` равно 2 и значение ключа `b` равно 2
- `%($ >= 10 AND $ <= 20)` — объект содержит ключ со значением от 10 до 20

Путь также может содержать следующие подстановочные знаки, обозначающие «каждый»:

- `#:` — каждый индекс в массиве;
- `%:` — каждый ключ объекта;
- `*:` — каждый элемент массива или ключ объекта.

Рассмотрим следующий пример.

```
%.#:($ >= 0 AND $ <= 1)
```

Это условие можно прочитать как: существует минимум один ключ, для которого значением является массив чисел от 0 до 1.

Мы можем переписать этот пример в следующей форме с дополнительными скобками.

```
%(#:($ >= 0 AND $ <= 1))
```

Первый подстановочный знак `%` проверяет, что выражение в скобках истинно как минимум для одного значения в объекте. Второй подстановочный знак `#:` проверяет, что значение — массив, все элементы которого удовлетворяют условию в скобках.

Мы можем переписать этот пример без подстановочного знака `#:` следующим образом.

```
%(NOT #(NOT ($ >= 0 AND $ <= 1)) AND $ IS ARRAY)
```

В этом примере мы преобразуем утверждение, что каждый элемент массива удовлетворяет некоторому условию, в утверждение, что ни один элемент не удовлетворяет тому же условию.

Ниже приведены примеры использования путей.

- `numbers.#: IS NUMERIC` — каждый элемент массива numbers является числом.
- `*:($ IS OBJECT OR $ IS BOOLEAN)` — документ JSON представляет собой структуру вложенных объектов, на уровне листьев содержащую логические значения.

- `#:.%:($ >= 0 AND $ <= 1)` — каждый элемент массива представляет собой объект, содержащий числовые значения от 0 до 1.
- `documents.#:.% = *` — ключу documents соответствует массив из объектов, содержащих минимум один ключ.
- `%.#: ($ IS STRING)` — объект JSON содержит минимум один массив со строками.
- `#.% = true` — как минимум один элемент массива является объектом, содержащим минимум одно значение true.

Использование операторов пути и скобок требует дополнительных разъяснений. Когда операторы пути используются неоднократно, они могут ссылаться на различные значения, тогда как, используя скобки и оператор `$`, вы будете многократно обращаться к одному значению. Взгляните на следующие примеры.

- `# < 10 AND # > 20` — существует элемент меньше 10 и существует другой элемент больше 20.
- `#($ < 10 AND $ > 20)` — существует элемент, который одновременно меньше 10 и больше 20 (что невозможно).
- `#($ >= 10 AND $ <= 20)` — существует элемент между 10 и 20.
- `# >= 10 AND # <= 20` — существует элемент, больший или равный 10, и другой элемент, меньший или равный 20. Запрос может быть удовлетворён массивом без элементов между 10 и 20, например [0,30].

Те же правила применяются при поиске внутри объектов и ветвей.

Операторы проверки типа и подстановочные знаки «каждый» полезны для проверки правильности схемы документа. Оператор сопоставления JsQuery `@@` является постоянным (IMMUTABLE) и может применяться в ограничениях CHECK. Взгляните на следующий пример.

```
CREATE TABLE js (
    id serial,
    data jsonb,
    CHECK (data @@ '
        name IS STRING AND
        similar_ids.#: IS NUMERIC AND
        points.#:(x IS NUMERIC AND y IS NUMERIC)'::jsquery));
```

В этом примере ограничение-проверка контролирует данные в колонке data типа jsonb: значением ключа name должна быть строка, значением ключа similar_ids — массив чисел, значением ключа points — массив объектов, содержащих числовые значения для ключей x и y.

Другие примеры можно найти в нашей *презентации для pgconf.eu*.

F.29.3. Индексы GIN

Расширение JsQuery содержит два класса операторов для GIN, которые реализуют различные варианты оптимизации запросов.

- jsonb_path_value_ops
- jsonb_value_path_ops

В этих классах GIN документы jsonb раскладываются на элементы. Каждый элемент сопоставляется с определённым значением и своим путём. Различаются эти классы операторов тем, как в них представлены элементы, как они сравниваются и используются для оптимизации поиска.

Например, документ jsonb `{"a": [{"b": "xyz", "c": true}, 10], "d": {"e": [7, false]}}` будет разложен на следующие элементы:

- `"a".#."b"."xyz"`
- `"a".#."c".true`
- `"a".#.10`
- `"d"."e".#.7`
- `"d"."e".#.false`

Так как JsQuery не поддерживает поиск в определённом элементе массива, мы рассматриваем все элементы массива как одинаковые. Таким образом, все элементы массива помечаются одинаковым знаком `#` в пути.

Основной проблемой такого представления элементов является его размер. Так, в показанном примере ключ «a» присутствует три раза. В большом документе со множеством ветвей и длинными ключами размер такого наивного представления элементов выходит за рамки разумного. Эти два класса операторов решают эту проблему, но с некоторыми различиями.

F.29.3.1. jsonb_path_value_ops

Класс jsonb_path_value_ops представляет элемент в виде пары хеша пути и значения. Это иллюстрирует следующий псевдокод.

```
(hash(элемент_пути_1.элемент_пути_2. ... .элемент_пути_n); значение)
```

При сравнении записей хеш пути является старшей частью записи, а значение — младшей. Это определяет характеристики данного класса операторов. Так как путь хешируется и оказывается в верхней части записи, нам нужно знать полный путь к значению, чтобы использовать его для поиска. Однако если путь известен, мы можем применять и точный поиск, и поиск значений в интервале.

F.29.3.2. jsonb_value_path_ops

Класс jsonb_value_path_ops представляет элемент в виде пары значения и фильтра Блума для пути.

```
(значение; bloom(элемент_пути_1) | bloom(элемент_пути_2) | ... | bloom(элемент_пути_n))
```

При сравнении записей значение является старшей частью записи, а фильтр Блума для пути — младшей. Это определяет характеристики данного класса операторов. Так как значение находится в старшей части, мы можем выполнять очень эффективно только поиск точного значения. Поиск значений в интервале возможен, но для этого придётся отфильтровать все другие пути, в которых будут найдены подходящие значения. Фильтр Блума по элементам пути позволяет использовать индекс для условий, содержащих в путях `%` и `*`.

F.29.3.3. Оптимизация запросов

Классы операторов JsQuery выполняют сложную оптимизацию запросов. Для разработчика или администратора очень ценно иметь возможность видеть результаты такой оптимизации. Но к сожалению, классы операторов не могут добавлять что-либо своё в вывод EXPLAIN. Именно поэтому JsQuery предоставляет следующие функции, позволяющие увидеть, как конкретный класс операторов оптимизирует заданный запрос.

- gin_debug_query_path_value(jsquery) — для jsonb_path_value_ops
- gin_debug_query_value_path(jsquery) — для jsonb_value_path_ops

Эти функции выдают текстовое представление дерева запросов, листьями в котором являются записи, искомые в индексе GIN. Следующие примеры показывают различия в результатах оптимизации запросов для разных классов операторов.

```
# SELECT gin_debug_query_path_value('x = 1 AND (*.y = 1 OR y = 2)');
 gin_debug_query_path_value
----------------------------
 x = 1 , entry 0           +

# SELECT gin_debug_query_value_path('x = 1 AND (*.y = 1 OR y = 2)');
 gin_debug_query_value_path
----------------------------
 AND                       +
   x = 1 , entry 0         +
   OR                      +
     *.y = 1 , entry 1     +
```

```
          y = 2 , entry 2       +
```

К сожалению, для jsonb ещё нет статистики. Вот почему оптимизатор JsQuery должен принимать императивные решения, выбирая условия, обрабатываемые с индексом. При этом он руководствуется предположением, что некоторые типы условий являются менее избирательными, чем другие. Таким образом, оптимизатор разделяет условия на следующие категории избирательности (перечисленные по убыванию избирательности).

1. Равняется (x = c)
2. В интервале (c1 < x < c2)
3. Не равняется (x > c)
4. Является (x is type)
5. Любое (x = *)

Оптимизатор избегает использования индекса для менее избирательных условий, когда это возможно. Например, в запросе `x = 1 AND y > 0` условие `x = 1` считается более избирательным, чем `y > 0`. Поэтому для вычисления `y > 0` индекс не используется.

```
# SELECT gin_debug_query_path_value('x = 1 AND y > 0');
 gin_debug_query_path_value
----------------------------
 x = 1 , entry 0           +
```

Решения, принимаемые оптимизатором без учёта статистики, могут быть неточными. Поэтому в JsQuery поддерживаются подсказки. Комментарии `/*-- index */` и `/*-- noindex */`, помещённые в условия, указывают оптимизатору соответственно использовать или не использовать индекс.

```
SELECT gin_debug_query_path_value('x = 1 AND y /*-- index */ > 0');
 gin_debug_query_path_value
----------------------------
 AND                       +
   x = 1 , entry 0         +
   y > 0 , entry 1         +

SELECT gin_debug_query_path_value('x /*-- noindex */ = 1 AND y > 0');
 gin_debug_query_path_value
 ----------------------------
  y > 0 , entry 0           +
```

F.29.4. Авторы

- Фёдор Сигаев `<teodor@sigaev.ru>`, Postgres Professional, Москва, Россия
- Александр Коротков `<aekorotkov@gmail.com>`, Postgres Professional, Москва, Россия
- Олег Бартунов `<oleg@sai.msu.su>`, Postgres Professional, Москва, Россия

F.29.5. Благодарности

Мы выражаем признательность спонсору разработки, компании *Wargaming.net*.

F.30. lo — управление большими объектами

Модуль `lo` поддерживает управление большими объектами (БО или LO, Large Objects, иногда BLOB, Binary Large OBjects). Он реализует тип данных `lo` и триггер `lo_manage`.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.30.1. Обоснование

Одна из проблем драйвера JDBC (она распространяется и на драйвер ODBC) в том, что спецификация типа предполагает, что ссылки на BLOB хранятся в таблице, и если запись меняется, связанный BLOB удаляется из базы.

Но с Postgres Pro этого не происходит. Большие объекты обрабатываются как самостоятельные объекты; запись в таблице может ссылаться на большой объект по OID, но при этом на один и тот же объект могут ссылаться несколько записей таблицы, так что система не удаляет большой объект, только потому что вы меняете или удаляете такую запись.

Это не проблема для приложений, ориентированных на Postgres Pro, но стандартный код, использующий JDBC или ODBC, не будет удалять эти объекты, в результате чего они окажутся потерянными — объектами, которые никак не задействованы, а просто занимают место на диске.

Модуль `lo` позволяет решить эту проблему, добавляя триггер к таблицам, которые содержат столбцы, ссылающиеся на БО. Этот триггер по сути просто вызывает `lo_unlink`, когда вы удаляете или изменяете значение, ссылающееся на большой объект. Данный триггер предполагает, что на любой большой объект, на который ссылается контролируемый им столбец, указывает только одна ссылка!

Этот модуль также предоставляет тип данных `lo`, который просто является *доменом* на базе `oid`. Он может быть полезен для выделения столбцов, содержащих ссылки на большие объекты, среди столбцов, содержащих другие OID. Для использования триггера применять тип `lo` необязательно, но этот тип может быть полезен для отслеживания столбцов в вашей базе, представляющих большие объекты, с которыми работает триггер. Кроме того, поступали сообщения, что драйвер ODBC не работает корректно, если для столбцов BLOB используется не тип `lo`.

F.30.2. Как его использовать

Пример его использования:

```
CREATE TABLE image (title text, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE FUNCTION lo_manage(raster);
```

Для каждого столбца, который будет содержать уникальные ссылки на большие объекты, создайте триггер `BEFORE UPDATE OR DELETE` и передайте имя столбца в качестве единственного аргумента триггера. Вы также можете сделать, чтобы триггер срабатывал только при изменениях в столбце, указав `BEFORE UPDATE OF` *`имя_столбца`*. Если вам нужно иметь в одной таблице несколько столбцов `lo`, создайте отдельный триггер для каждого (при этом обязательно нужно дать всем триггерам в одной таблице разные имена).

F.30.3. Ограничения

- При удалении таблицы, однако, всё равно будут потеряны относящиеся к ней объекты, так как триггер не будет выполняться. Этого можно избежать, выполнив перед `DROP TABLE` команду `DELETE FROM` *`таблица`*.

  То же касается и команды `TRUNCATE`.

Если у вас уже есть, или вы подозреваете, что есть потерянные большие объекты, обратите внимание на модуль vacuumlo, который может помочь вычистить их. Имеет смысл периодически запускать vacuumlo в качестве меры, дополняющей действие триггера `lo_manage`.

- Некоторые клиентские программы могут создавать собственные таблицы, но не создавать для них соответствующие триггеры. Кроме того, и пользователи могут не создавать такие триггеры (забывая о них, либо не зная, как это сделать).

F.30.4. Автор

Питер Маунт `<peter@retep.org.uk>`

F.31. ltree — тип данных для представления меток данных в иерархической древовидной структуре

Этот модуль реализует тип данных `ltree` для представления меток данных в иерархической древовидной структуре. Он также предоставляет расширенные средства для поиска в таких деревьях.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.31.1. Определения

Метка — это последовательность буквенно-цифровых символов, знаков подчёркивания и дефисов. Допустимые диапазоны буквенно-цифровых символов зависят от локали базы данных. Например, в локали C допускаются символы `A-Za-z0-9_-`. Метки должны занимать меньше 1000 символов.

Примеры: `42`, `Personal_Services`

Путь метки — это последовательность из нуля или нескольких разделённых точками меток (например, `L1.L2.L3`), представляющая путь от корня иерархического дерева к конкретному узлу. Путь не может содержать больше 65535 меток.

Пример: `Top.Countries.Europe.Russia`

Модуль `ltree` предоставляет несколько типов данных:

- `ltree` хранит путь метки.
- `lquery` представляет напоминающий регулярные выражения запрос для поиска нужных значений `ltree`. В нём простое слово выбирает соответствующую метку в заданном пути, а звёздочка (`*`) — ноль или более любых меток. Отдельные компоненты можно соединить точками и получить запрос, которому будет соответствовать весь путь с указанными метками. Например:

`foo`	*Выбирает путь ровно с одной меткой foo*
`*.foo.*`	*Выбирает путь, содержащий метку foo*
`*.foo`	*Выбирает путь с последней меткой foo*

  И для звёздочки, и для простых слов можно добавить количественное значение, определяющее число меток, которые будут соответствовать этому компоненту:

`*{n}`	*Выбирает ровно n меток*
`*{n,}`	*Выбирает как минимум n меток*
`*{n,m}`	*Выбирает не меньше n, но и не более m меток*
`*{,m}`	*Выбирает не больше m меток — равнозначно `*{0,m}`*
`foo{n,m}`	*Выбирает как минимум n, но не больше m вхождений foo*
`foo{,}`	*Выбирает любое количество вхождений foo, в том числе ноль*

  В отсутствие явного числового ограничения символу звёздочки по умолчанию соответствует любое количество меток (то есть `{,}`), а обычному слову — ровно одно вхождение (то есть `{1}`).

  После отличного от звёздочки элемента `lquery` могут быть добавлены модификаторы, позволяющие выбрать не только простые совпадения:

`@`	*Выбирает совпадение без учёта регистра; например, запросу a@ соответствует A*
`*`	*Выбирает любую метку с заданным префиксом, например, запросу foo* соответствует foobar*
`%`	*Выбирает в метке начальные слова, разделённые подчёркиваниями*

  Поведение модификатора `%` несколько нетривиальное. Он пытается найти соответствие по словам, а не по всей метке. Например, запрос `foo_bar%` выбирает `foo_bar_baz`, но не `foo_barbaz`. В сочетании с `*` сопоставление префикса применяется отдельно к каждому слову, например запрос `foo_bar%*` выбирает `foo1_bar2_baz`, но не `foo1_br2_baz`.

Также можно записать несколько различных меток, отличных от звёздочек, через знак `|` (обозначающий ИЛИ) для выбора любой из этих меток, либо добавить знак `!` (НЕ) в начале группы без звёздочки для выбора метки, не соответствующей ни одной из указанных альтернатив. Количественное ограничение, если оно требуется, задаётся в конце группы; это означает, что оно действует на всю группу в целом (то есть ограничивает число меток, соответствующих или не соответствующих одной из альтернатив).

Расширенный пример `lquery`:

```
Top.*{0,2}.sport*@.!football|tennis{1,}.Russ*|Spain
a.  b.     c.      d.                   e.
```

Этот запрос выберет путь, который:

a. начинается с метки `Top`

b. и затем включает от нуля до двух меток до

c. метки, начинающейся с префикса `sport` (без учёта регистра)

d. затем одну или несколько меток, отличных от `football` и `tennis`

e. и заканчивается меткой, которая начинается подстрокой `Russ` или в точности равна `Spain`.

- `ltxtquery` представляет подобный полнотекстовому запрос поиска подходящих значений `ltree`. Значение `ltxtquery` содержит слова, возможно с модификаторами `@`, `*`, `%` в конце; эти модификаторы имеют то же значение, что и в `lquery`. Слова можно объединять символами `&` (И), `|` (ИЛИ), `!` (НЕ) и скобками. Ключевое отличие от `lquery` состоит в том, что `ltxtquery` выбирает слова независимо от их положения в пути метки.

  Пример `ltxtquery`:

  ```
  Europe & Russia*@ & !Transportation
  ```

  Этот запрос выберет пути, содержащие метку `Europe` или любую метку с начальной подстрокой `Russia` (без учёта регистра), но не пути, содержащие метку `Transportation`. Положение этих слов в пути не имеет значения. Кроме того, когда применяется `%`, слово может быть сопоставлено с любым другим отделённым подчёркиваниями словом в метке, вне зависимости от его положения.

Замечание: `ltxtquery` допускает пробелы между символами, а `ltree` и `lquery` — нет.

F.31.2. Операторы и функции

Для типа `ltree` определены обычные операторы сравнения `=`, `<>`, `<`, `>`, `<=`, `>=`. Сравнение сортирует пути в порядке движения по дереву, а потомки узла сортируются по тексту метки. В дополнение к ним есть и специализированные операторы, перечисленные в Таблице F.17.

Таблица F.17. Операторы `ltree`

Оператор	Описание
`ltree @> ltree` → `boolean`	Левый аргумент является предком правого (или равен ему)?
`ltree <@ ltree` → `boolean`	Левый аргумент является потомком правого (или равен ему)?
`ltree ~ lquery` → `boolean` `lquery ~ ltree` → `boolean`	Значение `ltree` соответствует `lquery`?
`ltree ? lquery[]` → `boolean`	

Оператор	**Описание**
`lquery[] ? ltree` → `boolean`	Значение `ltree` соответствует одному из `lquery` в массиве?
`ltree @ ltxtquery` → `boolean` `ltxtquery @ ltree` → `boolean`	Значение `ltree` соответствует `ltxtquery`?
`ltree || ltree` → `ltree`	Соединяет два пути `ltree`.
`ltree || text` → `ltree` `text || ltree` → `ltree`	Преобразует текст в `ltree` и соединяет с путём.
`ltree[] @> ltree` → `boolean` `ltree <@ ltree[]` → `boolean`	Массив содержит предка `ltree`?
`ltree[] <@ ltree` → `boolean` `ltree @> ltree[]` → `boolean`	Массив содержит потомка `ltree`?
`ltree[] ~ lquery` → `boolean` `lquery ~ ltree[]` → `boolean`	Массив содержит какой-либо путь, соответствующий `lquery`?
`ltree[] ? lquery[]` → `boolean` `lquery[] ? ltree[]` → `boolean`	Массив `ltree` содержит путь, соответствующий какому-либо `lquery`?
`ltree[] @ ltxtquery` → `boolean` `ltxtquery @ ltree[]` → `boolean`	Массив содержит путь, соответствующий `ltxtquery`?
`ltree[] ?@> ltree` → `ltree`	Выдаёт первый элемент массива, являющийся предком `ltree`, или NULL, если такого элемента нет.
`ltree[] ?<@ ltree` → `ltree`	Выдаёт первый элемент массива, являющийся потомком `ltree`, или NULL, если такого элемента нет.
`ltree[] ?~ lquery` → `ltree`	Выдаёт первый элемент массива, соответствующий `lquery`, или NULL, если такого элемента нет.
`ltree[] ?@ ltxtquery` → `ltree`	Выдаёт первый элемент массива, соответствующий `ltxtquery`, или NULL, если такого элемента нет.

Операторы `<@`, `@>`, `@` и `~` имеют аналоги в виде `^<@`, `^@>`, `^@`, `^~`, которые отличатся только тем, что не используют индексы. Они полезны только для тестирования.

Доступные функции перечислены в Таблице F.18.

Таблица F.18. Функции `ltree`

Функция	Описание	Примеры
`subltree` (`ltree`, *`start`* `integer`, *`end`* `integer`) → `ltree`	Выдаёт подпуть `ltree` от позиции *`start`* до позиции *`end`*-1 (отсчитывая от 0).	`subltree('Top.Child1.Child2', 1, 2)` → `Child1`
`subpath` (`ltree`, *`offset`* `integer`, *`len`* `integer`) → `ltree`	Выдаёт подпуть `ltree`, начиная с позиции *`offset`*, длиной *`len`*. Если *`offset`* меньше нуля, подпуть начинается с этого смещения от конца пути. Если *`len`* меньше нуля, будет отброшено заданное число меток с конца строки.	`subpath('Top.Child1.Child2', 0, 2)` → `Top.Child1`
`subpath` (`ltree`, *`offset`* `integer`) → `ltree`	Выдаёт подпуть `ltree`, начиная с позиции *`offset`* и до конца пути. Если *`offset`* меньше нуля, подпуть начинается с этого смещения от конца пути.	`subpath('Top.Child1.Child2', 1)` → `Child1.Child2`
`nlevel` (`ltree`) → `integer`	Выдаёт число меток в пути.	`nlevel('Top.Child1.Child2')` → `3`
`index` (*`a`* `ltree`, *`b`* `ltree`) → `integer`	Выдаёт позицию первого вхождения *`b`* в *`a`*; -1, если искомого вхождения нет.	`index('0.1.2.3.5.4.5.6.8.5.6.8', '5.6')` → `6`
`index` (*`a`* `ltree`, *`b`* `ltree`, *`offset`* `integer`) → `integer`	Выдаёт позицию первого вхождения *`b`* в *`a`* или -1, если искомого вхождения нет. Поиск начинается с позиции *`offset`*; если *`offset`* меньше 0, начальное смещение *`-offset`* отсчитывается от конца пути.	`index('0.1.2.3.5.4.5.6.8.5.6.8', '5.6', -4)` → `9`
`text2ltree` (`text`) → `ltree`	Приводит значение `text` к типу `ltree`.	
`ltree2text` (`ltree`) → `text`	Приводит `ltree` к типу `text`.	
`lca` (`ltree` [, `ltree` [, ...]]) → `ltree`	Вычисляет наибольшего общего предка путей (принимая до 8 аргументов).	`lca('1.2.3', '1.2.3.4.5.6')` → `1.2`
`lca` (`ltree[]`) → `ltree`	Вычисляет наибольшего общего предка путей в массиве.	`lca(array['1.2.3'::ltree,'1.2.3.4'])` → `1.2`

F.31.3. Индексы

`ltree` поддерживает несколько типов индексов, которые могут ускорить означенные операции:

- B-дерево по значениям `ltree`: `<`, `<=`, `=`, `>=`, `>`
- Хеш-индекс по значениям `ltree`: `=`
- GiST по значениям `ltree` (класс операторов `gist_ltree_ops`): `<`, `<=`, `=`, `>=`, `>`, `@>`, `<@`, `@`, `~`, `?`

  Класс операторов GiST `gist_ltree_ops` аппроксимирует набор меток пути в виде сигнатуры битовой карты. В его необязательном целочисленном параметре `siglen` можно задать размер сигнатуры в байтах. Размер сигнатуры по умолчанию — 8 байт. Параметр может принимать положительные значения до 2024, кратные выравниванию по целым (`int`) (4 байта для большинства машин). При увеличении размера сигнатуры поиск работает точнее (сканируется меньшая область в индексе и меньше страниц кучи), но сам индекс становится больше.

Пример создания такого индекса с размером сигнатуры по умолчанию (8 байт):

```
CREATE INDEX path_gist_idx ON test USING GIST (path);
```

Пример создания такого индекса с длиной сигнатуры 100 байт:

```
CREATE INDEX path_gist_idx ON test USING GIST (path gist_ltree_ops(siglen=100));
```

- GiST-индекс по массиву `ltree[]` (`gist__ltree_ops` opclass): `ltree[] <@ ltree`, `ltree @> ltree[]`, `@`, `~`, `?`

  Класс операторов GiST `gist__ltree_ops` работает подобно `gist_ltree_ops` и также принимает в параметре длину сигнатуры. По умолчанию значение `siglen` в `gist__ltree_ops` составляет 28 байт.

  Пример создания такого индекса с размером сигнатуры по умолчанию (28 байт):

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path);
  ```

  Пример создания такого индекса с длиной сигнатуры 100 байт:

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path
   gist__ltree_ops(siglen=100));
  ```

  Примечание: Индекс этого типа является неточным.

F.31.4. Пример

Для этого примера используются следующие данные (это же описание данных находится в файле `contrib/ltree/ltreetest.sql` в дистрибутиве исходного кода):

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING GIST (path);
CREATE INDEX path_idx ON test USING BTREE (path);
CREATE INDEX path_hash_idx ON test USING HASH (path);
```

В итоге мы получаем таблицу `test`, наполненную данными, представляющими следующую иерархию:

```
                        Top
                     /   |  \
             Science Hobbies Collections
                 /       |              \
        Astronomy   Amateurs_Astronomy Pictures
```

```
            /  \                                |
Astrophysics  Cosmology                Astronomy
                                         /  |    \
                                  Galaxies Stars Astronauts
```

Мы можем выбрать потомки в иерархии наследования:

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
------------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

Несколько примеров выборки по путям:

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                     path
-----------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!pictures@.Astronomy.*';
                path
------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Ещё несколько примеров полнотекстового поиска:

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@';
                path
------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
                path
------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Образование пути с помощью функций:

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE path <@
 'Top.Science.Astronomy';
```

```
                 ?column?
------------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

Эту процедуру можно упростить, создав функцию SQL, вставляющую метку в определённую позицию в пути:

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@
 'Top.Science.Astronomy';
                ins_label
------------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.31.5. Трансформации

Расширение `ltree_plpython3u` реализует трансформации типа `ltree` для языка PL/Python. Если вы установите эти трансформации и укажете их при создании функции, значения `ltree` будут отображаться в списки Python. (Однако обратное преобразование не поддерживается.)

Внимание

Расширение, реализующее трансформации, настоятельно рекомендуется устанавливать в одну схему с `ltree`. Выбор какой-либо другой схемы, которая может содержать объекты, созданные злонамеренным пользователем, чреват угрозами безопасности во время установки расширения.

F.31.6. Авторы

Разработку осуществили Фёдор Сигаев (`<teodor@stack.net>`) и Олег Бартунов (`<oleg@sai.msu.su>`). Дополнительные сведения можно найти на странице *http://www.sai.msu.su/~megera/postgres/gist/*. Авторы выражают благодарность Евгению Родичеву за полезные дискуссии. Замечания и сообщения об ошибках приветствуются.

F.32. mchar — дополнительные типы данных для совместимости с Microsoft SQL Server

Модуль `mchar` предоставляет дополнительные типы данных для совместимости с Microsoft SQL Server (MS SQL).

F.32.1. Обзор

Этот модуль был разработан для улучшения поддержки системы 1С:Предприятие, самой популярной в России CRM и ERP-платформы.

Он содержит реализацию типов MCHAR и MVARCHAR, которые с точностью до ошибок совместимы с типами MS SQL CHAR и VARCHAR, соответственно. Эти типы используют библиотеку ICU для сравнения строк и преобразования регистра, так что они действуют одинаково в разных операционных системах.

В Postgres Pro также есть расширение citext, предоставляющие типы, подобные MCHAR. Но это расширение не воспроизводит поведение MS-SQL в отношении пробельных символов в конце строки.

Отличия от стандартных типов Postgres Pro CHAR и VARCHAR:

- Сравнение без учёта регистра
- Обработка пробельных символов в конце строки
- Эти типы всегда хранятся как двухбайтовые строки Unicode вне зависимости от кодировки базы данных.

F.32.2. Дополнительные типы

- `mchar` — аналог типа char в MS SQL
- `mvarchar` — аналог типа varchar в MS SQL

F.32.3. Функциональность типов MCHAR и MVARCHAR

- Определяется функция `length(str)`
- Определяется функция `substr(str, pos[, length])`
- Определяется оператор `||`, применяемый для конкатенации любых аргументов mchar и mvarchar
- Определяется набор операторов: `<`, `<=`, `=`, `>=` и `>` для сравнения символов без учёта регистра (ICU)
- Определяется набор операторов: `&<`, `&<=`, `&=`, `&>=` и `&>` для сравнения с учётом регистра (ICU)
- Неявное приведение между типами `mchar` и `mvarchar`
- Поддержка B-деревьев и хеш-индексов
- Поддержка оператора `LIKE [ESCAPE]`
- Поддержка оператора `SIMILAR TO [ESCAPE]`
- Поддержка оператора ~ (регулярные выражения POSIX)
- Поддержка индексов для оператора *LIKE*

F.32.4. Авторы

Олег Бартунов `<oleg@sai.msu.ru>`
Фёдор Сигаев `<teodor@sigaev.ru>`

F.33. online_analyze — обновление статистики после операций `INSERT`, `UPDATE`, `DELETE`, `SELECT INTO` или `COPY`.

Модуль `online_analyze` предоставляет набор функций, которые немедленно обновляют статистику после операций `INSERT`, `UPDATE`, `DELETE`, `SELECT INTO` или `COPY` в целевых таблицах.

F.33.1. Загрузка модуля

Чтобы использовать модуль `online_analyze`, загрузите разделяемую библиотеку:

```
LOAD 'online_analyze';
```

F.33.2. Настройка модуля

Вы можете настроить `online_analyze`, используя следующие дополнительные переменные (для них показаны значения по умолчанию):

- online_analyze.enable = on

  Включает `online_analyze`.

- online_analyze.verbose = on

  Выполняет `ANALYZE VERBOSE`.

  > **Примечание**
  >
  > Так как `verbose` является зарезервированным ключевым словом языка SQL, в SQL-запросах данный параметр нужно заключать в кавычки. Например:
  >
  > ```
  > ALTER SYSTEM SET "online_analyze.verbose" = 'off';
  > ```

- online_analyze.scale_factor = 0.1

  Процент от размера таблицы, при котором начинается немедленный анализ (этот параметр подобен autovacuum_analyze_scale_factor).

- online_analyze.threshold = 50

  Минимальное число изменений строк, после которого может начаться немедленный анализ (этот параметр подобен autovacuum_analyze_threshold).

- online_analyze.min_interval = 10000

  Минимальный интервал времени между вызовами `ANALYZE` для отдельной таблицы (в миллисекундах).

- online_analyze.table_type = "all"

  Типы таблиц, для которых выполняется немедленный анализ: `all` (все), `persistent` (постоянные), `temporary` (временные), `none` (никакие).

- online_analyze.exclude_tables = ""

  Список таблиц, исключаемых из немедленного анализа.

- online_analyze.include_tables = ""

  Список таблиц, подвергаемых немедленному анализу (параметр `online_analyze.include_tables` переопределяет `online_analyze.exclude_tables`).

- online_analyze.local_tracking = off

Включает в `online_analyze` отслеживание временных таблиц в рамках обслуживающего процесса. Когда эта переменная отключена (`off`), `online_analyze` использует для временных таблиц системную статистику по умолчанию.

- online_analyze.lower_limit = 0

  Минимальное число строк в таблиц, при котором будет срабатывать `online_analyze`.

- online_analyze.capacity_threshold = 100000

  Максимальное число временных таблиц, сохраняемых в локальном кеше.

F.33.3. Авторы

Фёдор Сигаев `<teodor@sigaev.ru>`

F.34. pageinspect — низкоуровневое исследование страниц баз данных

Модуль `pageinspect` предоставляет функции, позволяющие исследовать страницы баз данных на низком уровне, что бывает полезно для отладки. Все эти функции могут вызывать только суперпользователи.

F.34.1. Функции общего назначения

`get_raw_page(relname text, fork text, blkno bigint) returns bytea`

Функция `get_raw_page` считывает указанный блок отношения с заданным именем и возвращает копию значения `bytea`. Это позволяет получить одну согласованную во времени копию блока. В параметре *fork* нужно передать `'main'`, чтобы обратиться к основному слою данных, `'fsm'` — к карте свободного пространства, `'vm'` — к карте видимости, либо `'init'` — к слою инициализации.

`get_raw_page(relname text, blkno bigint) returns bytea`

Упрощённая версия `get_raw_page` для чтения данных из основного слоя. Синоним `get_raw_page(relname, 'main', blkno)`

`page_header(page bytea) returns record`

Функция `page_header` показывает поля, общие для всех страниц кучи и индекса Postgres Pro.

В качестве аргумента ей передаётся образ страницы, полученный в результате вызова `get_raw_page`. Например:

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn     | checksum | flags  | lower | upper | special | pagesize | version |
 prune_xid
-----------+----------+--------+-------+-------+---------+----------+---------
+-----------
 0/24A1B50 |        0 |      1 |   232 |   368 |    8192 |     8192 |       4 |
    0
```

Возвращаемые столбцы соответствуют полям в структуре `PageHeaderData`.

Поле `checksum` содержит контрольную сумму, сохранённую для страницы. Эта сумма может быть неверной при повреждении страницы. Если в данном экземпляре кластера контрольные суммы отключены, это значение не имеет смысла.

`page_checksum(page bytea, blkno bigint) returns smallint`

Функция `page_checksum` вычисляет контрольную сумму страницы, которая должна была бы находиться в заданном блоке.

В качестве аргумента ей передаётся образ страницы, полученный в результате вызова `get_raw_page`. Например:

```
test=# SELECT page_checksum(get_raw_page('pg_class', 0), 0);
 page_checksum
---------------
         13443
```

Заметьте, что вычисление контрольной суммы зависит от номера блока, поэтому обеим функциям нужно передавать одинаковые номера (за исключением случаев эзотерической отладки).

Контрольную сумму, вычисленную этой функцией, можно сравнить с полем `checksum` результата функции `page_header`. Если контрольные суммы для текущего экземпляра БД включены, эти значения должны быть равны.

`fsm_page_contents(page bytea) returns text`

Функция `fsm_page_contents` показывает внутреннюю структуру узла на странице FSM. Например:

```
test=# SELECT fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
```

Данный запрос выводит несколько текстовых строк, по одной строке для каждого узла двоичного дерева на заданной странице. Нулевые узлы при этом пропускаются. Также выводится так называемый указатель «вперёд», который указывает на следующий слот, получаемый с этой страницы.

F.34.2. Функции для исследования кучи

`heap_page_items(page bytea) returns setof record`

Функция `heap_page_items` показывает все указатели линейных блоков на странице кучи. Для используемых блоков также выводятся заголовки кортежей. При этом показываются все кортежи, независимо от того, были ли видны они в снимке MVCC в момент копирования исходной страницы.

В качестве аргумента ей передаётся образ страницы кучи, полученный в результате вызова `get_raw_page`. Например:

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

Распаковать биты флагов `t_infomask` и `t_infomask2` для кортежей кучи позволяет функция `heap_tuple_infomask_flags`.

`tuple_data_split(rel_oid oid, t_data bytea, t_infomask integer, t_infomask2 integer, t_bits text [, do_detoast bool]) returns bytea[]`

Функция `tuple_data_split` разделяет данные кортежей на атрибуты так, как это происходит внутри сервера.

```
test=# SELECT tuple_data_split('pg_class'::regclass, t_data, t_infomask,
 t_infomask2, t_bits) FROM heap_page_items(get_raw_page('pg_class', 0));
```

В качестве аргументов этой функции должны передаваться атрибуты, возвращаемые функцией `heap_page_items`.

Если параметр *`do_detoast`* равен `true`, полученные атрибуты будут распакованы по мере необходимости. Если он не задан, подразумевается `false`.

`heap_page_item_attrs(page bytea, rel_oid regclass [, do_detoast bool]) returns setof record`

Функция `heap_page_item_attrs` похожа на `heap_page_items`, но возвращает неструктурированное содержимое кортежа в виде массива атрибутов, которые могут быть распакованы, если установлен флаг *`do_detoast`* (по умолчанию они не распаковываются).

В качестве аргумента ей передаётся образ страницы кучи, полученный в результате вызова `get_raw_page`. Например:

```
test=# SELECT * FROM heap_page_item_attrs(get_raw_page('pg_class', 0),
 'pg_class'::regclass);
```

`heap_tuple_infomask_flags(t_infomask integer, t_infomask2 integer) returns record`

`heap_tuple_infomask_flags` декодирует значения `t_infomask` и `t_infomask2`, которые возвращает функция `heap_page_items`, и выдаёт массивы с именами флагов в понятном человеку виде: один, перечисляющий все флаги по отдельности, а другой — комбинированные. Например:

```
test=# SELECT t_ctid, raw_flags, combined_flags
```

```
         FROM heap_page_items(get_raw_page('pg_class', 0)),
           LATERAL heap_tuple_infomask_flags(t_infomask, t_infomask2)
         WHERE t_infomask IS NOT NULL OR t_infomask2 IS NOT NULL;
```

Вызывая эту функцию, ей следует передавать в аргументах те атрибуты, которые возвращает `heap_page_items`.

Комбинированные флаги выводятся для тех макросов на уровне исходного кода, в которых объединяются значения нескольких битов, например `HEAP_XMIN_FROZEN`.

F.34.3. Функции для индексов-B-деревьев

`bt_metap(relname text) returns record`

`bt_metap` выдаёт информацию о метастранице индекса B-дерево. Например:

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-------------+-------
magic                     | 340322
version                   | 4
root                      | 1
level                     | 0
fastroot                  | 1
fastlevel                 | 0
last_cleanup_num_delpages | 0
last_cleanup_num_tuples   | 230
allequalimage             | f
```

`bt_page_stats(relname text, blkno bigint) returns record`

`bt_page_stats` выдаёт сводную информацию по странице данных индекса-B-дерева. Например:

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 224
dead_items    | 0
avg_item_size | 16
page_size     | 8192
free_size     | 3668
btpo_prev     | 0
btpo_next     | 0
btpo_level    | 0
btpo_flags    | 3
```

`bt_multi_page_stats(relname text, blkno bigint, blk_count bigint) returns setof record`

`bt_multi_page_stats` выдаёт ту же информацию, что и `bt_page_stats`, но для каждой страницы диапазона страниц, начиная с *`blkno`* для *`blk_count`* страниц. При отрицательном значении *`blk_count`* информация выводится для всех страниц, начиная с *`blkno`* до конца индекса. Например:

```
test=# SELECT * FROM bt_multi_page_stats('pg_proc_oid_index', 5, 2);
-[ RECORD 1 ]-+-----
blkno         | 5
type          | l
live_items    | 367
dead_items    | 0
avg_item_size | 16
page_size     | 8192
free_size     | 808
```

```
btpo_prev     | 4
btpo_next     | 6
btpo_level    | 0
btpo_flags    | 1
-[ RECORD 2 ]-+-----
blkno         | 6
type          | l
live_items    | 367
dead_items    | 0
avg_item_size | 16
page_size     | 8192
free_size     | 808
btpo_prev     | 5
btpo_next     | 7
btpo_level    | 0
btpo_flags    | 1
```

`bt_page_items(relname text, blkno bigint) returns setof record`

`bt_page_items` выдаёт подробную информацию обо всех элементах на странице индекса B-дерево. Например:

```
test=# SELECT itemoffset, ctid, itemlen, nulls, vars, data, dead, htid, tids[0:2] AS
 some_tids
        FROM bt_page_items('tenk2_hundred', 5);
 itemoffset |   ctid    | itemlen | nulls | vars |          data           | dead |
 htid  |      some_tids
------------+-----------+---------+-------+------+-------------------------+------
+--------+----------------------
          1 | (16,1)    |      16 | f     | f    | 30 00 00 00 00 00 00 00 |      |
       |
          2 | (16,8292) |     616 | f     | f    | 24 00 00 00 00 00 00 00 | f    |
 (1,6)  | {"(1,6)","(10,22)"}
          3 | (16,8292) |     616 | f     | f    | 25 00 00 00 00 00 00 00 | f    |
 (1,18) | {"(1,18)","(4,22)"}
          4 | (16,8292) |     616 | f     | f    | 26 00 00 00 00 00 00 00 | f    |
 (4,18) | {"(4,18)","(6,17)"}
          5 | (16,8292) |     616 | f     | f    | 27 00 00 00 00 00 00 00 | f    |
 (1,2)  | {"(1,2)","(1,19)"}
          6 | (16,8292) |     616 | f     | f    | 28 00 00 00 00 00 00 00 | f    |
 (2,24) | {"(2,24)","(4,11)"}
          7 | (16,8292) |     616 | f     | f    | 29 00 00 00 00 00 00 00 | f    |
 (2,17) | {"(2,17)","(11,2)"}
          8 | (16,8292) |     616 | f     | f    | 2a 00 00 00 00 00 00 00 | f    |
 (0,25) | {"(0,25)","(3,20)"}
          9 | (16,8292) |     616 | f     | f    | 2b 00 00 00 00 00 00 00 | f    |
 (0,10) | {"(0,10)","(0,14)"}
         10 | (16,8292) |     616 | f     | f    | 2c 00 00 00 00 00 00 00 | f    |
 (1,3)  | {"(1,3)","(3,9)"}
         11 | (16,8292) |     616 | f     | f    | 2d 00 00 00 00 00 00 00 | f    |
 (6,28) | {"(6,28)","(11,1)"}
         12 | (16,8292) |     616 | f     | f    | 2e 00 00 00 00 00 00 00 | f    |
 (0,27) | {"(0,27)","(1,13)"}
         13 | (16,8292) |     616 | f     | f    | 2f 00 00 00 00 00 00 00 | f    |
 (4,17) | {"(4,17)","(4,21)"}
(13 rows)
```

Это листовая страница B-дерева. В ней все кортежи, относящиеся к таблице, оказались кортежами со списками идентификаторов (каждый содержит 100 шестибайтных идентификаторов). Помимо них, по смещению `itemoffset` 1 находится кортеж «верхний ключ». В данном примере `ctid` содержит закодированную информацию о каждом кортеже, хотя на уровне листьев записи часто содержат в `ctid` непосредственно TID кучи. В `tids` содержатся идентификаторы (TID) в виде списка идентификаторов.

На внутренней странице (она здесь не показана), компонент номера блока в `ctid` содержит «ссылку вниз», то есть номер блока другой страницы в самом индексе. Компонент смещения (второе число) в `ctid` содержит закодированную информацию о кортеже, в частности, количество столбцов в нём (при отсечении суффикса ненужные конечные столбцы могут быть отброшены). Когда столбцы отбрасываются, считается, что они содержат значение «минус бесконечность».

В столбце `htid` показывается TID кортежа в куче, вне зависимости от его нижележащего представления. Это значение должно совпадать с `ctid` или может быть декодировано из альтернативного представления, применяющегося в списках идентификаторов и в кортежах на внутренних страницах. В кортежах на внутренних страницах столбец с TID кучи на уровне реализации обычно отсутствует, что выглядит как значение NULL в столбце `htid`.

Заметьте, что первый элемент в любой, кроме самой правой, странице (то есть в любой странице с ненулевым значением в поле `btpo_next`) представляет собой «верхний ключ», то есть его поле `data` задаёт верхнюю границу для всех элементов, находящихся на странице, а поле `ctid` не указывает на другой блок. Кроме того, на внутренних страницах в первом действительном элементе данных (в первом элементе после верхнего ключа) обязательно будут отброшены все столбцы, и поэтому поле `data` не будет содержать фактического значения. Однако такой элемент содержит в своём поле `ctid` корректную ссылку вниз.

За дополнительной информацией о структуре индексов-B-деревьев обратитесь к Подразделу 62.1.4.1. Об исключении дубликатов и о списках идентификаторов подробнее рассказывается в Подразделе 62.1.4.3.

`bt_page_items(page bytea) returns setof record`

Функции `bt_page_items` также можно передать страницу в виде значения `bytea`. Для получения образа страницы, который она может принять, следует использовать функцию `get_raw_page`. Так, последний пример можно переписать следующим образом:

```
test=# SELECT itemoffset, ctid, itemlen, nulls, vars, data, dead, htid, tids[0:2] AS
 some_tids
        FROM bt_page_items(get_raw_page('tenk2_hundred', 5));
 itemoffset |   ctid    | itemlen | nulls | vars |          data           | dead |
 htid   |      some_tids
------------+-----------+---------+-------+------+-------------------------+------
+--------+----------------------
          1 | (16,1)    |      16 | f     | f    | 30 00 00 00 00 00 00 00 |      |
        |
          2 | (16,8292) |     616 | f     | f    | 24 00 00 00 00 00 00 00 | f    |
 (1,6)  | {"(1,6)","(10,22)"}
          3 | (16,8292) |     616 | f     | f    | 25 00 00 00 00 00 00 00 | f    |
 (1,18) | {"(1,18)","(4,22)"}
          4 | (16,8292) |     616 | f     | f    | 26 00 00 00 00 00 00 00 | f    |
 (4,18) | {"(4,18)","(6,17)"}
          5 | (16,8292) |     616 | f     | f    | 27 00 00 00 00 00 00 00 | f    |
 (1,2)  | {"(1,2)","(1,19)"}
          6 | (16,8292) |     616 | f     | f    | 28 00 00 00 00 00 00 00 | f    |
 (2,24) | {"(2,24)","(4,11)"}
          7 | (16,8292) |     616 | f     | f    | 29 00 00 00 00 00 00 00 | f    |
 (2,17) | {"(2,17)","(11,2)"}
          8 | (16,8292) |     616 | f     | f    | 2a 00 00 00 00 00 00 00 | f    |
 (0,25) | {"(0,25)","(3,20)"}
          9 | (16,8292) |     616 | f     | f    | 2b 00 00 00 00 00 00 00 | f    |
 (0,10) | {"(0,10)","(0,14)"}
```

```
         10 | (16,8292) |     616 | f     | f    | 2c 00 00 00 00 00 00 00 | f    |
 (1,3)  | {"(1,3)","(3,9)"}
         11 | (16,8292) |     616 | f     | f    | 2d 00 00 00 00 00 00 00 | f    |
 (6,28) | {"(6,28)","(11,1)"}
         12 | (16,8292) |     616 | f     | f    | 2e 00 00 00 00 00 00 00 | f    |
 (0,27) | {"(0,27)","(1,13)"}
         13 | (16,8292) |     616 | f     | f    | 2f 00 00 00 00 00 00 00 | f    |
 (4,17) | {"(4,17)","(4,21)"}
(13 rows)
```

В остальном данная функция работает так, как описано выше.

F.34.4. Функции для индексов BRIN

`brin_page_type(page bytea) returns text`

Функция `brin_page_type` возвращает тип страницы для заданной страницы индекса BRIN или выдаёт ошибку, если эта страница не является корректной страницей индекса BRIN. Например:

```
test=# SELECT brin_page_type(get_raw_page('brinidx', 0));
 brin_page_type
----------------
 meta
```

`brin_metapage_info(page bytea) returns record`

Функция `brin_metapage_info` возвращает разнообразные сведения о метастранице индекса BRIN. Например:

```
test=# SELECT * FROM brin_metapage_info(get_raw_page('brinidx', 0));
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+----------------
 0xA8109CFA |       1 |             4 |              2
```

`brin_revmap_data(page bytea) returns setof tid`

Функция `brin_revmap_data` выдаёт список идентификаторов кортежей со страницы сопоставлений зон индекса BRIN. Например:

```
test=# SELECT * FROM brin_revmap_data(get_raw_page('brinidx', 2)) LIMIT 5;
  pages
---------
 (6,137)
 (6,138)
 (6,139)
 (6,140)
 (6,141)
```

`brin_page_items(page bytea, index oid) returns setof record`

Функция `brin_page_items` выдаёт содержимое, сохранённое в странице данных BRIN. Например:

```
test=# SELECT * FROM brin_page_items(get_raw_page('brinidx', 5),
                                     'brinidx')
       ORDER BY blknum, attnum LIMIT 6;
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder | empty |    value
------------+--------+--------+----------+----------+-------------+-------
+--------------
        137 |      0 |      1 | t        | f        | f           | f     |
        137 |      0 |      2 | f        | f        | f           | f     | {1 ..
 88}
        138 |      4 |      1 | t        | f        | f           | f     |
```

```
        138 |       4 |       2 | f        | f        | f           | f     | {89 ..
 176}
        139 |       8 |       1 | t        | f        | f           | f     |
        139 |       8 |       2 | f        | f        | f           | f     | {177 ..
 264}
```

Возвращаемые столбцы соответствуют полям в структурах `BrinMemTuple` и `BrinValues`.

F.34.5. Функции для индексов GIN

`gin_metapage_info(page bytea) returns record`

Функция `gin_metapage_info` выдаёт информацию о метастранице индекса GIN. Например:

```
test=# SELECT * FROM gin_metapage_info(get_raw_page('gin_index', 0));
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 7
n_entry_pages    | 6
n_data_pages     | 0
n_entries        | 693
version          | 2
```

`gin_page_opaque_info(page bytea) returns record`

Функция `gin_page_opaque_info` выдаёт информацию из непрозрачной области индекса GIN, например, тип страницы. Например:

```
test=# SELECT * FROM gin_page_opaque_info(get_raw_page('gin_index', 2));
 rightlink | maxoff |         flags
-----------+--------+------------------------
         5 |      0 | {data,leaf,compressed}
(1 row)
```

`gin_leafpage_items(page bytea) returns setof record`

Функция `gin_leafpage_items` выдаёт информацию о данных, хранящихся в сжатой странице индекса GIN на уровне листьев. Например:

```
test=# SELECT first_tid, nbytes, tids[0:5] AS some_tids
        FROM gin_leafpage_items(get_raw_page('gin_test_idx', 2));
 first_tid | nbytes |                        some_tids
-----------+--------+----------------------------------------------------------
 (8,41)    |    244 | {"(8,41)","(8,43)","(8,44)","(8,45)","(8,46)"}
 (10,45)   |    248 | {"(10,45)","(10,46)","(10,47)","(10,48)","(10,49)"}
 (12,52)   |    248 | {"(12,52)","(12,53)","(12,54)","(12,55)","(12,56)"}
 (14,59)   |    320 | {"(14,59)","(14,60)","(14,61)","(14,62)","(14,63)"}
 (167,16)  |    376 | {"(167,16)","(167,17)","(167,18)","(167,19)","(167,20)"}
 (170,30)  |    376 | {"(170,30)","(170,31)","(170,32)","(170,33)","(170,34)"}
 (173,44)  |    197 | {"(173,44)","(173,45)","(173,46)","(173,47)","(173,48)"}
(7 rows)
```

F.34.6. Функции для индексов GiST

`gist_page_opaque_info(page bytea) returns record`

Функция `gist_page_opaque_info` выдаёт информацию из непрозрачной области страницы индекса GiST, например значения NSN, rightlink и тип страницы. Например:

```
test=# SELECT * FROM gist_page_opaque_info(get_raw_page('test_gist_idx', 2));
 lsn | nsn | rightlink | flags
-----+-----+-----------+--------
 0/1 | 0/0 |         1 | {leaf}
(1 row)
```

`gist_page_items(page bytea, index_oid regclass) returns setof record`

Функция `gist_page_items` возвращает информацию о данных, хранящихся на странице индекса GiST. Например:

```
test=# SELECT * FROM gist_page_items(get_raw_page('test_gist_idx', 0),
 'test_gist_idx');
 itemoffset |   ctid    | itemlen | dead |             keys
------------+-----------+---------+------+-------------------------------
          1 | (1,65535) |      40 | f    | (p)=("(185,185),(1,1)")
          2 | (2,65535) |      40 | f    | (p)=("(370,370),(186,186)")
          3 | (3,65535) |      40 | f    | (p)=("(555,555),(371,371)")
          4 | (4,65535) |      40 | f    | (p)=("(740,740),(556,556)")
          5 | (5,65535) |      40 | f    | (p)=("(870,870),(741,741)")
          6 | (6,65535) |      40 | f    | (p)=("(1000,1000),(871,871)")
(6 rows)
```

`gist_page_items_bytea(page bytea) returns setof record`

Функция работает так же, как `gist_page_items`, но возвращает данные ключа в виде неразобранных двоичных данных `bytea`. Поскольку она не пытается декодировать ключ, ей не требуется знать, какой индекс задействован. Например:

```
test=# SELECT * FROM gist_page_items_bytea(get_raw_page('test_gist_idx', 0));
 itemoffset |   ctid    | itemlen | dead |
 key_data
------------+-----------+---------+------+-----------------------------------------
-------------------------------------------
          1 | (1,65535) |      40 | f    |
 \x00000100ffff28000000000000c064400000000000c06440000000000000f03f000000000000f03f
          2 | (2,65535) |      40 | f    |
 \x00000200ffff28000000000000c074400000000000c074400000000000e064400000000000e06440
          3 | (3,65535) |      40 | f    |
 \x00000300ffff280000000000000207f400000000000207f400000000000d074400000000000d07440
          4 | (4,65535) |      40 | f    |
 \x00000400ffff28000000000000c084400000000000c08440000000000000307f400000000000307f40
          5 | (5,65535) |      40 | f    |
 \x00000500ffff28000000000000f089400000000000f089400000000000c884400000000000c88440
          6 | (6,65535) |      40 | f    |
 \x00000600ffff28000000000000208f400000000000208f400000000000f889400000000000f88940
          7 | (7,65535) |      40 | f    |
 \x00000700ffff28000000000000408f400000000000408f400000000000288f400000000000288f40
(7 rows)
```

F.34.7. Функции для хеш-индексов

`hash_page_type(page bytea) returns text`

Функция `hash_page_type` возвращает тип страницы для заданной страницы хеш-индекса. Например:

```
test=# SELECT hash_page_type(get_raw_page('con_hash_index', 0));
 hash_page_type
----------------
 metapage
```

`hash_page_stats(page bytea) returns setof record`

Функция `hash_page_stats` возвращает информацию о странице группы или переполнения хеш-индекса. Например:

```
test=# SELECT * FROM hash_page_stats(get_raw_page('con_hash_index', 1));
-[ RECORD 1 ]---+-----------
live_items      | 407
dead_items      | 0
page_size       | 8192
free_size       | 8
hasho_prevblkno | 4096
hasho_nextblkno | 8474
hasho_bucket    | 0
hasho_flag      | 66
hasho_page_id   | 65408
```

`hash_page_items(page bytea) returns setof record`

Функция `hash_page_items` возвращает информацию о данных, хранящихся на странице группы или переполнения хеш-индекса. Например:

```
test=# SELECT * FROM hash_page_items(get_raw_page('con_hash_index', 1)) LIMIT 5;
 itemoffset |   ctid    |    data
------------+-----------+------------
          1 | (899,77)  | 1053474816
          2 | (897,29)  | 1053474816
          3 | (894,207) | 1053474816
          4 | (892,159) | 1053474816
          5 | (890,111) | 1053474816
```

`hash_bitmap_info(index oid, blkno bigint) returns record`

Функция `hash_bitmap_info` показывает состояние бита в странице битовой карты для определённой страницы переполнения хеш-индекса. Например:

```
test=# SELECT * FROM hash_bitmap_info('con_hash_index', 2052);
 bitmapblkno | bitmapbit | bitstatus
-------------+-----------+-----------
          65 |         3 | t
```

`hash_metapage_info(page bytea) returns record`

`hash_metapage_info` возвращает информацию, хранящуюся в метастранице хеш-индекса. Например:

```
test=# SELECT magic, version, ntuples, ffactor, bsize, bmsize, bmshift,
test-#     maxbucket, highmask, lowmask, ovflpoint, firstfree, nmaps, procid,
test-#     regexp_replace(spares::text, '(,0)*}', '}') as spares,
test-#     regexp_replace(mapp::text, '(,0)*}', '}') as mapp
test-# FROM hash_metapage_info(get_raw_page('con_hash_index', 0));
-[ RECORD 1 ]-------------------------------------------------
------------------------------
magic     | 105121344
version   | 4
ntuples   | 500500
ffactor   | 40
bsize     | 8152
bmsize    | 4096
bmshift   | 15
maxbucket | 12512
highmask  | 16383
```

```
lowmask   | 8191
ovflpoint | 28
firstfree | 1204
nmaps     | 1
procid    | 450
spares    | {0,0,0,0,0,0,1,1,1,1,1,1,1,1,3,4,4,4,45,55,58,59,
508,567,628,704,1193,1202,1204}
mapp      | {65}
```

F.35. passwordcheck — проверка надёжности пароля

Модуль `passwordcheck` проверяет пароли пользователей, задаваемые командами CREATE ROLE и ALTER ROLE (модуль не работает, если пользователю назначен нестандартный профиль). Если пароль признаётся слишком слабым, он не принимается и команда завершается ошибкой.

Чтобы задействовать этот модуль, добавьте строку `'$libdir/passwordcheck'` в переменную shared_preload_libraries в `postgresql.conf`, а затем перезапустите сервер.

Внимание

Чтобы незашифрованные пароли не передавались по сети, не записывались в журнал сервера и не стали каким-либо образом известны администратору баз данных, Postgres Pro позволяет пользователю передавать предварительно зашифрованные пароли. Используя это, клиентские программы могут шифровать пароль, прежде чем передавать его серверу.

Это ограничивает полезность модуля `passwordcheck`, так как в этом случае можно только попытаться угадать пароль. Поэтому использовать `passwordcheck` не рекомендуется, когда требуется высокий уровень безопасности. Более безопасно будет применить внешний вариант проверки подлинности, например GSSAPI (см. Главу 19), а не использовать пароли, хранящиеся в базе данных.

F.36. pg_buffercache — информация о состоянии буферного кеша Postgres Pro

Модуль `pg_buffercache` даёт возможность понять, что происходит в общем кеше буферов в реальном времени, а также позволяет вытеснять данные из него на низком уровне в целях тестирования.

Этот модуль предоставляет функцию `pg_buffercache_pages()` (обёрнута в представление `pg_buffercache`), а также функции `pg_buffercache_summary()`, `pg_buffercache_usage_counts()` и `pg_buffercache_evict()`.

Функция `pg_buffercache_pages()` возвращает набор записей, каждая строка которого описывает состояние одного буфера в общем кеше. Для удобства пользования обёрткой функции является представление `pg_buffercache`.

Функция `pg_buffercache_summary()` возвращает одну строку со сводной информацией о состоянии общего кеша буферов.

Функция `pg_buffercache_usage_counts()` возвращает набор записей, в каждой строке которого указано количество буферов в разбивке по счётчику использования.

По умолчанию использование упомянутых выше функций разрешено только суперпользователям и ролям с правами роли `pg_monitor`. Дать доступ другим ролям можно с помощью `GRANT`.

Функция `pg_buffercache_evict()` позволяет вытеснять блоки из пула буферов по идентификатору блока. Вызывать эту функцию разрешено только суперпользователям.

F.36.1. Представление `pg_buffercache`

Определения столбцов, содержащихся в представлении, показаны в Таблице F.19.

Таблица F.19. Столбцы `pg_buffercache`

Тип столбца **Описание**
`bufferid` `integer` ID, в диапазоне 1..`shared_buffers`
`relfilenode` `oid` (ссылается на `pg_class` `.relfilenode`) Номер файлового узла для отношения
`reltablespace` `oid` (ссылается на `pg_tablespace` `.oid`) OID табличного пространства, содержащего отношение
`reldatabase` `oid` (ссылается на `pg_database` `.oid`) OID базы данных, содержащей отношение
`relforknumber` `smallint` Номер слоя в отношении
`relblocknumber` `bigint` Номер страницы в отношении
`isdirty` `boolean` Страница загрязнена?
`usagecount` `smallint` Счётчик обращений по часовой стрелке
`pinning_backends` `integer` Число обслуживающих процессов, закрепивших этот буфер

Для каждого буфера в общем кеше выдаётся одна строка. Для неиспользуемых буферов все поля равны NULL, за исключением `bufferid`. Общие системные каталоги показываются как относящиеся к базе данных под номером 0.

Так как кеш используется совместно всеми базами данных, обычно в нём находятся и страницы из отношений, не принадлежащих текущей базе данных. Это означает, что для некоторых строк при соединении с `pg_class` не найдутся соответствующие строки, либо соединение будет некорректным. Если вы хотите выполнить соединение с `pg_class`, будет правильным ограничить соединение строками, в которых `reldatabase` содержит OID текущей базы данных или ноль.

Для копирования данных состояния, которые будут отображаться, блокировки в менеджере буферов не устанавливаются. Поэтому обращения к представлению `pg_buffercache` меньше сказываются на обычной активности буферов, но набор результатов, полученный для всех буферов, в целом может оказаться несогласованным. Однако внутри каждого отдельного буфера согласованность информации гарантируется.

F.36.2. Функция `pg_buffercache_summary()`

Определения столбцов, возвращаемых функцией, показаны в Таблице F.20.

Таблица F.20. Выходные столбцы `pg_buffercache_summary()`

Тип столбца Описание
`buffers_used int4` Число используемых общих буферов
`buffers_unused int4` Число неиспользуемых общих буферов
`buffers_dirty int4` Число загрязнённых общих буферов
`buffers_pinned int4` Число закреплённых общих буферов
`usagecount_avg float8` Среднее значение счётчика использования общих буферов

Функция `pg_buffercache_summary()` возвращает одну строку со сводной информацией о состоянии всех общих буферов. Подобную и более подробную информацию можно получить из представления `pg_buffercache`, но `pg_buffercache_summary()` обходится существенно дешевле.

Как и представление `pg_buffercache`, `pg_buffercache_summary()` не получает блокировки менеджера буферов, поэтому при параллельной работе в базе данных допустима незначительная погрешность результатов функции.

F.36.3. Функция `pg_buffercache_usage_counts()`

Определения столбцов, возвращаемых функцией, показаны в Таблице F.21.

Таблица F.21. Выходные столбцы `pg_buffercache_usage_counts()`

Тип столбца Описание
`usage_count int4` Возможное значение счётчика использования буферов
`buffers int4` Число буферов в разбивке по счётчику использования
`dirty int4` Число загрязнённых буферов в разбивке по счётчику использования
`pinned int4` Число закреплённых буферов в разбивке по счётчику использования

Функция `pg_buffercache_usage_counts()` возвращает набор строк со сводной информацией о состоянии всех общих буферов, агрегированных по возможным значениям счётчика использования. Подобную и более подробную информацию можно получить с помощью представления `pg_buffercache`, но `pg_buffercache_usage_counts()` обходится существенно дешевле.

Как и представление `pg_buffercache`, `pg_buffercache_usage_counts()` не получает блокировки менеджера буферов, поэтому при параллельной работе в базе данных допустима незначительная погрешность результатов функции.

F.36.4. Функция `pg_buffercache_evict()`

Функция `pg_buffercache_evict()` принимает идентификатор буфера, который отображается в столбце `bufferid` представления `pg_buffercache`. Возвращает true в случае успеха и false, если буфер был недействителен, не мог быть вытеснен, так как был закреплён, или снова загрязнился после попытки записи. После завершения функции результат сразу становится устаревшим, поскольку из-за параллельно выполняемых операций буфер может снова стать действительным в любое время. Эта функция предназначена только для тестирования.

F.36.5. Пример вывода

```
regression=# SELECT n.nspname, c.relname, count(*) AS buffers
             FROM pg_buffercache b JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             JOIN pg_namespace n ON n.oid = c.relnamespace
             GROUP BY n.nspname, c.relname
             ORDER BY 3 DESC
             LIMIT 10;

  nspname   |        relname         | buffers
------------+------------------------+---------
 public     | delete_test_table      |     593
 public     | delete_test_table_pkey |     494
 pg_catalog | pg_attribute           |     472
 public     | quad_poly_tbl          |     353
 public     | tenk2                  |     349
 public     | tenk1                  |     349
 public     | gin_test_idx           |     306
 pg_catalog | pg_largeobject         |     206
 public     | gin_test_tbl           |     188
 public     | spgist_text_tbl        |     182
(10 rows)


regression=# SELECT * FROM pg_buffercache_summary();
 buffers_used | buffers_unused | buffers_dirty | buffers_pinned | usagecount_avg
--------------+----------------+---------------+----------------+----------------
          248 |        2096904 |            39 |              0 |       3.141129
(1 row)


regression=# SELECT * FROM pg_buffercache_usage_counts();
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 |   14650 |     0 |      0
           1 |    1436 |   671 |      0
           2 |     102 |    88 |      0
           3 |      23 |    21 |      0
```

```
               4 |        9 |      7 |      0
               5 |      164 |    106 |      0
(6 rows)
```

F.36.6. Авторы

Марк Кирквуд `<markir@paradise.net.nz>`

Предложения по конструкции: Нейл Конвей `<neilc@samurai.com>`

Советы по отладке: Том Лейн `<tgl@sss.pgh.pa.us>`

F.37. pgcrypto — функции защитного преобразования

Модуль `pgcrypto` предоставляет функции защитного преобразования для Postgres Pro.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

Для работы `pgcrypto` требуется OpenSSL — модуль не будет установлен, если Postgres Pro был собран без OpenSSL.

F.37.1. Стандартные функции хеширования

F.37.1.1. `digest()`

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

Вычисляет двоичный хеш данных (*`data`*). Параметр *`type`* выбирает используемый алгоритм. Поддерживаются стандартные алгоритмы: `md5`, `sha1`, `sha224`, `sha256`, `sha384` и `sha512`. Кроме того, эта функция автоматически поддерживает любые другие алгоритмы хеширования, которые поддерживает OpenSSL.

Если вы хотите получить дайджест в виде шестнадцатеричной строки, примените `encode()` к результату. Например:

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.37.1.2. `hmac()`

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

Вычисляет имитовставку на основе хеша для данных *`data`* с ключом *`key`*. Параметр *`type`* имеет то же значение, что и для `digest()`.

Эта функция похожа на `digest()`, но вычислить хеш с ней можно, только зная ключ. Это защищает от сценария подмены данных и хеша вместе с ними.

Если размер ключа больше размера блока хеша, он сначала хешируется, а затем используется в качестве ключа хеширования данных.

F.37.2. Функции хеширования пароля

Функции `crypt()` и `gen_salt()` разработаны специально для хеширования паролей. Функция `crypt()` выполняет хеширование, а `gen_salt()` подготавливает параметры алгоритма для неё.

Алгоритмы в `crypt()` отличаются от обычных алгоритмов хеширования MD5 и SHA1 в следующих аспектах:

1. Они медленные. Так как объём данных невелик, это единственный способ усложнить перебор паролей.
2. Они используют случайное значение, называемое *солью*, чтобы у пользователей с одинаковыми паролями защищённые пароли оказывались разными. Это также обеспечивает дополнительную защиту от получения обратного алгоритма.
3. Они включают в результат тип алгоритма, что допускает сосуществование паролей, хешированных разными алгоритмами.
4. Некоторые из них являются адаптируемыми — то есть с ростом производительности компьютеров эти алгоритмы можно настроить так, чтобы они стали медленнее, при этом сохраняя совместимость с существующими паролями.

В Таблице F.22 перечислены алгоритмы, поддерживаемые функцией `crypt()`.

Таблица F.22. Алгоритмы, которые поддерживает `crypt()`

Алгоритм	Макс. длина пароля	Адаптивный?	Размер соли (бит)	Размер результата	Описание
`bf`	72	да	128	60	На базе Blowfish, вариация 2a
`md5`	без ограничений	нет	48	34	crypt на базе MD5
`xdes`	8	да	24	20	Расширенный DES
`des`	8	нет	12	13	Изначальный crypt из UNIX

F.37.2.1. `crypt()`

```
crypt(password text, salt text) returns text
```

Вычисляет хеш пароля (*`password`*) в стиле crypt(3). Для сохранения нового пароля необходимо вызвать `gen_salt()`, чтобы сгенерировать новое значение соли (*`salt`*). Для проверки пароля нужно передать сохранённое значение хеша в параметре *`salt`* и проверить, соответствует ли результат сохранённому значению.

Пример установки нового пароля:

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

Пример проверки пароля:

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

Этот запрос возвращает `true`, если введённый пароль правильный.

F.37.2.2. `gen_salt()`

```
gen_salt(type text [, iter_count integer ]) returns text
```

Вычисляет новое случайное значение соли для функции `crypt()`. Строка соли также говорит `crypt()`, какой алгоритм использовать.

Параметр *`type`* задаёт алгоритм хеширования. Принимаются следующие варианты: `des`, `xdes`, `md5` и `bf`.

Параметр *`iter_count`* позволяет пользователю указать счётчик итераций для алгоритма, который его принимает. Чем больше это число, тем больше времени уйдёт на вычисление хеша пароля, а значит, тем больше времени понадобится, чтобы взломать его. Хотя со слишком большим значением время вычисления хеша может вырасти до нескольких лет — это вряд ли практично. Когда параметр *`iter_count`* опускается, применяется количество итераций по умолчанию. Множество допустимых значений для *`iter_count`* зависит от алгоритма, как показано в Таблице F.23.

Таблица F.23. Счётчики итераций для `crypt()`

Алгоритм	По умолчанию	Мин.	Макс.
`xdes`	725	1	16777215
`bf`	6	4	31

Для `xdes` есть дополнительное ограничение: счётчик итераций должен быть нечётным.

При выборе подходящего числа итераций учтите, что оригинальный алгоритм DES crypt был рассчитан так, чтобы выдавать 4 хеша в секунду на компьютерах того времени. Если за секунду будет вычисляться меньше 4 хешей, скорее всего, возникнут определённые неудобства при пользовании. С другой стороны, скорость больше, чем 100 хешей в секунду, вероятно, будет слишком высокой.

В Таблице F.24 дана сводка относительной скорости различных алгоритмов хеширования. В таблице показано, сколько времени уйдёт на перебор всех комбинацией символов в восьмисимвольном пароле, в предположении, что пароль содержит только буквы в нижнем регистре, либо буквы в верхнем и нижнем регистре, а также цифры. В строках `crypt-bf` числа после косой черты показывают значение параметра *`iter_count`* функции `gen_salt`.

Таблица F.24. Скорости алгоритмов хеширования

Алгоритм	**Хешей/сек.**	**Для** `[a-z]`	**Для** `[A-Za-z0-9]`	**Длительность относительно** `md5`
`crypt-bf/8`	1792	4 года	3927 лет	100k
`crypt-bf/7`	3648	2 года	1929 лет	50k
`crypt-bf/6`	7168	1 год	982 лет	25k
`crypt-bf/5`	13504	188 дней	521 лет	12.5k
`crypt-md5`	171584	15 дней	41 год	1k
`crypt-des`	23221568	157.5 минут	108 дней	7
`sha1`	37774272	90 минут	68 дней	4
`md5` (хеш)	150085504	22.5 минут	17 дней	1

Замечания:

- Для расчётов использовался процессор Intel Mobile Core i3.
- Показатели алгоритмов `crypt-des` и `crypt-md5` взяты из вывода теста программы John the Ripper v1.6.38.
- Показатели `md5` получены программой mdcrack 1.2.
- Показатели `sha1` получены программой lcrack-20031130-beta.
- Показатели `crypt-bf` получены простой программой, обрабатывающей в цикле 1000 паролей из 8 символов. Таким способом можно показать скорость с разным числом итераций. Для справки: `john -test` показывает 13506 циклов/с для `crypt-bf/5`. (Это очень небольшое различие в результатах согласуется с тем фактом, что реализация `crypt-bf` в `pgcrypto` не отличается от применяемой в программе John the Ripper.)

Заметьте, что вариант «перепробовать все комбинации» не вполне реалистичен. Обычно перебор паролей производится с применением словарей, которые содержат и обычные слова, и их различные видоизменения. Поэтому даже похожие на слова пароли обычно можно подобрать быстрее, чем за указанное время, тогда как 6-символьный несловесный пароль может избежать взлома. А может и не избежать.

F.37.3. Функции защитного преобразования на базе PGP

Функции, описанные здесь, реализуют часть стандарта OpenPGP (*RFC 4880*), относящуюся к защитному преобразованию. Они поддерживают защитное преобразование как с симметричным, так и с открытым ключом.

Защищённое PGP сообщение состоит из 2 частей или *пакетов*:

- Пакет, содержащий сеансовый ключ — либо симметричный, либо открытый (в защищённом виде).

- Пакет, содержащий данные, защищённые сеансовым ключом.

При защитном преобразовании с симметричным ключом (то есть, паролем):

1. Заданный пароль хешируется по алгоритму String2Key (S2K). Этот алгоритм подобен алгоритмам `crypt()` — специально замедлен и добавляет случайную соль — но на выход выдаёт двоичный ключ полной длины.
2. Если требуется отдельный сеансовый ключ, генерируется новый случайный ключ. В противном случае в качестве сеансового будет использоваться непосредственно ключ S2K.
3. Когда используется непосредственно ключ S2K, в пакет сеансового ключа помещаются только параметры S2K. В противном случае сеансовый ключ защищается ключом S2K и помещается в пакет сеансового ключа.

При защитном преобразовании с открытым ключом:

1. Генерируется новый случайный сеансовый ключ.
2. Он защищается открытым ключом и помещается в пакет сеансового ключа.

В любом случае данные, которые должны быть защищены, обрабатываются так:

1. Необязательная подготовка данных: сжатие, перекодировка в UTF-8 и/или преобразование концов строк.
2. Перед данными добавляется блок случайных байт. Это равносильно использованию случайного вектора инициализации.
3. В конце добавляется хеш SHA1 случайного префикса и данных.
4. Всё это защищается сеансовым ключом и помещается в пакет данных.

F.37.3.1. `pgp_sym_encrypt()`

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

Защищает данные (*`data`*) симметричным ключом PGP *`psw`*. В *`options`* могут передаваться параметры защитного преобразования, описанные ниже.

F.37.3.2. `pgp_sym_decrypt()`

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

Делает обратное преобразование сообщения, защищённого симметричным ключом PGP.

Обратно преобразовывать данные `bytea` функцией `pgp_sym_decrypt` запрещено. Это ограничение введено, чтобы не допустить вывода некорректных символьных данных. Обратно преобразовывать изначально текстовые данные с помощью `pgp_sym_decrypt_bytea` можно без ограничений.

Аргумент *`options`* может содержать параметры защитного преобразования, описанные ниже.

F.37.3.3. `pgp_pub_encrypt()`

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

Защищает данные (*`data`*) открытым ключом PGP (*`key`*). Если передать этой функции закрытый ключ, она выдаст ошибку.

Аргумент *`options`* может содержать параметры защитного преобразования, описанные ниже.

F.37.3.4. `pgp_pub_decrypt()`

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
```

```
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns
 bytea
```

Обратно преобразовывает сообщение, защищённое открытым ключом. В *`key`* должен передаваться закрытый ключ, соответствующий открытому ключу, применяющемуся при защитном преобразовании. Если секретный ключ защищён паролем, этот пароль нужно передать в параметре *`psw`*. Если пароля нет, но необходимо передать параметры, вы должны передать пустой пароль.

Обратно преобразовывать данные `bytea` функцией `pgp_pub_decrypt` запрещено. Это ограничение введено, чтобы исключить вывод недопустимых символьных данных. Обратно преобразовывать изначально текстовые данные с помощью `pgp_pub_decrypt_bytea` можно без ограничений.

Аргумент *`options`* может содержать параметры защитного преобразования, описанные ниже.

F.37.3.5. `pgp_key_id()`

```
pgp_key_id(bytea) returns text
```

`pgp_key_id` извлекает идентификатор ключа из открытого или закрытого ключа PGP. Она также может выдать идентификатор ключа, которым были защищены данные, если ей передаётся защищённое сообщение.

Она может выдать два специальных идентификатора ключа:

- `SYMKEY`

  Сообщение защищено симметричным ключом.

- `ANYKEY`

  Сообщение защищено открытом ключом, но идентификатор ключа был удалён. Это означает, что вам надо будет перепробовать ключи, чтобы подобрать подходящий. Сама библиотека `pgcrypto` не генерирует такие сообщения.

Заметьте, что разные ключи могут иметь одинаковый идентификатор. Это редкое, но не невероятное явление. В таком случае клиентское приложение должно пытаться обратно преобразовать данные с каждым ключом, пока не найдёт подходящий — примерно так же, как и с `ANYKEY`.

F.37.3.6. `armor()`, `dearmor()`

```
armor(data bytea [ , keys text[], values text[] ]) returns text
dearmor(data text) returns bytea
```

Эти функции переводят двоичные данные в/из формата PGP «ASCII Armor», по сути представляющий собой кодировку Base64 с контрольными суммами и дополнительным форматированием.

Если задаются массивы *`keys`* и *`values`*, для каждой пары ключ/значения в формат Armor добавляется *заголовок Armor*. Оба массива должны быть одномерными и иметь одинаковую длину. Задаваемые ключи и значения могут содержать только символы ASCII.

F.37.3.7. `pgp_armor_headers`

```
pgp_armor_headers(data text, key out text, value out text) returns setof record
```

Функция `pgp_armor_headers()` извлекает заголовки Armor из параметра *`data`*. Она возвращает набор строк с двумя столбцами, key и value. Если в ключах или значениях оказываются символы не ASCII, они воспринимаются как UTF-8.

F.37.3.8. Параметры функций PGP

Имена параметров подобны принятым в GnuPG. Значение параметра должно задаваться после знака равно; друг от друга параметры отделяются запятыми. Например:

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

Все эти параметры, кроме `convert-crlf`, применяются только к функциям защитного преобразования. Функции защитного преобразования получают параметры из данных PGP.

Вероятно, самые интересные параметры — это `compress-algo` и `unicode-mode`. Остальные должны иметь достаточно адекватные значения по умолчанию.

F.37.3.8.1. cipher-algo

Выбирает алгоритм защитного преобразования.

Значения: bf, aes128, aes192, aes256, 3des, cast5
По умолчанию: aes128
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.37.3.8.2. compress-algo

Выбирает алгоритм сжатия. Принимается, только если Postgres Pro собран с zlib.

Значения:
  0 — без сжатия
  1 — сжатие ZIP
  2 — сжатие ZLIB (= ZIP плюс метаданные и CRC блоков)
По умолчанию: 0
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.37.3.8.3. compress-level

Определяет уровень сжатия. Чем больше уровень, тем меньшего объёма результат, но длительнее процесс. Значение 0 отключает сжатие.

Значения: 0, 1-9
По умолчанию: 6
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.37.3.8.4. convert-crlf

Определяет, преобразовывать ли `\n` в `\r\n` при защитном преобразовании и `\r\n` в `\n` при обратном преобразовании. В RFC 4880 требуется, чтобы текстовые данные хранились с переводами строк в виде `\r\n`. Воспользуйтесь этим параметром, чтобы поведение полностью соответствовало RFC.

Значения: 0, 1
По умолчанию: 0
Применим к: pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.37.3.8.5. disable-mdc

Не защищать данные хешем SHA-1. Единственная разумная причина использовать этот параметр — добиться совместимости с древними программами PGP, вышедшими до того, как в RFC 4880 была предусмотрена защита пакетов с SHA-1. Все последние реализации с gnupg.org и pgp.com прекрасно поддерживают это.

Значения: 0, 1
По умолчанию: 0
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.37.3.8.6. sess-key

Использовать отдельный сеансовый ключ. Для защитного преобразования с открытым ключом всегда используется отдельный сеансовый ключ; этот параметр предназначен для защитного преобразования с симметричным ключом, которое по умолчанию использует непосредственно ключ S2K.

Значения: 0, 1
По умолчанию: 0

Применим к: pgp_sym_encrypt

F.37.3.8.7. s2k-mode

Режим алгоритма S2K.

Значения:
  0 — Без соли. Опасно!
  1 — С солью, но с фиксированным числом итераций.
  3 — С переменным числом итераций.
По умолчанию: 3
Применим к: pgp_sym_encrypt

F.37.3.8.8. s2k-count

Число итераций для алгоритма S2K. Оно должно быть не меньше 1024 и не больше 65011712.

По умолчанию: случайное значение между 65536 и 253952
Применим к: pgp_sym_encrypt, только с s2k-mode=3

F.37.3.8.9. s2k-digest-algo

Выбирает алгоритм хеширования, который будет использоваться для вычисления S2K.

Значения: md5, sha1
По умолчанию: sha1
Применим к: pgp_sym_encrypt

F.37.3.8.10. s2k-cipher-algo

Выбирает защитное преобразование, который будет использоваться для отдельного сеансового ключа.

Значения: bf, aes, aes128, aes192, aes256
По умолчанию: используется cipher-algo
Применим к: pgp_sym_encrypt

F.37.3.8.11. unicode-mode

Определяет, преобразовывать ли текстовые данные из внутренней кодировки базы данных в UTF-8 и обратно. Если кодировка базы уже UTF-8, перекодировка не производится, но сообщение помечается как UTF-8. Без данного параметра этого не происходит.

Значения: 0, 1
По умолчанию: 0
Применим к: pgp_sym_encrypt, pgp_pub_encrypt

F.37.3.9. Формирование ключей PGP с применением GnuPG

Формирование нового ключа:

```
gpg --gen-key
```

Предпочитаемый тип ключей: «DSA and Elgamal».

Для защитного преобразования RSA вы должны создать главный ключ либо DSA, либо RSA только для подписания, а затем добавить подключ для защитного преобразования, выполнив команду `gpg --edit-key`.

Просмотр списка ключей:

```
gpg --list-secret-keys
```

Экспорт открытого ключа в формате «ASCII Armor»:

```
gpg -a --export KEYID > public.key
```

Экспорт закрытого ключа в формате «ASCII Armor»:

```
gpg -a --export-secret-keys KEYID > secret.key
```

Прежде чем передавать эти ключи функциям PGP, вы должны применить функцию `dearmor()` к этим ключам. Либо, если вы можете обработать двоичные данные, уберите `-a` из команды.

Дополнительную информацию вы можете получить в руководстве `man gpg`, *The GNU Privacy Handbook* (Руководство GNU по обеспечению конфиденциальности) и другой документации на сайте *https://www.gnupg.org/*.

F.37.3.10. Ограничения кода PGP

- Не поддерживается подписывание. Это также означает, что принадлежность подключа защитного преобразования главному ключу не проверяется.
- Не поддерживается использование ключа защитного преобразования в качестве главного ключа. Так как подобная практика обычно не приветствуется, это не должно быть проблемой.
- Нет поддержки нескольких подключей. Это может представляться проблемой, так как такие ключи не редкость. С другой стороны, вы всё равно не должны использовать обычные ключи GPG/PGP с `pgcrypto`, а должны создать новые, учитывая, что это другой сценарий использования.

F.37.4. Низкоуровневые функции защитного преобразования

Эти функции выполняют только защитное преобразование данных; они не предоставляют расширенные возможности защитного преобразования PGP. Таким образом, с ними связаны следующие проблемы:

1. Они используют ключ пользователя непосредственно в качестве ключа защитного преобразования.
2. Они не обеспечивают проверку целостности, которая должна выявлять модификацию защищённых данных.
3. Они рассчитаны на то, что пользователи будут управлять всеми параметрами защитного преобразования самостоятельно, даже вектором инициализации.
4. Они не рассчитаны на текст.

Поэтому с появлением поддержки защитного преобразования PGP использовать низкоуровневые функции защитного преобразования не рекомендуется.

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

Эти функции производят прямое и обратное защитное преобразование данных, применяя метод, заданный параметром *`type`*. Строка *`type`* имеет следующий формат:

`алгоритм` `[ -` *`режим`* `] [ /pad:` *`дозаполнение`* `]`

где допустимый *`алгоритм`*:

- `bf` — Blowfish
- `aes` — AES (Rijndael-128, -192 или -256)

допустимый *`режим`*:

- `cbc` — следующий блок зависит от предыдущего (по умолчанию)
- `ecb` — каждый блок защищается отдельно (только для тестирования)

и допустимое *дозаполнение*:

- `pkcs` — данные могут быть любой длины (по умолчанию)
- `none` — размер данных должен быть кратен размеру блока защитного преобразования

Так что, например, эти вызовы равнозначны:

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

Для функций `encrypt_iv` и `decrypt_iv` параметр *`iv`* задаёт начальное значение для режима CBC; для ECB он игнорируется. Оно обрезается или дополняется нулями, если его размер не равен ровно размеру блока. В функциях без этого параметра оно по умолчанию заполняется нулями.

F.37.5. Функции получения случайных данных

```
gen_random_bytes(count integer) returns bytea
```

Возвращает защищённые стойкие случайные байты в количестве *`count`*. За один вызов можно получить максимум 1024 байт. Это ограничение предотвращает исчерпание пула энтропии.

```
gen_random_uuid() returns uuid
```

Возвращает UUID версии 4 (случайный). (Эта функция в данном модуле считается устаревшей, она вызывает одноимённую функцию ядра.)

F.37.6. Примечания

F.37.6.1. Конфигурирование

Модуль `pgcrypto` настраивается согласно установкам, полученным в главном скрипте `configure` Postgres Pro. На его конфигурацию влияют аргументы `--with-zlib` и `--with-ssl=openssl`.

При компиляции с zlib функции защитного преобразования PGP могут сжимать данные перед применением преобразования.

Для работы `pgcrypto` требуется OpenSSL. Без этой библиотеки модуль не будет собран и установлен.

Чтобы использовать устаревшие алгоритмы защитного преобразования, такие как DES или Blowfish, при компиляции с OpenSSL версии 3.0.0 и выше, нужно активировать поставщиков этих алгоритмов в файле конфигурации `openssl.cnf`.

F.37.6.2. Обработка NULL

Как и положено по стандарту SQL, все эти функции возвращают NULL, если один из аргументов — NULL. Это может угрожать безопасности при неаккуратном использовании.

F.37.6.3. Ограничения безопасности

Все функции `pgcrypto` выполняются внутри сервера баз данных. Это означает, что все данные и пароли передаются между функциями `pgcrypto` и клиентскими приложениями открытым текстом. Поэтому вы должны:

1. Подключаться локально или использовать подключения SSL.
2. Доверять и системе, и администратору баз данных.

Если это невозможно, лучше произвести защитное преобразование в клиентском приложении.

Эта реализация не противостоит *атакам по сторонним каналам*. Например, время, требующееся для выполнения функции обратного защитного преобразования `pgcrypto`, будет разным для разного защищённого текста заданного размера.

F.37.7. Автор

Марко Крин `<markokr@gmail.com>`

Модуль `pgcrypto` заимствует код из следующих источников:

Алгоритм	**Автор**	**Источник исходного кода**
Защитное преобразование DES	Дэвид Буррен и другие	FreeBSD, libcrypt
Хеширование MD5	Пол-Хеннинг Камп	FreeBSD, libcrypt
Защитное преобразование Blowfish	Solar Designer	www.openwall.com

F.38. pg_freespacemap — просмотр карты свободного пространства

Модуль `pg_freespacemap` предоставляет средства для исследования карты свободного пространства (FSM). В нём реализована функция `pg_freespace`, точнее, две перегруженных функции. Эти функции показывают значение, записанное в карте свободного пространства для данной страницы, либо для всех страниц отношения.

По умолчанию его использование разрешено только суперпользователям и ролям с правами роли `pg_stat_scan_tables`. Дать доступ другим можно с помощью `GRANT`.

F.38.1. Функции

`pg_freespace(rel regclass IN, blkno bigint IN) returns int2`

Возвращает объём свободного пространства на странице для отношения, заданного параметром `blkno`, согласно FSM.

`pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)`

Выдаёт объём свободного пространства на каждой странице отношения, согласно FSM. Возвращается набор кортежей `(blkno bigint, avail int2)`, по одному кортежу для каждой страницы в отношении.

Значения, хранимые в карте свободного пространства, не являются точными. Они округляются до 1/256 величины `BLCKSZ` (до 32 байт со значением `BLCKSZ` по умолчанию) и не поддерживаются в актуальном состоянии при каждом добавлении и изменении кортежей.

Для индексов отслеживаются полностью неиспользованные страницы, а не свободное пространство в страницах. Таким образом, эти значения отражают только то, что страница используется или свободна.

F.38.2. Пример вывода

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)
```

```
postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.38.3. Автор

Исходную версию разработал Марк Кирквуд `<markir@paradise.net.nz>`. Для версии 8.4 с новой реализацией FSM код адаптировал Хейкки Линнакангас `<heikki@enterprisedb.com>`

F.39. pg_proaudit — регистрация различных событий, связанных с безопасностью

Расширение `pg_proaudit` позволяет регистрировать различные события, связанные с безопасностью.

`pg_proaudit` работает параллельно со стандартными средствами протоколирования PostgreSQL (сборщиком сообщений) и не зависит от них. Журнал событий безопасности расширения `pg_proaudit` сохраняется отдельно от журнала работы сервера. При запуске Postgres Pro Standard расширение `pg_proaudit` запускает специальный фоновый процесс, который записывает в журнал информацию о событиях безопасности.

Правила регистрации событий сохраняются в файле конфигурации `pg_proaudit.conf`, располагающемся в каталоге данных кластера (`PGDATA`). Он представляет собой текстовый файл, который можно редактировать непосредственно, используя инструменты операционной системы. Для изменения его содержимого средствами SQL предусмотрены несколько функций `pg_proaudit`. В представлении `pg_proaudit_settings` можно посмотреть текущие правила `pg_proaudit`, даже если они ещё не были сохранены в файл `pg_proaudit.conf`.

Все фиксируемые события разделяются на следующие классы:

- команды DDL для создания, изменения и удаления объектов СУБД (баз данных, табличных пространств, схем, таблиц, представлений, последовательностей, языков, функций)
- команды управления доступом к объектам баз данных (`GRANT`, `REVOKE`)
- команды DML для обращения к объектам баз данных (`INSERT`, `UPDATE`, `DELETE`, `SELECT`, `TRUNCATE` для таблиц/представлений, `EXECUTE` для функций)
- события аутентификации/отключения от баз данных
- все команды, выполняемые определённым пользователем

События безопасности могут записываться как в централизованное средство журналирования операционной системы (syslog), так и в обычные файлы. Оба варианта могут использоваться одновременно. Все записи `pg_proaudit` в syslog имеют метку `AUDIT`, что позволяет однозначно их определять. Файлы журнала событий могут записываться в форматах CEF (общий формат записи событий, Common Event Format) и CSV (значения, разделённые запятыми, Comma Separated Values). Каждое событие записывается в отдельную строку, в которой выделяются следующие поля:

- дата и время события
- имя пользователя (`current_user`)
- имя базы данных
- идентификатор серверного процесса (PID)
- уровень важности: `INFO` или `ERROR`
- последовательный номер команды в сеансе
- номер подкоманды в сложных командах (`CREATE TABLE ... AS SELECT ...`)
- имя оператора
- тип объекта
- имя объекта
- результат выполнения оператора: `SUCCESS` (успех) или `FAILURE` (отказ)
- дополнительная информация, например сообщение об ошибке в случае результата `FAILURE` или параметры подключения для события `AUTHENTICATE`; незавершённые попытки аутентификации помечаются как [EOF. No credentials provided] (Конец файла. Учётные данные не предоставлены)
- текст SQL-команды
- параметры команды (например, для `PREPARE`)
- имя пользователя (`session_user`)
- уникальный идентификатор события (в формате *UUIDv7*)
- идентификатор транзакции (XID)
- виртуальный идентификатор транзакции (VXID)

Событие содержит информацию о значениях `session_user` и `current_user`, которые используются в рамках пользовательского сеанса. Поэтому можно идентифицировать пользователя, даже если он сменит идентификатор пользователя с помощью команды `SET ROLE`.

Значения XID и VXID могут быть нулевыми, если событие не относится к транзакции (например, событие `DISCONNECT`).

Вы можете определить каталог, в котором будут храниться файлы журналов безопасности и настроить ротацию этих файлов. В `pg_proaudit` предусмотрена возможность переключения на новый файл журнала либо после определённого интервала времени, либо по достижении определённого размера файла журнала. Это позволяет организовать процедуру очистки журналов безопасности.

Пользователь Postgres Pro с атрибутом `SUPERUSER` должен давать доступ к расширению `pg_proaudit` и журналам событий безопасности только пользователям, исполняющим роль администратора информационной безопасности.

F.39.1. Установка расширения `pg_proaudit`

Расширение `pg_proaudit` включено в состав Postgres Pro Standard как стандартное расширение. Чтобы задействовать `pg_proaudit`, выполните следующие действия:

1. Добавьте `pg_proaudit` в переменную `shared_preload_libraries` в файле `postgresql.conf`:

   ```
   shared_preload_libraries = 'pg_proaudit'
   ```
2. Перезагрузите сервер базы данных для применения изменений. Чтобы убедиться, что библиотека `pg_proaudit` установлена правильно, выполните следующую команду:

   ```
   SHOW shared_preload_libraries;
   ```
3. В каждой базе данных, для которой должны регистрироваться события безопасности, создайте расширение `pg_proaudit`, выполнив следующий запрос:

   ```
   CREATE EXTENSION pg_proaudit;
   ```

   Расширение `pg_proaudit` добавляет несколько функций для управления файлом `pg_proaudit.conf`, представление `pg_proaudit_settings`, показывающее текущие правила `pg_proaudit`, и событийные триггеры.

F.39.2. Удаление расширения `pg_proaudit`

Чтобы корректно удалить `pg_proaudit`, выполните следующие шаги:

1. Удалите расширение `pg_proaudit`, выполнив следующий запрос:

   ```
   DROP EXTENSION pg_proaudit;
   ```
2. Удалите `pg_proaudit` из переменной `shared_preload_libraries` в файле `postgresql.conf`.

   Пропустите этот шаг, если в кластере несколько баз данных и удалить расширение необходимо только для одной из них. В данном случае перед удалением расширения рекомендуется удалить правила регистрации событий, относящиеся к соответствующей базе данных.

F.39.3. Функции для настройки записи событий безопасности в журнал

Для настройки регистрации событий в `pg_proaudit` предусмотрен SQL-интерфейс, состоящий из различных функций, а также представление `pg_proaudit_settings`.

```
pg_proaudit_set_rule(db_name text, event_type text, object_type text, object_name text,
 role_name text, comment text)
```

Создаёт правило регистрации событий с заданными параметрами. Регистрация начинается сразу после завершения функции `pg_proaudit_set_rule()`, но файл `pg_proaudit.conf` не изменяется. Для сохранения изменений в файле `pg_proaudit.conf` вызовите функцию `pg_proaudit_save()`.

Аргументы:

- *`db_name`* — имя базы данных, для которой задаётся правило регистрации событий. Указание NULL или пустой строки означает, что регистрируются события для всех баз данных, в которых создано расширение `pg_proaudit`. При указании значения current_database() события регистрируются для текущей базы данных.
- *`event_type`* — тип событий, которые должны регистрироваться, включая имена операторов SQL, а также названия событий `AUTHENTICATE` и `DISCONNECT`. Если указано значение `ALL`, NULL или пустая строка, включается регистрация всех возможных событий для объектов заданного типа. Например, для объектов типа `TABLE` ключевое слово `ALL` включает регистрацию команд `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `COPY`, а также `CREATE`, `ALTER` и `DROP`. Также можно настроить регистрацию классов событий, используя следующие значения: `ALL_DDL`, `ALL_DDL_NONTEMP`, `ALL_DML`, `ALL_DML_NONTEMP`, `ALL_MOD`, `ALL_PROC` и `ALL_ROLE`. Полный список допустимых значений параметра *`event_type`* приведён в Подразделе F.39.5.
- *`object_type`* — тип объектов, для которых должны регистрироваться события безопасности. Если указано значение `ALL`, NULL или пустая строка, включается регистрация событий для всех типов объектов. Например, для регистрации всех попыток доступа к сторонним таблицам укажите тип объекта `FOREIGN TABLE` для события `SELECT`. Если в *`event_type`* передаётся `AUTHENTICATE`, `DISCONNECT`, `SET` или `RESET`, в *`object_type`* должен передаваться NULL. Для всех событий, относящихся к действиям с пользователями, например `CREATE USER` или `DROP USER`, в качестве типа объекта нужно передавать `ROLE`. Поддерживаются следующие типы объектов: `COMPOSITE TYPE`, `DATABASE`, `EVENT TRIGGER`, `FUNCTION`, `INDEX`, `PREPARED STATEMENT`, `ROLE`, `SEQUENCE`, `SCHEMA`, `TABLE`, `FOREIGN TABLE`, `TOAST TABLE`, `TABLESPACE`, `VIEW`, `MATERIALIZED VIEW`, `CATALOG RELATION` и `CATALOG FUNCTION`.

Примечание

Чтобы регистрировать события для объектов системного каталога, необходимо включить параметр конфигурации pg_proaudit.log_catalog_access. В противном случае эти события не будут зарегистрированы, даже если в правиле указан тип объекта `CATALOG RELATION`, `CATALOG FUNCTION` или `ALL`.

- *`object_name`* — имя объекта, для которого задаётся правило регистрации событий. При указании NULL или пустой строки регистрация событий включается для всех имён объектов.
- *`role_name`* — имя роли, для которой задаётся правило регистрации событий. Если указано, позволяет регистрировать действия, совершённые пользователем с правами этой роли. Это означает, что как минимум один из атрибутов пользовательского сеанса, `session_user` или `current_user`, должен либо совпадать с `role_name`, либо прямо или косвенно являться членом указанной роли. При указании пустой строки или значения NULL включается регистрация событий для всех имён ролей, а при указании значения current_role() события регистрируются для текущей роли.
- *`comment`* — комментарий, описывающий созданное правило регистрации событий. Этот аргумент не влияет на выполнение правила и не отражается в журнале.

`pg_proaudit_remove_rule(db_name text, event_type text, object_type text, object_name text, role_name text)`

Удаляет конкретное правило регистрации событий с заданными параметрами. Для сохранения изменений в файле конфигурации `pg_proaudit.conf` вызовите функцию `pg_proaudit_save()`.

Примечание

Если правило, которое вы хотите удалить, настроено для регистрации типа событий `DISCONNECT`, то события отключения могут продолжать регистрироваться после удаления правила. За дополнительными сведениями обратитесь к Подразделу F.39.3.1.

Аргументы:

- *`db_name`* — имя базы данных, для которой удаляется правило регистрации событий.
- *`event_type`* — тип событий, для которого удаляется правило регистрации. Список возможных значений приведён в Подразделе F.39.5.
- *`object_type`* — тип объекта, для которого удаляется правило регистрации событий.
- *`object_name`* — имя объекта, для которого удаляется правило регистрации событий.
- *`role_name`* — имя роли, для которой удаляется правило регистрации событий.

`pg_proaudit_show()`

Возвращает список регистрируемых событий в виде таблицы. Эта функция применяется в представлении `pg_proaudit_settings`.

`pg_proaudit_reload()`

Считывает конфигурацию регистрации из файла `pg_proaudit.conf`. Вы должны вызывать эту функцию, если файл `pg_proaudit.conf` изменялся непосредственно средствами операционной системы.

`pg_proaudit_reset()`

Удаляет все правила регистрации событий. Для сохранения изменённой конфигурации в файле `pg_proaudit.conf` вызовите функцию `pg_proaudit_save()`.

`pg_proaudit_save()`

Сохраняет правила регистрации событий из памяти в файле `pg_proaudit.conf`. Файл `pg_proaudit.conf` размещается в каталоге данных кластера (`PGDATA`). Изменить расположение файла `pg_proaudit.conf` нельзя.

F.39.3.1. Обработка событий отключения от базы данных

Когда настроено правило для регистрации типа событий `DISCONNECT`, и пользователь аутентифицируется в базе данных, то расширение `pg_proaudit` проверяет, удовлетворяет ли правилу соответствующее событие отключения и нужно ли его регистрировать.

Если такое событие должно быть зарегистрировано, то удаление правила не оказывает влияния на него. Событие будет зарегистрировано вне зависимости от того, существует ли правило на момент отключения.

Рассмотрим следующий пример:

1. Администратор настраивает следующее правило регистрации событий:

   ```
   SELECT pg_proaudit_set_rule ('postgres', 'DISCONNECT', null, null, null, 'Any
    disconnect from the postgres DB');
   ```

2. Пользователь подключается к базе данных `postgres` и аутентифицируется в ней.

   Поскольку правило действует, позже будет зарегистрировано соответствующее событие отключения.

3. Администратор удаляет правило регистрации событий, при этом пользователь всё ещё подключён к базе данных:

   ```
   SELECT pg_proaudit_remove_rule('postgres', 'DISCONNECT', null, null, null);
   ```

4. Пользователь отключается от базы данных.

   Соответствующее событие отключения регистрируется, несмотря на то, что правило было удалено.

F.39.3.2. Обработка имён объектов

Строковые представления имён базы данных, пользователя, роли и других объектов сравниваются независимо от регистра, даже если имя объекта в базе данных заключено в кавычки, и хранятся в нижнем регистре.

Строки, представляющие имена объектов, необходимо передавать функциям `pg_proaudit_set_rule` и `pg_proaudit_remove_rule` как обычные строки: одинарную кавычку необходимо удвоить без экранирования, а другие символы передаются как есть, без экранирования или переноса.

Например, правила для таблиц `table1` и `"TaBlE1"` будут одинаковыми, а их имена можно передавать символами как верхнего, так и нижнего регистра, как и имена ролей. Перед созданием правила для баз данных pg_proaudit проверяет, действительно ли существует указанная база данных, и эта проверка чувствительна к регистру.

В следующих примерах показано создание «странных» имён объектов с указанием корректных правил для них.

```
CREATE TABLE "TAbLe''123"();
SELECT pg_proaudit_set_rule(current_database(), null, null, 'public.TabLe''''123',
 null, 'tst cmnt');
```

```
CREATE TABLE "taBlE""123"();
SELECT pg_proaudit_set_rule(null, null, null, 'public.taBlE"123', null);
```

```
CREATE SCHEMA "(BIG!) Schema";
CREATE TABLE "(BIG!) Schema"."My Tab.lE"();
SELECT pg_proaudit_set_rule(null, null, null, '(BIG!) Schema.My Tab.lE', null);
```

```
CREATE ROLE "Some '@#$%' person" WITH LOGIN;
SELECT pg_proaudit_set_rule(null, 'authenticate', null, null, 'Some ''@#$%'' person');
```

```
CREATE ROLE "Some '@#$%' person" WITH SUPERUSER;
\c - "Some '@#$%' person"
SELECT pg_proaudit_set_rule(null, 'disconnect', null, null, current_user);
```

```
CREATE DATABASE " D B 1";
SELECT pg_proaudit_set_rule(' D B 1', 'authenticate', null, null, null);
```

```
CREATE DATABASE " D b 2";
\c " D b 2"
CREATE EXTENSION pg_proaudit;
SELECT pg_proaudit_set_rule(current_database(), 'disconnect', null, null, null);
```

F.39.4. Представление pg_proaudit_settings

В этом представлении показываются текущие правила `pg_proaudit`, даже если они ещё не были сохранены в файл `pg_proaudit.conf`. Представление `pg_proaudit_settings` образуют следующие столбцы:

- *`db_name`* (`text`) — имя базы данных, в которой регистрируются события.
- *`event_type`* (`text`) — тип регистрируемых событий.
- *`object_type`* (`text`) — тип объекта, для которого должны регистрироваться события безопасности.
- *`object_name`* (`text`) — имя объекта, для которого должны регистрироваться события.
- *`role_name`* (`text`) — роль, от имени которой должны выполняться регистрируемые действия.
- *`comment`* (`text`) — комментарий, описывающий созданное правило регистрации событий.

F.39.5. События безопасности

Можно настроить расширение `pg_proaudit` для регистрации событий безопасности определённых классов или конкретных событий, указав соответствующее значение в аргументе *`event_type`* функции pg_proaudit_set_rule().

Поддерживаются следующие классы событий безопасности:

- `ALL_DDL`: `CREATE`, `ALTER`, `DROP` для любого объекта базы данных, за исключением хранимых процедур и функций.
- `ALL_DDL_NONTEMP`: такой же класс, как `ALL_DDL`, но его область действия ограничена только теми объектами базы данных, которые не находятся во временных схемах `pg_temp_`*`nnn`*.
- `ALL_DML`: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` для любых типов таблиц; `EXECUTE` для функций и хранимых процедур.
- `ALL_DML_NONTEMP`: такой же класс, как `ALL_DML`, но его область действия ограничена только теми объектами базы данных, которые не находятся во временных схемах `pg_temp_`*`nnn`*.
- `ALL_MOD`: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` для любых типов таблиц.
- `ALL_PROC`: `CREATE`, `ALTER`, `DROP` для любых функций и хранимых процедур.
- `ALL_ROLE`: `CREATE`, `ALTER`, `DROP` в отношении `USER`, `ROLE`, `GROUP`, `PROFILE`, а также выполнение команды `GRANT`.

Поддерживаются следующие события безопасности:

- AUTHENTICATE
- DISCONNECT
- ALTER AGGREGATE
- ALTER COLLATION
- ALTER CONVERSION
- ALTER DATABASE
- ALTER DEFAULT PRIVILEGES
- ALTER DOMAIN
- ALTER EVENT TRIGGER
- ALTER EXTENSION
- ALTER FOREIGN DATA WRAPPER
- ALTER FOREIGN TABLE
- ALTER FUNCTION
- ALTER INDEX
- ALTER LANGUAGE
- ALTER LARGE OBJECT
- ALTER MATERIALIZED VIEW
- ALTER OPERATOR
- ALTER OPERATOR CLASS
- ALTER OPERATOR FAMILY
- ALTER POLICY
- ALTER PROFILE
- ALTER ROLE, ALTER USER, ALTER GROUP
- ALTER RULE
- ALTER SCHEMA
- ALTER SEQUENCE
- ALTER SERVER
- ALTER SYSTEM
- ALTER TABLE
- ALTER TABLESPACE
- ALTER TEXT SEARCH CONFIGURATION
- ALTER TEXT SEARCH DICTIONARY
- ALTER TEXT SEARCH PARSER
- ALTER TEXT SEARCH TEMPLATE
- ALTER TRIGGER
- ALTER TYPE
- ALTER USER MAPPING
- ALTER VIEW
- CLUSTER
- COMMENT
- COPY

- CREATE ACCESS METHOD
- CREATE AGGREGATE
- CREATE CAST
- CREATE COLLATION
- CREATE CONVERSION
- CREATE DATABASE
- CREATE DOMAIN
- CREATE EVENT TRIGGER
- CREATE EXTENSION
- CREATE FOREIGN DATA WRAPPER
- CREATE FOREIGN TABLE
- CREATE FUNCTION
- CREATE INDEX
- CREATE LANGUAGE
- CREATE MATERIALIZED VIEW
- CREATE OPERATOR
- CREATE OPERATOR CLASS
- CREATE OPERATOR FAMILY
- CREATE POLICY
- CREATE PROFILE
- CREATE ROLE, CREATE USER, CREATE GROUP
- CREATE RULE
- CREATE SCHEMA
- CREATE SEQUENCE
- CREATE SERVER
- CREATE TABLE, CREATE TABLE AS, SELECT INTO
- CREATE TABLESPACE
- CREATE TEXT SEARCH CONFIGURATION
- CREATE TEXT SEARCH DICTIONARY
- CREATE TEXT SEARCH PARSER
- CREATE TEXT SEARCH TEMPLATE
- CREATE TRANSFORM
- CREATE TRIGGER
- CREATE TYPE
- CREATE USER MAPPING
- CREATE VIEW
- DEALLOCATE
- DELETE
- DO
- DROP ACCESS METHOD
- DROP AGGREGATE
- DROP CAST
- DROP COLLATION
- DROP CONVERSION
- DROP DATABASE
- DROP DOMAIN
- DROP EVENT TRIGGER
- DROP EXTENSION
- DROP FOREIGN DATA WRAPPER
- DROP FOREIGN TABLE
- DROP FUNCTION
- DROP INDEX
- DROP LANGUAGE
- DROP MATERIALIZED VIEW
- DROP OPERATOR
- DROP OPERATOR CLASS
- DROP OPERATOR FAMILY
- DROP OWNED

- DROP POLICY
- DROP PROFILE
- DROP ROLE, DROP USER, DROP GROUP
- DROP RULE
- DROP SCHEMA
- DROP SEQUENCE
- DROP SERVER
- DROP TABLE
- DROP TABLESPACE
- DROP TEXT SEARCH CONFIGURATION
- DROP TEXT SEARCH DICTIONARY
- DROP TEXT SEARCH PARSER
- DROP TEXT SEARCH TEMPLATE
- DROP TRANSFORM
- DROP TRIGGER
- DROP TYPE
- DROP USER MAPPING
- DROP VIEW
- EXECUTE
- GRANT
- INSERT
- PREPARE
- REASSIGN OWNED
- REFRESH MATERIALIZED VIEW
- REINDEX
- RESET
- REVOKE
- SECURITY LABEL
- SELECT
- SET
- UPDATE
- TRUNCATE TABLE

F.39.6. Параметры конфигурации журнала событий безопасности

Расширение `pg_proaudit` предоставляет несколько параметров конфигурации для управления файлами журналов событий безопасности. Эти параметры можно задать непосредственно в файле конфигурации `postgresql.conf` или посредством команды `ALTER SYSTEM`. Чтобы изменения вступили в силу, вызовите функцию `pg_reload_conf()` или перезапустите сервер баз данных. Для дополнительной настройки можно использовать параметры конфигурации syslog_ident и syslog_facility.

`pg_proaudit.log_destination` (`string`)

Определяет метод фиксирования событий безопасности. Возможные значения:

- `csvlog` — записывать события безопасности в файл CSV.
- `ceflog` — записывать события безопасности в файл CEF. CEF — это открытый текстовый формат, предназначенный для записей журнала. Он содержит стандартный заголовок и переменное расширение, оформленное в виде пар ключ-значение.
- `syslog` — передавать события безопасности в syslog.

Данный параметр может содержать одно или несколько этих значений, разделённых запятыми.

По умолчанию: `csvlog`

`pg_proaudit.log_catalog_access` (`boolean`)

Определяет, будут ли регистрироваться обращения к объектам системного каталога в схеме `pg_catalog`.

Со значением `off` такие события не регистрируются, даже если есть правила для типов объектов `CATALOG RELATION`, `CATALOG FUNCTION` или `ALL`.

Не рекомендуется задавать для этого параметра значение `on`, если вы не планируете регистрировать события для объектов системного каталога. В противном случае это может влиять на производительность Postgres Pro, даже если правила для упомянутых типов объектов отсутствуют.

По умолчанию: `off`

`pg_proaudit.log_command_text` (`boolean`)

Определяет, будет ли записываться в журнал текст команд SQL, вызывающих события безопасности.

По умолчанию включено.

`pg_proaudit.log_directory` (`string`)

Задаёт путь к каталогу для хранения файлов журналов в форматах CSV и CEF. Это может быть абсолютный путь или путь, заданный относительно каталога данных (`PGDATA`). Этот параметр используется, если для параметра pg_proaudit.log_destination указано значение `csvlog` или `ceflog`.

По умолчанию: `pg_proaudit`

`pg_proaudit.log_filename` (`string`)

Определяет шаблон имён для файлов журналов событий безопасности. Этот шаблон может содержать спецпоследовательности с %, похожие на те, которые перечислены в спецификации `strftime` стандарта Open Group (*http://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html*). Этот параметр используется, если для параметра pg_proaudit.log_destination указано значение `csvlog` или `ceflog`.

По умолчанию: `postgresql-%Y-%m-%d_%H%M%S.log`

`pg_proaudit.log_rotation_size` (`integer`)

Задаёт максимальный размер файла журнала, в килобайтах. По достижении этого размера `pg_proaudit` создаёт для записи событий безопасности новый файл. Если для параметра установлено значение `0`, создание новых файлов в зависимости от размера текущего файла отключается. Этот параметр используется, если для параметра pg_proaudit.log_destination указано значение `csvlog` или `ceflog`.

По умолчанию: 10 МБ

`pg_proaudit.log_rotation_age` (`integer`)

Задаёт максимальное время жизни файла журнала, в минутах. По истечении этого времени `pg_proaudit` создаёт новый файл для записи событий безопасности. Если для параметра установлено значение `0`, создание новых файлов по истечении времени отключается. Этот параметр используется, если для параметра pg_proaudit.log_destination указано значение `csvlog` или `ceflog`.

По умолчанию: 1 день

`pg_proaudit.log_truncate_on_rotation` (`boolean`)

Определяет, должны ли усекаться файлы журналов при переключении записи на уже существующий файл журнала. Если для параметра установлено значение `off`, `pg_proaudit` продолжает запись в конец файла. Этот параметр используется, если для параметра pg_proaudit.log_destination указано значение `csvlog` или `ceflog`.

По умолчанию: `off`

`pg_proaudit.max_rules_count` (`integer`)

Определяет максимально допустимое количество правил. Чтобы изменение параметра вступило в силу, необходимо перезагрузить сервер базы данных.

По умолчанию: 500

F.39.7. Просмотр журнала событий безопасности

Файлы журналов событий безопасности представляют собой текстовые файлы, которые можно просмотреть средствами операционной системы. Чтобы прочитать файлы журналов на уровне SQL, вы можете применить расширение `file_fdw` — обёртку сторонних данных для обращения к файлам на сервере. Для этого выполните следующие действия:

1. Установите `file_fdw` и создайте определение стороннего сервера:

   ```
   CREATE EXTENSION file_fdw;
   CREATE SERVER pg_proauditlog FOREIGN DATA WRAPPER file_fdw;
   ```

2. Создайте стороннюю таблицу, определив её столбцы и указав абсолютный путь к файлу журнала. Фактический путь определяется параметрами `pg_proaudit.log_directory` и `pg_proaudit.log_filename`.

   ```
   CREATE FOREIGN TABLE pg_proaudit_log
         ( log_time timestamp(3) with time zone,
           current_usr_name text,
           database_name text,
           session_pid text,
           error_severity text,
           session_line_num bigint,
           session_line_subcommand_num bigint,
           event_type text,
           object_type text,
           object_name text,
           status text,
           error_message text,
           query_text text,
           query_args text,
           session_usr_name text,
           uuid text,
           xid text,
           vxid text )

   SERVER pg_proauditlog
   OPTIONS (filename 'absolute_file_path_to_log_file.csv', FORMAT 'csv' );
   ```

Убедитесь в том, что параметр `pg_proaudit.log_destination` содержит значение `csvlog`, включающее запись событий безопасности в файлы CSV.

F.39.8. Примеры

В качестве примера давайте настроим регистрацию следующих событий:

- аутентификация/отключение от базы данных `postgres`
- все действия пользователя, если как минимум один из атрибутов пользовательского сеанса, `session_user` или `current_user`, либо явно установлен в значение `postgres`, либо прямо или косвенно является членом этой роли
- создание, изменение и удаление любых таблиц
- все операции с таблицей `app_table`, принадлежащей схеме `public`

Все события должны записываться в файлы в формате CSV и храниться неделю. Для обращения к журналу событий необходимо организовать доступ на уровне SQL. Для решения этой задачи выполните следующее:

Воспользовавшись psql, убедитесь, что предварительная подготовка расширения `pg_proaudit` в базе данных `postgres` выполнена:

```
SHOW shared_preload_libraries;
 shared_preload_libraries
--------------------------
 pg_proaudit

\dx pg_proaudit
                   List of installed extensions
    Name     | Version | Schema |           Description
-------------+---------+--------+----------------------------------
 pg_proaudit | 2.0     | public | provides auditing functionality
```

Добавьте следующие строки в файл конфигурации `postgresql.conf`:

```
pg_proaudit.log_destination = 'csvlog'
pg_proaudit.log_directory = 'audit'
pg_proaudit.log_filename = 'audit-%u.csv'
pg_proaudit.log_rotation_age = 1440
pg_proaudit.log_rotation_size = 0
pg_proaudit.log_truncate_on_rotation = on
pg_proaudit.log_command_text = on
```

Чтобы изменения вступили в силу, выполните:

```
SELECT pg_reload_conf();
```

Проверьте, были ли установлены желаемые значения следующих параметров:

```
SHOW pg_proaudit.log_destination;
SHOW pg_proaudit.log_directory;
SHOW pg_proaudit.log_filename;
SHOW pg_proaudit.log_rotation_age;
SHOW pg_proaudit.log_rotation_size;
SHOW pg_proaudit.log_truncate_on_rotation;
SHOW pg_proaudit.log_command_text;
```

Предположим, что ваша переменная окружения `PGDATA` указывает на каталог данных кластера. Так как в `pg_proaudit.log_directory` задан относительный путь, файлы журналов будут находиться в каталоге `$PGDATA/audit`. Создадим пустые файлы для семи дней недели и сделаем их доступными только для владельца:

```
touch $PGDATA/audit/audit-1.csv
touch $PGDATA/audit/audit-2.csv
touch $PGDATA/audit/audit-3.csv
touch $PGDATA/audit/audit-4.csv
touch $PGDATA/audit/audit-5.csv
touch $PGDATA/audit/audit-6.csv
touch $PGDATA/audit/audit-7.csv
chmod 600 $PGDATA/audit/audit-*.csv
```

Создайте таблицу для чтения записей журнала:

```
CREATE TABLE pg_proaudit_log (
  log_time timestamp(3) with time zone,
  current_usr_name text,
```

```
    database_name text,
    session_pid text,
    error_severity text,
    session_line_num bigint,
    session_line_subcommand_num bigint,
    event_type text,
    object_type text,
    object_name text,
    status text,
    error_message text,
    query_text text,
    query_args text,
    session_usr_name text,
    uuid text,
    xid text,
    vxid text
  );
```

Установите расширение `file_fdw` и создайте сторонний сервер:

```
CREATE EXTENSION file_fdw;
CREATE SERVER pg_proauditlog FOREIGN DATA WRAPPER file_fdw;
```

Теперь создадим для таблицы `pg_proaudit_log` семь дочерних сторонних таблиц, для каждого дня недели:

```
CREATE FOREIGN TABLE pg_proaudit_log_1 () INHERITS (pg_proaudit_log) SERVER
 pg_proauditlog
   OPTIONS (filename '/path_to_PGDATA/audit/audit-1.csv',  FORMAT 'csv');
CREATE FOREIGN TABLE pg_proaudit_log_2 () INHERITS (pg_proaudit_log) SERVER
 pg_proauditlog
   OPTIONS (filename '/path_to_PGDATA/audit/audit-2.csv',  FORMAT 'csv');
CREATE FOREIGN TABLE pg_proaudit_log_3 () INHERITS (pg_proaudit_log) SERVER
 pg_proauditlog
   OPTIONS (filename '/path_to_PGDATA/audit/audit-3.csv',  FORMAT 'csv');
CREATE FOREIGN TABLE pg_proaudit_log_4 () INHERITS (pg_proaudit_log) SERVER
 pg_proauditlog
   OPTIONS (filename '/path_to_PGDATA/audit/audit-4.csv',  FORMAT 'csv');
CREATE FOREIGN TABLE pg_proaudit_log_5 () INHERITS (pg_proaudit_log) SERVER
 pg_proauditlog
   OPTIONS (filename '/path_to_PGDATA/audit/audit-5.csv',  FORMAT 'csv');
CREATE FOREIGN TABLE pg_proaudit_log_6 () INHERITS (pg_proaudit_log) SERVER
 pg_proauditlog
   OPTIONS (filename '/path_to_PGDATA/audit/audit-6.csv',  FORMAT 'csv');
CREATE FOREIGN TABLE pg_proaudit_log_7 () INHERITS (pg_proaudit_log) SERVER
 pg_proauditlog
   OPTIONS (filename '/path_to_PGDATA/audit/audit-7.csv',  FORMAT 'csv');
```

Чтобы настроить регистрацию интересующих событий безопасности, подключитесь к базе данных `postgres` и выполните следующие команды:

```
SELECT pg_proaudit_set_rule (current_database(), 'AUTHENTICATE', null, null, null, 'Any
 authentication in the current DB');
SELECT pg_proaudit_set_rule (current_database(), 'DISCONNECT', null, null, null, 'Any
 disconnect from the current DB');
SELECT pg_proaudit_set_rule (current_database(), 'ALL', 'TABLE', null, null, 'Any
 operations with any table in the current DB');
SELECT pg_proaudit_set_rule (current_database(), 'ALL', null, null, 'postgres', 'Any
 operation by "postgres" user in the current DB');
```

Создайте таблицу `app_table` и включите регистрацию всех операций с этой таблицей:

```
CREATE TABLE app_table (id int, name text);
SELECT pg_proaudit_set_rule (current_database(), 'ALL', null, 'public.app_table',
 null);
```

Проверьте, работает ли регистрация событий должным образом:

```
SELECT * FROM pg_proaudit_settings;
 db_name  |  event_type  | object_type |   object_name    | role_name |
      comment
----------+--------------+-------------+------------------+-----------
+-----------------------------------------------------
 postgres | authenticate | ALL         |                  |           | Any
 authentication in the current DB
 postgres | disconnect   | ALL         |                  |           | Any disconnect
 from the current DB
 postgres | ALL          | table       |                  |           | Any operations
 with any table in the current DB
 postgres | ALL          | ALL         |                  | postgres  | Any operation
 by "postgres" user in the current DB
 postgres | ALL          | ALL         | public.app_table |           |
(5 rows)
```

Сохраните эти правила регистрации событий в файле `pg_proaudit.conf`, чтобы они не были потеряны при перезапуске сервера:

```
SELECT pg_proaudit_save();
```

Выполним несколько запросов, обращающихся к таблице `app_table`:

```
INSERT INTO app_table VALUES (1, 'first');
SELECT * FROM app_table;
```

Проверьте записи в журнале, относящиеся к таблице `app_table`:

```
SELECT to_char(log_time, 'DD.MM.YY HH24:MI:SS') AS when, current_usr_name,
      session_pid, event_type, query_text, session_usr_name
FROM pg_proaudit_log
WHERE object_name = 'public.app_table';
       when        | current_usr_name | session_pid |  event_type  |
 query_text                  | session_usr_name
-------------------+------------------+-------------+--------------
+-------------------------------------------+------------------
 27.09.23 12:44:27 | postgres         | 2010        | CREATE TABLE | CREATE TABLE
 app_table (id int, name text); | postgres
 27.09.23 12:45:55 | postgres         | 2010        | INSERT       | INSERT INTO
 app_table VALUES (1, 'first');  | postgres
 27.09.23 12:46:00 | postgres         | 2010        | SELECT       | SELECT * FROM
 app_table;                     | postgres
(3 rows)
```

Мы настроили еженедельную ротацию файлов журналов с переключением файлов журналов каждый день. Это означает, что запросы к таблице `pg_proaudit_log` будут возвращать только события, произошедшие за последнюю неделю. Более старые события будут автоматически удаляться при ротации файлов. Для дополнительного ограничения доступа к определённым записям журнала вы можете создать отдельные представления на базе таблицы `pg_proaudit_log` и разрешить чтение этих представлений, используя встроенные механизмы управления доступом Postgres Pro.

F.40. pgpro_autopart — динамическое создание секций

Расширение pgpro_autopart позволяет создавать секции динамически, то есть производит автоматическое секционирование при добавлении или изменении данных в таблице. Самый быстрый способ создания секций — вручную или по расписанию с помощью планировщика. Однако для некоторых задач скорость добавления данных не имеет большого значения. Для таких задач pgpro_autopart реализует автоматическое секционирование с помощью триггеров для представления секционированной таблицы.

F.40.1. Установка

Расширение pgpro_autopart входит в состав Postgres Pro. Чтобы его задействовать, создайте расширение следующим запросом:

```
CREATE EXTENSION pgpro_autopart;
```

F.40.2. Использование

Расширение pgpro_autopart использует для автоматического секционирования функцию `ap_enable_automatic_partition_creation`.

Примечание

Обратите внимание, что расширение работает только с новыми таблицами. Его нельзя использовать для существующих секционированных таблиц, поскольку оно не может отслеживать секции, которые были созданы пользователями ранее.

Сначала функция добавляет к имени указанной таблицы префикс `real_`, а затем создаёт представление с первоначальным именем таблицы и триггеры `INSTEAD OF INSERT/UPDATE` для этого представления.

Примечание

Обратите внимание, что администратор баз данных должен вручную предоставлять права на представление, созданное функцией `ap_enable_automatic_partition_creation`. Ожидается, что это будут те же права, что и для родительской таблицы.

Эти триггеры работают следующим образом:

- `INSTEAD OF INSERT`: при попытке добавить запись в представление pgpro_autopart ищет секцию для этой записи. Если секция не найдена, она создаётся, после чего запись добавляется в секционированную таблицу.
- `INSTEAD OF UPDATE`: при попытке изменить запись в представлении pgpro_autopart ищет секцию для этой записи. Если секция не найдена, она создаётся, после чего запись изменяется в секционированной таблице.

Примечание

Оператор `INSERT` или `UPDATE` должен содержать значение ключа секционирования.

При использовании последовательности в качестве значения ключа необходимо явно задать это значение с помощью функции `nextval()`, поскольку опустить его нельзя. В противном случае операция `INSERT` или `UPDATE` завершится ошибкой.

В настоящее время поддерживаются только таблицы с секционированием по диапазону значения ключа (`BY RANGE`), которым должен быть один столбец типа `date`, `timestamp`, `timestamptz`, `smallint`, `int` или `bigint`.

При создании секций для ключа секционирования типа `date`/`timestamp` допустимыми значениями интервала являются `year`, `quarter`, `month` и `day`. То есть новая секция создаётся для каждого нового года, квартала, месяца или дня.

Имена новых секций следуют строго определённому формату: к имени секционированной таблицы добавляется суффикс, указывающий интервал, для которого создаётся секция. Например:

`day`

`real_t_day_2024_05_13` для таблицы `real_t_day`

`month`

`real_t_month_2024_05` для таблицы `real_t_month`

`quarter`

`real_t_quarter_2024_2` для таблицы `real_t_quarter`

`year`

`real_t_year_2025` для таблицы `real_t_year`

`int`

`real_t_int_120_130` для таблицы `real_t_int`

`bigint`

`"real_t_bigint_-60_-50"` для таблицы `real_t_bigint`

Триггеры `INSTEAD OF INSERT/UPDATE` проверяют наличие секции для *каждой* вставляемой или изменяемой записи, что может несколько замедлить работу.

Поскольку исходная секционированная таблица переименовывается, для продолжения работы непосредственно с этой таблицей следует добавлять префикс `real_`.

Важно

Помните, что максимальная длина имени таблицы в Postgres Pro — 63 байта. Когда расширение pgpro_autopart создаёт секцию, оно автоматически добавляет до 29 байт (префикс `real_`, два подчёркивания, символы границ) к имени таблицы для типа `int` и до 51 байта для ключа секционирования типа `bigint`. Ответственность за выбор соответствующих имён таблиц, чтобы избежать ошибок, лежит на пользователе.

F.40.3. Представление ap_tables_view

Таблицы с включённым автоматическим секционированием показаны в представлении `ap_tables_view`. Ниже приведён пример данных в этом представлении.

```
SELECT * FROM ap_tables_view;
 apt_relname | apt_relschema |                                    apt_mode
-------------+---------------
+------------------------------------------------------------------------------
 t_month1    | user_schema   | automatic partition creation with using triggers on VIEW
 (C-implementation)
```

```
  t_bigint1   | user_schema   | automatic partition creation with using triggers on VIEW
 (C-implementation)
(2 rows)
```

F.40.4. Функции

`ap_enable_automatic_partition_creation(a_relname text[, a_relschema text], a_interval text) returns void`

Эта функция предназначена для таблиц с ключом секционирования типа `date`, `timestamp` или `timestampz`. Она переименовывает секционированную таблицу `a_relname` (схемы `a_relschema`), добавляя префикс `real_`, а затем создаёт представление с тем же именем, что и у исходной таблицы, и добавляет для него триггеры `INSTEAD OF INSERT/UPDATE`. Эти триггеры при необходимости создают новую секцию, границы которой определяются интервалом, заданным в параметре *`a_interval`* (год, квартал, месяц, день).

`ap_enable_automatic_partition_creation(a_relname text[, a_relschema text], a_interval smallint, a_firstval smallint) returns void`
`ap_enable_automatic_partition_creation(a_relname text[, a_relschema text], a_interval int, a_firstval int) returns void`
`ap_enable_automatic_partition_creation(a_relname text[, a_relschema text], a_interval bigint, a_firstval bigint) returns void`

Эти функции используются для таблиц с ключом секционирования типа `smallint`, `int` или `bigint`. Они переименовывают секционированную таблицу `a_relname` (схемы `a_relschema`), добавляя префикс `real_`. Затем вызванная функция создаёт представление с тем же именем, что и у исходной таблицы, и триггеры `INSTEAD OF INSERT/UPDATE` для него. Эти триггеры при необходимости создают новую секцию, границы которой определяются исходным значением `a_firstval`, от которого отсчитываются интервалы, и значением *`a_interval`*, определяющим длину интервала.

`ap_disable_automatic_partition_creation(a_relname text[, a_relschema text]) returns void`

Функция удаляет триггеры, созданные функцией `ap_enable_automatic_partition_creation` для представления указанной секционированной таблицы `a_relname` (схемы `a_relschema`). Она также удаляет представление и переименовывает секционированную таблицу, убирая из её имени префикс `real_`.

F.40.5. Пример

В следующем примере показано использование расширения pgpro_autopart с ключом секционирования типа `bigint`.

Создайте расширение.

```
CREATE EXTENSION pgpro_autopart;
```

Создайте таблицу, секционированную с указанием `BY RANGE` с одностолбцовым ключом типа `bigint`.

```
CREATE TABLE t_bigint (b bigint, i int) PARTITION BY RANGE (b);
```

Используйте расширение, чтобы переименовать таблицу `t_bigint` в `real_t_bigint`, создать представление `t_bigint` для этой таблицы, а затем создать триггеры `INSTEAD OF INSERT/UPDATE`. Обратите внимание, что начальное значение для создания секций — 100, а секции создаются с интервалом 10 в обоих направлениях.

```
SELECT ap_enable_automatic_partition_creation('t_bigint', 10, 100);
 ap_enable_automatic_partition_creation
----------------------------------------
 
(1 row)
```

Добавьте две записи в таблицу. При добавлении первой записи будет автоматически создана секция `real_t_bigint_110_120`.

```
INSERT INTO t_bigint VALUES (111, 1);
NOTICE:  New partition "public"."real_t_bigint_110_120" created
INSERT 0 1
INSERT INTO t_bigint VALUES (114, 2);
INSERT 0 1
```

Измените у одной из записей поле ключа. При этом будет автоматически создана секция `real_t_bigint_-60_-50`.

```
UPDATE t_bigint  SET b = -55 WHERE b = 114 RETURNING *;
NOTICE:  New partition "public"."real_t_bigint_-60_-50" created
  b  | i
-----+---
 -55 | 2
(1 row)
UPDATE 1
```

Проверьте секции таблицы `real_t_bigint`. Их должно быть две.

```
SELECT
  c.oid::pg_catalog.regclass AS "name",
  pg_catalog.pg_get_expr(c.relpartbound, c.oid) AS "condition"
FROM
  pg_catalog.pg_class c,
  pg_catalog.pg_inherits i
WHERE
  c.oid = i.inhrelid AND i.inhparent = 'real_t_bigint'::regclass;
          name           |              condition
-------------------------+-------------------------------------
 real_t_bigint_110_120   | FOR VALUES FROM ('110') TO ('120')
 "real_t_bigint_-60_-50"   | FOR VALUES FROM ('-60') TO ('-50')
(2 rows)
```

Проверьте данные в таблице `real_t_bigint` и её секциях. В таблице должно быть две записи:

```
SELECT * FROM real_t_bigint;
  b  | i
-----+---
 -55 | 2
 111 | 1
(2 rows)
```

В секции `real_t_bigint_110_120` должна быть одна запись:

```
SELECT * FROM real_t_bigint_110_120;
  b  | i
-----+---
 111 | 1
(1 row)
```

Также должна быть одна запись в секции `real_t_bigint_-60_-50`.

```
SELECT * FROM "real_t_bigint_-60_-50";
  b  | i
-----+---
 -55 | 2
(1 row)
```

Отключите триггеры, удалите представление и переименуйте таблицу `real_t_bigint` в `t_bigint`.

```
SELECT ap_disable_automatic_partition_creation('t_bigint');
 ap_disable_automatic_partition_creation
-----------------------------------------

(1 row)
```

Удалите таблицу и расширение.

```
DROP TABLE t_bigint;
DROP TABLE
DROP EXTENSION pgpro_autopart;
DROP EXTENSION
```

F.41. pgpro_bindump — модуль протокола репликации для резервного копирования и восстановления

pgpro_bindump — это модуль, предоставляющий дополнительные команды для протокола репликации Postgres Pro, в частности для использования с утилитой *pg_probackup3*. pgpro_bindump имеет следующие преимущества:

- Упрощённая связь с сервером. Для подключения к серверу баз данных pgpro_bindump не требует протокола SSH и связанных с ним настроек.
- Интегрированная передача WAL. Файлы WAL передаются вместе с файлами данных, что снижает риск потери информации при резервном копировании.
- Оптимизированное хранение. Файлы копируются не в исходном виде, а архивируются в собственном формате pg_probackup3 — `CBOR` (RFC 8949 Concise Binary Object Representation, Сжатое представление двоичных объектов), что повышает эффективность хранения.

F.41.1. Архитектура

Для обработки подключений от внешних приложений создаётся выделенный обслуживающий процесс. Для отправки запросов к этому процессу можно использовать утилиту pg_probackup3 или собственное приложение.

При необходимости можно устанавливать несколько соединений для одного приложения.

Поскольку модуль pgpro_bindump имеет прямой доступ к файлам базы данных, дополнительных разрешений не требуется.

Приложение pgpro_bindump использует библиотеку *libpgprobackup* вместе со встроенным модулем кодирования для передачи данных в приложение.

Плагин `walsender_plugin_libraries` упрощает процесс добавления и использования новых команд репликации, включая пользовательские.

F.41.2. Ограничения и требования

В настоящее время pgpro_bindump имеет следующие ограничения и требования:

- Для работы pgpro_bindump требуются `libpgprobackup.so` и `libpb3_encoder.so`.

F.41.3. Установка и настройка

Модуль pgpro_bindump входит в состав Postgres Pro как часть пакета `postgrespro-std-17-contrib` (подробные инструкции по установке приведены в Главе 16). После установки Postgres Pro выполните шаги ниже.

1. Задайте следующие параметры в файле `postgresql.conf`:

```
shared_preload_libraries = 'pgpro_bindump'
wal_level = 'replica' # or 'logical'
walsender_plugin_libraries = 'pgpro_bindump'
```

2. Перезапустите экземпляр Postgres Pro.

F.41.4. Команды репликации

Модуль pgpro_bindump поддерживает следующие команды репликации:

- `just-noop` — команда-заглушка, не выполняющая никаких действий.
- `start_backup` — запускает процесс резервного копирования.

- `copy_files` — копирует файлы в формате `CBOR`.
- `stop_backup` — останавливает процесс резервного копирования.

Резервное копирование выполняется в режиме PRO — расширенном режиме источника данных, включающем все доступные оптимизации и улучшения, в том числе проприетарный протокол репликации и специализированные команды репликации.

Для отправки команд из пользовательского приложения в pgpro_bindump и получения приложением выходных данных из pgpro_bindump используйте libpq.

> **Примечание**
>
> Для работы требуется активное подключение к работающему серверу Postgres Pro.

F.41.4.1. start_backup

```
start_backup LABEL метка, INCR_MODE инкрементальный_режим, START_LSN начальный_lsn,
 TRANSFER_MODE режим_передачи,
[TRANSFER_BUFFER_SIZE размер_буфера]
```

Запускает процесс резервного копирования. Передаёт приложению следующие данные:

- `start_lsn` — LSN на момент начала резервного копирования.
- `start_tli` — идентификатор линии времени на момент начала резервного копирования.
- `min_xid` — минимальный идентификатор транзакции на момент начала резервного копирования.
- `min_multixact` — минимальный идентификатор мультитранзакции на момент начала резервного копирования.
- `tablespace_list` — список табличных пространств, включённых в резервную копию.
- `database_list` — список баз данных, включённых в резервную копию.

`LABEL` *`метка`*

Имя резервной копии.

`INCR_MODE` *`инкрементальный_режим`*

Тип инкрементального копирования. Допустимые значения: `PTRACK`, `DELTA`.

`START_LSN` *`начальный_lsn`*

Значение типа `XLogRecPtr`, представляющее LSN на момент начала резервного копирования.

`TRANSFER_MODE` *`режим_передачи`*

Способ передачи данных с сервера в приложение. Возможные значения:

- `raw` — данные передаются в несжатом виде блоками произвольного размера.
- `packed` — данные передаются в упакованном виде блоками по 128 КБ с общим заголовком.

`packed` — значение по умолчанию.

`TRANSFER_BUFFER_SIZE` *`размер_буфера`*

Числовое значение, определяющее размер буфера. Единственное возможное значение — значение по умолчанию: 131072 (128 КБ).

F.41.4.2. copy_files

```
copy_files VERIFY_CHECKSUMS [TRUE | FALSE], COMPRESS_ALG алгоритм_сжатия,
 COMPRESS_LVL уровень_сжатия, TRANSFER_MODE режим_передачи,
[INCR_MODE инкрементальный_режим], [START_LSN начальный_lsn],
 [TRANSFER_BUFFER_SIZE размер_буфера],
[WORKER_INFO число_процессов:ид_процесса]
```

Передаёт блоки файлов данных и файлов WAL, упакованные в формате `CBOR`.

`VERIFY_CHECKSUMS`

Логическое значение (`true` или `false`), определяющее необходимость проверки контрольных сумм.

`COMPRESS_ALG` *`алгоритм_сжатия`*

Используемый алгоритм сжатия. Допустимые значения: `zlib`, `zstd`, `lz4`, `none`.

`COMPRESS_LVL` *`уровень_сжатия`*

Уровень сжатия. Возможные значения: 1, 2, 3, 4, 5.

`TRANSFER_MODE` *`режим_передачи`*

Способ передачи данных с сервера в приложение. Возможные значения:

- `raw` — данные передаются в несжатом виде блоками произвольного размера.
- `packed` — данные передаются в упакованном виде блоками по 128 КБ с общим заголовком.

`packed` — значение по умолчанию.

`INCR_MODE` *`инкрементальный_режим`*

Тип инкрементального копирования. Допустимые значения: `PTRACK`, `DELTA`.

`START_LSN` *`начальный_lsn`*

Значение типа `XLogRecPtr`, представляющее LSN на момент начала резервного копирования.

`TRANSFER_BUFFER_SIZE` *`размер_буфера`*

Числовое значение, определяющее размер буфера. Единственное возможное значение — значение по умолчанию: 131072 (128 КБ).

`WORKER_INFO` *`число_процессов`*`:`*`ид_процесса`*

Количество процессов и идентификатор процессора.

F.41.4.3. stop_backup

```
stop_backup STREAM [TRUE | FALSE], COMPRESS_ALG алгоритм_сжатия,
 COMPRESS_LVL уровень_сжатия,
[EXTERNALS внешние_каталоги] [TRANSFER_BUFFER_SIZE размер_буфера],
 [TRANSFER_MODE режим_передачи]
```

Завершает процесс копирования и закрывает текущее соединение. Отправляет в приложение следующие данные:

- `stop_lsn` — LSN на момент завершения резервного копирования.
- `stop_tli` — идентификатор линии времени на момент завершения резервного копирования.
- Файл `backup.control`.
- Набор файлов WAL (только при включённой передаче WAL).

`STREAM`

Логическое значение (`true` или `false`), определяющее, включена ли потоковая передача WAL.

`COMPRESS_ALG` *`алгоритм_сжатия`*

Используемый алгоритм сжатия. Допустимые значения: `zlib`, `zstd`, `lz4`, `none`.

`COMPRESS_LVL` *`уровень_сжатия`*

Уровень сжатия. Возможные значения: 1, 2, 3, 4, 5.

`EXTERNALS` *`внешние_каталоги`*

Список внешних каталогов.

`TRANSFER_BUFFER_SIZE` *`размер_буфера`*

Числовое значение, определяющее размер буфера. Единственное возможное значение — значение по умолчанию: 131072 (128 КБ).

`TRANSFER_MODE` *`режим_передачи`*

Способ передачи данных с сервера в приложение. Возможные значения:

- `raw` — данные передаются в несжатом виде блоками произвольного размера.
- `packed` — данные передаются в упакованном виде блоками по 128 КБ с общим заголовком.

`packed` — значение по умолчанию.

F.41.5. Авторы

Postgres Professional, Москва, Россия.

F.42. pg_prewarm — предварительная загрузка данных отношений в кеш буферов

Модуль `pg_prewarm` предоставляет удобную возможность загружать данные отношений в кеш операционной системы или в кеш буферов Postgres Pro. Предварительную загрузку можно выполнить вручную, вызвав функцию `pg_prewarm`, или автоматически, добавив `pg_prewarm` в shared_preload_libraries. Во втором случае система запустит фоновый процесс, который будет периодически записывать содержимое разделяемых буферов в файл `autoprewarm.blocks` с тем, чтобы эти блоки подгружались в память при запуске сервера, используя два дополнительных фоновых процесса.

F.42.1. Функции

```
pg_prewarm(regclass, mode text default 'buffer', fork text default 'main',
           first_block int8 default null,
           last_block int8 default null) RETURNS int8
```

Первый аргумент задаёт отношение, которое будет «разогрето». Во втором указывается метод «разогрева», из описанных ниже; в третьем задаётся целевой слой отношения, обычно `main`. В четвёртом аргументе можно передать номер первого разогреваемого блока (`NULL` принимается как синоним нуля), а в пятом — номер последнего блока (`NULL` означает последний блок отношения). Возвращает эта функция количество разогретых блоков.

Эта функция поддерживает три режима разогрева. В режиме `prefetch` выдаются асинхронные запросы предвыборки данных операционной системе, если они поддерживаются, либо происходит ошибка. В режиме `read` считывается заданный диапазон блоков; в отличие от `prefetch` это происходит синхронно и поддерживается во всех ОС и любыми сборками, но может быть медленнее. В режиме `buffer` запрошенный диапазон блоков считывается в кеш буферов базы данных.

Заметьте, что с любым из этих методов попытка разогреть больше блоков, чем может уместиться в кеше (в кеше ОС в режимах `prefetch` и `read`, либо в кеше Postgres Pro в режиме `buffer`) скорее всего приведёт к тому, что блоки с меньшими номерами будут вытеснены из кеша при чтении последующих блоков. Кроме того, разогретые данные никаким специальным образом не защищаются от вытеснения из кеша, так что возможна ситуация, когда из-за другой активности только что разогретые блоки будут вытеснены вскоре после чтения; с другой стороны при таком разогреве из кеша могут быть вытеснены другие данные. Поэтому разогрев обычно наиболее полезен при загрузке, когда кеши в основном пусты.

```
autoprewarm_start_worker() RETURNS void
```

Запустить основной рабочий процесс авторазогрева. Обычно он запускается автоматически, но эта функция полезна, если автоматический разогрев не был настроен при запуске сервера и вы хотите запустить этот процесс позже.

```
autoprewarm_dump_now() RETURNS int8
```

Обновить `autoprewarm.blocks` немедленно. Это может быть полезно, если рабочий процесс авторазогрева не работает, но вы хотите, чтобы авторазогрев был произведён при перезапуске. Эта функция возвращает число записей, внесённых в `autoprewarm.blocks`.

F.42.2. Параметры конфигурации

`pg_prewarm.autoprewarm` (`boolean`)

Указывает, должен ли сервер запускать рабочий процесс авторазогрева. По умолчанию он включён. Задать этот параметр можно только при запуске сервера.

`pg_prewarm.autoprewarm_interval` (`integer`)

Задаёт интервал между обновлениями файла `autoprewarm.blocks`. Значение по умолчанию — 300 сек. При значении, равном 0, файл будет сохраняться не периодически, а только при отключении сервера.

Эти параметры должны задаваться в `postgresql.conf`. Обычное использование выглядит так:

```
# postgresql.conf
shared_preload_libraries = 'pg_prewarm'

pg_prewarm.autoprewarm = true
pg_prewarm.autoprewarm_interval = 300s
```

F.42.3. Автор

Роберт Хаас `<rhaas@postgresql.org>`

F.43. pg_query_state — возможность узнавать текущее состояние выполнения запросов в работающем обслуживающем процессе

Модуль `pg_query_state` даёт возможность узнавать текущее состояние выполнения запросов в работающем обслуживающем процессе.

F.43.1. Обзор

Каждый обычный (не служебный) оператор запроса (SELECT/INSERT/UPDATE/DELETE) после этапа оптимизации/планирования переводится в дерево плана, представляющего собой своего рода императивное представление SQL-запроса. Команда EXPLAIN ANALYZE позволяет просмотреть статистику выполнения, собранную с каждого узла этого дерева плана (общее время выполнения, число строк, выданных верхних узлам и т. д.). Но эта статистика собирается только после выполнения запроса. Данный модуль позволяет просмотреть актуальную статистику запроса, работающего во внешнем обслуживающем процессе. При этом формат результирующего вывода практически идентичен выводу обычной команды EXPLAIN ANALYZE. Таким образом, пользователи могут отслеживать собственно процесс выполнения запросов. В действительности, этот модуль может исследовать внешний обслуживающий процесс и определять его фактическое состояние. В частности, это полезно, когда обслуживающий процесс выполняет очень сложный запрос или «завис».

F.43.2. Сценарии использования

С помощью этого модуля возможно:

- выявлять сложные запросы (в совокупности с другими средствами мониторинга);
- наблюдать за выполнением запросов.

F.43.3. Установка

Чтобы установить pg_query_state, выполните следующие действия:

1. Добавьте имя библиотеки в параметр `shared_preload_libraries` в конфигурационном файле `postgresql.conf`:

   ```
   shared_preload_libraries = 'pg_query_state'
   ```

2. Перезагрузите сервер баз данных, чтобы изменения вступили в силу.
3. Создайте расширение pg_query_state, выполнив следующий запрос:

   ```
   CREATE EXTENSION pg_query_state;
   ```

4. Чтобы обновить расширение pg_query_state, выполните команду `ALTER EXTENSION UPDATE`.

F.43.4. Функции

F.43.4.1. pg_query_state

```
pg_query_state (integer  pid,
                verbose  boolean DEFAULT FALSE,
                costs    boolean DEFAULT FALSE,
                timing   boolean DEFAULT FALSE,
                buffers  boolean DEFAULT FALSE,
                triggers boolean DEFAULT FALSE,
                format   text    DEFAULT 'text')
  returns TABLE (pid integer,
                 frame_number integer,
```

```
                    query_text text,
                    plan text,
                    leader_pid integer)
```

Получает от обслуживающего процесса с заданным `pid` состояние текущего запроса. При выполнении параллельного запроса может порождаться множество параллельных запросов, а при вызовах функций образуются вложенные подзапросы, так что состояние выполнения может представляться в виде стека запросов, поэтому данная функция возвращает результат типа `TABLE (pid integer, frame_number integer, query_text text, plan text, leader_pid integer)`. Он представляет структуру дерева, состоящего ведущий процесс и порождённые им рабочие процессы, идентифицируемые по `pid`. Каждый рабочий процесс ссылается на свой ведущий по столбцу `leader_pid`. Для ведущего процесса значение этого столбца — `NULL`. Состояние каждого процесса представляется в виде стека вызовов функций. Для каждого кадра этого стека устанавливается соответствие по столбцам `frame_number` (номер кадра, начиная с нуля), `query_text` (текст запроса) и `plan` (план) со столбцами текущей статистики.

Таким образом, пользователь может видеть состояние основного запроса и запросов, порождённых вызовами функции, для ведущего процесса и всех рабочих процессов, порождённых им.

В процессе выполнения некоторые узлы плана могут представлять циклы. Поэтому статистика по такому узлу состоит из двух частей: средняя статистика по предыдущим циклам, как в выводе EXPLAIN ANALYZE, и статистика для текущего, ещё не законченного цикла.

Необязательные аргументы:

- verbose — использовать EXPLAIN VERBOSE для вывода плана;
- costs — стоимости каждого узла;
- timing — выводить время исполнения для каждого узла; если сбор временной статистики на вызываемой стороне отключён, в результате будет выдаваться ПРЕДУПРЕЖДЕНИЕ `timing statistics disabled`;
- buffers — выводить использование буферов; если сбор статистики буферов на вызываемой стороне отключён, в результате будет выдаваться ПРЕДУПРЕЖДЕНИЕ `buffers statistics disabled`;
- triggers — включать в деревья планов статистику выполнения триггеров;
- format — задаёт формат EXPLAIN для вывода планов; возможные значения: text, xml, json и yaml.

Если вызываемый обслуживающий процесс не выполняет запросы, данная функция выводит сообщение о его состоянии, полученном из представления `pg_stat_activity`, если там есть информация о нём.

Вызывать эту функцию разрешено только суперпользователям или членам роли, которой принадлежит обслуживающий процесс. В противном случае функция печатает сообщение `permission denied`.

F.43.4.2. pg_progress_bar

```
pg_progress_bar (integer pid)
  returns FLOAT
```

Возвращает текущий прогресс выполнения запроса. Сначала функция получает от обслуживающего процесса с заданным `pid` состояние текущего запроса. Затем на основании информации из всех узлов плана она собирает общее фактическое и ожидаемое число строк и вычисляет их соотношение для всего дерева выполнения запроса. В результате функция возвращает числовое значение в диапазоне от `0` до `1`. Оно обозначает прогресс выполнения запроса.

Функцию можно использовать в графическом интерфейсе Postgres Pro, чтобы отобразить индикатор выполнения запроса.

Если информация о текущем состоянии запроса отсутствует или вычислить прогресс выполнения невозможно, выводится соответствующее сообщение.

Эту функцию могут вызывать только члены роли, которая владеет обслуживающим процессом, или суперпользователи.

F.43.4.3. pg_progress_bar_visual

```
pg_progress_bar_visual (integer pid,
                        integer delay)
  returns VOID
```

Циклически вычисляет прогресс выполнения запроса и выводит его в текстовом формате. Функция получает от обслуживающего процесса с заданным `pid` состояние текущего запроса через интервалы времени, определённые в параметре *`delay`* (в секундах).

Эта функция является циклическим вариантом функции pg_progress_bar, но возвращает `void`. В отличие от `pg_progress_bar`, функцию `pg_progress_bar_visual` можно использовать, чтобы отслеживать прогресс выполнения запроса без графического интерфейса.

Если информация о текущем состоянии запроса отсутствует или вычислить прогресс выполнения невозможно, выводится соответствующее сообщение.

Эту функцию могут вызывать только члены роли, которая владеет обслуживающим процессом, или суперпользователи.

F.43.5. Конфигурационные параметры

Для управления всем модулем и сбора определённых статистических показателей в процессе выполнения запроса доступен также набор доступных пользователю переменных GUC:

- pg_query_state.enable — отключить (или включить) pg_query_state полностью, значение по умолчанию: `true`
- pg_query_state.enable_timing — собирать статистику времени по каждому узлу, значение по умолчанию: `false`
- pg_query_state.enable_buffers — собирать статистику по использованию буферов; значение по умолчанию: `false`

Эти параметры должны устанавливаться на вызываемой стороне до выполнения запросов, статистика которых представляет интерес. Если параметр pg_query_state.enable_timing отключён, вызывающая сторона не может получать статистику по времени и выдаётся `ПРЕДУПРЕЖДЕНИЕ`; аналогично с параметром pg_query_state.enable_buffers.

F.43.6. Примеры использования

Установить максимальное число параллельных исполнителей для узла `Gather` равным 2:

```
      postgres=# set max_parallel_workers_per_gather = 2;
```

Предположим, что один обслуживающий процесс с `pid` = 49265 выполняет простой запрос:

```
postgres=# select pg_backend_pid();
pg_backend_pid
----------------
    49265
(1 row)
postgres=# select count(*) from foo join bar on foo.c1=bar.c1;
```

Другие обслуживающие процессы могут извлекать промежуточное состояние о выполнении этого запроса:

```
postgres=# \x
postgres=# select * from pg_query_state(49265);
-[ RECORD
 1 ]+---------------------------------------------------------------------------------------
pid          | 49265
frame_number | 0
query_text   | select count(*) from foo join bar on foo.c1=bar.c1;
plan         | Finalize Aggregate (Current loop: actual rows=0, loop number=1)
                                                +
             |   ->  Gather (Current loop: actual rows=0, loop number=1)
                                                +
             |         Workers Planned: 2
                                                +
             |         Workers Launched: 2
                                                +
             |         ->  Partial Aggregate (Current loop: actual rows=0, loop
 number=1)                                              +
             |               ->  Nested Loop (Current loop: actual rows=12, loop
 number=1)                                             +
             |                     Join Filter: (foo.c1 = bar.c1)
                                                +
             |                     Rows Removed by Join Filter: 5673232
                                                +
             |                     ->  Parallel Seq Scan on foo (Current loop: actual
 rows=12, loop number=1)                          +
             |                     ->  Seq Scan on bar (actual rows=500000 loops=11)
 (Current loop: actual rows=173244, loop number=12)
leader_pid   | (null)
-[ RECORD
 2 ]+---------------------------------------------------------------------------------------
pid          | 49324
frame_number | 0
query_text   | <parallel query>
plan         | Partial Aggregate (Current loop: actual rows=0, loop number=1)
                                                +
             |   ->  Nested Loop (Current loop: actual rows=10, loop number=1)
                                                +
             |         Join Filter: (foo.c1 = bar.c1)
                                                +
             |         Rows Removed by Join Filter: 4896779
                                                +
             |         ->  Parallel Seq Scan on foo (Current loop: actual rows=10, loop
 number=1)                                      +
             |         ->  Seq Scan on bar (actual rows=500000 loops=9) (Current loop:
 actual rows=396789, loop number=10)
leader_pid   | 49265
-[ RECORD
 3 ]+---------------------------------------------------------------------------------------
pid          | 49323
frame_number | 0
query_text   | <parallel query>
plan         | Partial Aggregate (Current loop: actual rows=0, loop number=1)
                                                +
             |   ->  Nested Loop (Current loop: actual rows=11, loop number=1)
                                                +
```

```
             |         Join Filter: (foo.c1 = bar.c1)
                                                +
             |         Rows Removed by Join Filter: 5268783
                                                +
             |         ->  Parallel Seq Scan on foo (Current loop: actual rows=11, loop
 number=1)                                      +
             |         ->  Seq Scan on bar (actual rows=500000 loops=10) (Current loop:
 actual rows=268794, loop number=11)
leader_pid   | 49265
```

В показанном выше примере обслуживающий процесс порождает два параллельных рабочих процесса с PID 49324 и 49323. Их значения столбца `leader_pid` показывают, что эти рабочие процессы относятся к основном ведущему. Узел `Seq Scan` показывает статистику по выполненным циклам (показывается среднее число строк, переданное узлу `Nested Loop`, и число выполненных циклов) и статистику по текущему узлу. Другие узлы показывают статистику только по текущему циклу, так как это первая итерация (номер цикла = 1).

Предположим, что первый обслуживающий процесс выполняет некоторую функцию:

```
postgres=# select n_join_foo_bar();
```

Другой процесс может получить такую информацию:

```
postgres=# select * from pg_query_state(49265);
-[ RECORD
 1 ]+-------------------------------------------------------------------------------------
pid          | 49265
frame_number | 0
query_text   | select n_join_foo_bar();
plan         | Result (Current loop: actual rows=0, loop number=1)
leader_pid   | (null)
-[ RECORD
 2 ]+-------------------------------------------------------------------------------------
pid          | 49265
frame_number | 1
query_text   | SELECT (select count(*) from foo join bar on foo.c1=bar.c1)
plan         | Result (Current loop: actual rows=0, loop number=1)
                                         +
             |   InitPlan 1 (returns $0)
                                         +
             |     ->  Aggregate (Current loop: actual rows=0, loop number=1)
                                         +
             |           ->  Nested Loop (Current loop: actual rows=51, loop number=1)
                                         +
             |                 Join Filter: (foo.c1 = bar.c1)
                                         +
             |                 Rows Removed by Join Filter: 51636304
                                         +
             |                 ->  Seq Scan on bar (Current loop: actual rows=52, loop
 number=1)                                +
             |                 ->  Materialize (actual rows=1000000 loops=51) (Current
 loop: actual rows=636355, loop number=52)+
             |                       ->  Seq Scan on foo (Current loop: actual
 rows=1000000, loop number=1)
leader_pid   | (null)
```

Первая строка соответствует вызову функции, вторая — запросу, содержащемуся в теле этой функции.

Мы можем получать результирующие планы в различных форматах (например, в json):

```
postgres=# select * from pg_query_state(pid := 49265, format := 'json');
-[ RECORD 1 ]+------------------------------------------------------------
pid          | 49265
frame_number | 0
query_text   | select * from n_join_foo_bar();
plan         | {                                                          +
             |   "Plan": {                                                +
             |     "Node Type": "Function Scan",                          +
             |     "Parallel Aware": false,                               +
             |     "Function Name": "n_join_foo_bar",                     +
             |     "Alias": "n_join_foo_bar",                             +
             |     "Current loop": {                                      +
             |       "Actual Loop Number": 1,                             +
             |       "Actual Rows": 0                                     +
             |     }                                                      +
             |   }                                                        +
             | }
leader_pid   | (null)
-[ RECORD 2 ]+------------------------------------------------------------
pid          | 49265
frame_number | 1
query_text   | SELECT (select count(*) from foo join bar on foo.c1=bar.c1)
plan         | {                                                          +
             |   "Plan": {                                                +
             |     "Node Type": "Result",                                 +
             |     "Parallel Aware": false,                               +
             |     "Current loop": {                                      +
             |       "Actual Loop Number": 1,                             +
             |       "Actual Rows": 0                                     +
             |     },                                                     +
             |     "Plans": [                                             +
             |       {                                                    +
             |         "Node Type": "Aggregate",                          +
             |         "Strategy": "Plain",                               +
             |         "Partial Mode": "Simple",                          +
             |         "Parent Relationship": "InitPlan",                 +
             |         "Subplan Name": "InitPlan 1 (returns $0)",         +
             |         "Parallel Aware": false,                           +
             |         "Current loop": {                                  +
             |           "Actual Loop Number": 1,                         +
             |           "Actual Rows": 0                                 +
             |         },                                                 +
             |         "Plans": [                                         +
             |           {                                                +
             |             "Node Type": "Nested Loop",                    +
             |             "Parent Relationship": "Outer",                +
             |             "Parallel Aware": false,                       +
             |             "Join Type": "Inner",                          +
             |             "Current loop": {                              +
             |               "Actual Loop Number": 1,                     +
             |               "Actual Rows": 610                           +
             |             },                                             +
             |             "Join Filter": "(foo.c1 = bar.c1)",            +
             |             "Rows Removed by Join Filter": 610072944,      +
             |             "Plans": [                                     +
             |               {                                            +
```

```
             |                 "Node Type": "Seq Scan",                   +
             |                 "Parent Relationship": "Outer",            +
             |                 "Parallel Aware": false,                   +
             |                 "Relation Name": "bar",                    +
             |                 "Alias": "bar",                            +
             |                 "Current loop": {                          +
             |                   "Actual Loop Number": 1,                 +
             |                   "Actual Rows": 611                       +
             |                 }                                          +
             |               },                                           +
             |               {                                            +
             |                 "Node Type": "Materialize",                +
             |                 "Parent Relationship": "Inner",            +
             |                 "Parallel Aware": false,                   +
             |                 "Actual Rows": 1000000,                    +
             |                 "Actual Loops": 610,                       +
             |                 "Current loop": {                          +
             |                   "Actual Loop Number": 611,               +
             |                   "Actual Rows": 73554                     +
             |                 },                                         +
             |                 "Plans": [                                 +
             |                   {                                        +
             |                     "Node Type": "Seq Scan",               +
             |                     "Parent Relationship": "Outer",        +
             |                     "Parallel Aware": false,               +
             |                     "Relation Name": "foo",                +
             |                     "Alias": "foo",                        +
             |                     "Current loop": {                      +
             |                       "Actual Loop Number": 1,             +
             |                       "Actual Rows": 1000000               +
             |                     }                                      +
             |                   }                                        +
             |                 ]                                          +
             |               }                                            +
             |             ]                                              +
             |           }                                                +
             |         ]                                                  +
             |       }                                                    +
             |     ]                                                      +
             |   }                                                        +
             | }
leader_pid   | (null)
```

F.43.7. Примеры использования индикатора выполнения запроса

Первый обслуживающий процесс выполняет следующий запрос:

```
postgres=#  INSERT INTO table_name SELECT generate_series(1,10000000);
```

С помощью другого обслуживающего процесса можно проверить прогресс выполнения этого запроса:

```
postgres=# SELECT pid FROM pg_stat_activity WHERE query LIKE 'insert%';
  pid
-------
 23877
(1 row)
```

```
postgres=# SELECT pg_progress_bar(23877);
 pg_progress_bar
-----------------
       0.6087927
(1 row)
```

Чтобы визуально отслеживать прогресс выполнения, выполните следующий запрос:

```
postgres=# SELECT pg_progress_bar_visual(23877, 1);
Progress = 0.043510
Progress = 0.085242
Progress = 0.124921
Progress = 0.168168
Progress = 0.213803
Progress = 0.250362
Progress = 0.292632
Progress = 0.331454
Progress = 0.367509
Progress = 0.407450
Progress = 0.448646
Progress = 0.488171
Progress = 0.530559
Progress = 0.565558
Progress = 0.608039
Progress = 0.645778
Progress = 0.654842
Progress = 0.699006
Progress = 0.735760
Progress = 0.787641
Progress = 0.832160
Progress = 0.871077
Progress = 0.911858
Progress = 0.956362
Progress = 0.995097
Progress = 1.000000
 pg_progress_bar_visual
------------------------
                      1
(1 row)
```

F.44. pgrowlocks — отображение информации о блокировке строк таблицы

Модуль `pgrowlocks` предоставляет функцию, показывающую информацию о блокировке строк для заданной таблицы.

По умолчанию его использование разрешено суперпользователям, ролям с правами роли `pg_stat_scan_tables` и пользователям с правом `SELECT` в заданной таблице.

F.44.1. Обзор

```
pgrowlocks(text) returns setof record
```

В параметре передаётся имя таблицы. В результате возвращается набор записей, в котором строка соответствует строке, заблокированной в таблице. Столбцы результата показаны в Таблице F.25.

Таблица F.25. Столбцы результата `pgrowlocks`

Имя	Тип	Описание
`locked_row`	`tid`	Идентификатор кортежа (TID) блокированной строки
`locker`	`xid`	Идентификатор блокирующей транзакции или идентификатор мультитранзакции, если это мультитранзакция (см. Раздел 64.1)
`multi`	`boolean`	True, если блокирующий субъект — мультитранзакция
`xids`	`xid[]`	Идентификаторы блокирующих транзакций (больше одной для мультитранзакции)
`modes`	`text[]`	Режим блокировки (больше одного для мультитранзакции), массив со значениями `For Key Share`, `For Share`, `For No Key Update`, `No Key Update`, `For Update`, `Update`.
`pids`	`integer[]`	Идентификаторы блокирующих обслуживающих процессов (больше одного для мультитранзакции)

Функция `pgrowlocks` запрашивает блокировку `AccessShareLock` для целевой таблицы и считывает строку за строкой для сбора информации о блокировке строк. Это происходит небыстро для большой таблицы. Заметьте, что:

1. Если таблица заблокирована в режиме `ACCESS EXCLUSIVE`, функция `pgrowlocks` будет блокироваться.
2. Функция `pgrowlocks` не гарантирует внутреннюю согласованность результатов. В ходе её выполнения могут быть установлены новые блокировки строк, либо освобождены старые.

Функция `pgrowlocks` не показывает содержимое заблокированных строк. Если вы хотите параллельно взглянуть на содержимое строк, можно проделать примерно следующее:

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

Однако учтите, что такой запрос будет очень неэффективным.

F.44.2. Пример вывода

```
=# SELECT * FROM pgrowlocks('t1');
 locked_row | locker | multi | xids  |     modes      |  pids
------------+--------+-------+-------+----------------+--------
 (0,1)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,2)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,3)      |    607 | f     | {607} | {"For Update"} | {3107}
 (0,4)      |    607 | f     | {607} | {"For Update"} | {3107}
(4 rows)
```

F.44.3. Автор

Тацуо Исии (Tatsuo Ishii)

F.45. pg_stat_statements — отслеживание статистики планирования и выполнения SQL-операторов

Модуль `pg_stat_statements` предоставляет возможность отслеживать статистику планирования и выполнения сервером всех операторов SQL.

Этот модуль нужно загружать, добавив `pg_stat_statements` в shared_preload_libraries в файле `postgresql.conf`, так как ему требуется дополнительная разделяемая память. Это значит, что для загрузки или выгрузки модуля необходимо перезапустить сервер. Кроме того, для функционирования этого модуля должны вычисляться идентификаторы запросов, что происходит автоматически, когда для параметра compute_query_id задаётся значение `auto` или `on` или же загружается сторонний модуль, вычисляющий идентификаторы запросов.

Когда модуль `pg_stat_statements` активен, он отслеживает статистику по всем базам данных на сервере. Для получения и обработки этой статистики модуль `pg_stat_statements` предоставляет представления `pg_stat_statements` и `pg_stat_statements_info`, а также вспомогательные функции `pg_stat_statements_reset` и `pg_stat_statements`. Эти объекты не доступны глобально, но их можно установить в определённой базе данных, выполнив команду `CREATE EXTENSION pg_stat_statements`.

F.45.1. Представление `pg_stat_statements`

Статистика, собираемая модулем, выдаётся через представление с именем `pg_stat_statements`. Это представление содержит отдельные строки для каждой комбинации идентификатора базы данных, идентификатора пользователя, идентификатора запроса и признака верхнего уровня (но в количестве, не превышающем максимальное число различных операторов, которые может отслеживать модуль). Столбцы представления показаны в Таблице F.26.

Таблица F.26. Столбцы `pg_stat_statements`

Тип столбца Описание
`userid` `oid` (ссылается на `pg_authid`.`oid`) OID пользователя, выполнявшего оператор
`dbid` `oid` (ссылается на `pg_database`.`oid`) OID базы данных, в которой выполнялся оператор
`toplevel` `bool` True, если данный запрос выполнялся на верхнем уровне (всегда true, если для параметра `pg_stat_statements.track` задано значение `top`)
`queryid` `bigint` Хеш-код для выявления одинаковых нормализованных запросов
`query` `text` Текст, представляющий оператор
`plans` `bigint` Число операций планирования этого оператора (если включён параметр `pg_stat_statements.track_planning`, иначе 0)
`total_plan_time` `double precision` Общее время, затраченное на планирование этого оператора в миллисекундах (если включён параметр `pg_stat_statements.track_planning`, иначе 0)
`min_plan_time` `double precision` Минимальное время, затраченное на планирование этого оператора, в миллисекундах. В столбце отобразится 0 при отключённом параметре `pg_stat_statements.track_planning` или если счётчик был сброшен при помощи функции `pg_stat_statements_reset` с параметром `minmax_only`, выставленным в `true`, и планирование больше не производилось.

Тип столбца Описание
`max_plan_time double precision` Максимальное время, затраченное на планирование этого оператора, в миллисекундах. В столбце отобразится 0 при отключённом параметре `pg_stat_statements.track_planning` или если счётчик был сброшен при помощи функции `pg_stat_statements_reset` с параметром `minmax_only`, выставленным в `true`, и планирование больше не производилось.
`mean_plan_time double precision` Среднее время, затраченное на планирование этого оператора в миллисекундах (если включён параметр `pg_stat_statements.track_planning`, иначе 0)
`stddev_plan_time double precision` Стандартное отклонение времени, затраченного на планирование этого оператора в миллисекундах (если включён параметр `pg_stat_statements.track_planning`, иначе 0)
`calls bigint` Число выполнений
`total_exec_time double precision` Общее время, затраченное на выполнение оператора, в миллисекундах
`min_exec_time double precision` Минимальное время, затраченное на выполнение оператора, в миллисекундах. В столбце отобразится 0, если оператор выполняется впервые после очистки статистики функцией `pg_stat_statements_reset` с параметром `minmax_only`, выставленным в `true`
`max_exec_time double precision` Максимальное время, затраченное на выполнение оператора, в миллисекундах. В столбце отобразится 0, если оператор выполняется впервые после очистки статистики функцией `pg_stat_statements_reset` с параметром `minmax_only`, выставленным в `true`
`mean_exec_time double precision` Среднее время, затраченное на выполнение оператора, в миллисекундах
`stddev_exec_time double precision` Стандартное отклонение времени, затраченного на выполнение оператора, в миллисекундах
`rows bigint` Общее число строк, полученных или затронутых оператором
`shared_blks_hit bigint` Общее число попаданий разделяемых блоков в кеш для данного оператора
`shared_blks_read bigint` Общее число разделяемых блоков, прочитанных данным оператором
`shared_blks_dirtied bigint` Общее число разделяемых блоков, «загрязнённых» данным оператором
`shared_blks_written bigint` Общее число разделяемых блоков, записанных данным оператором
`local_blks_hit bigint` Общее число попаданий локальных блоков в кеш для данного оператора
`local_blks_read bigint` Общее число локальных блоков, прочитанных данным оператором
`local_blks_dirtied bigint` Общее число локальных блоков, «загрязнённых» данным оператором
`local_blks_written bigint` Общее число локальных блоков, записанных данным оператором

Тип столбца	**Описание**
`temp_blks_read bigint`	Общее число временных блоков, прочитанных данным оператором
`temp_blks_written bigint`	Общее число временных блоков, записанных данным оператором
`shared_blk_read_time double precision`	Общее время, затраченное оператором на чтение разделяемых блоков, в миллисекундах (если включён track_io_timing, иначе ноль)
`shared_blk_write_time double precision`	Общее время, затраченное оператором на запись разделяемых блоков, в миллисекундах (если включён track_io_timing, иначе ноль)
`local_blk_read_time double precision`	Общее время, затраченное оператором на чтение локальных блоков, в миллисекундах (если включён track_io_timing, иначе ноль)
`local_blk_write_time double precision`	Общее время, затраченное оператором на запись локальных блоков, в миллисекундах (если включён track_io_timing, иначе ноль)
`temp_blk_read_time double precision`	Общее время, затраченное оператором на чтение блоков временных файлов, в миллисекундах (если включён track_io_timing, или ноль в противном случае)
`temp_blk_write_time double precision`	Общее время, затраченное оператором на запись блоков временных файлов, в миллисекундах (если включён track_io_timing, или ноль в противном случае)
`wal_records bigint`	Общее число записей WAL, сгенерированных при выполнении оператора
`wal_fpi bigint`	Общее число образов полных страниц в WAL, сгенерированных при выполнении оператора
`wal_bytes numeric`	Общий объём WAL, сгенерированный при выполнении оператора, в байтах
`jit_functions bigint`	Общее число функций, скомпилированных в JIT-код при выполнении данного оператора
`jit_generation_time double precision`	Общее время, затраченное на компиляцию JIT-кода при выполнении оператора, в миллисекундах
`jit_inlining_count bigint`	Сколько раз встраивались функции
`jit_inlining_time double precision`	Общее время, затраченное на встраивание функций при выполнении оператора, в миллисекундах
`jit_optimization_count bigint`	Число JIT-оптимизаций для данного оператора
`jit_optimization_time double precision`	Общее время, затраченное на JIT-оптимизацию при выполнении оператора, в миллисекундах
`jit_emission_count bigint`	Сколько раз выдавался код
`jit_emission_time double precision`	Общее время, затраченное на выдачу кода при выполнении оператора, в миллисекундах

Тип столбца	Описание
`jit_deform_count` `bigint`	Общее число функций преобразования кортежей, скомпилированных в JIT-код при выполнении данного оператора
`jit_deform_time` `double precision`	Общее время, затраченное оператором на компилирование функций преобразования кортежей в JIT-код, в миллисекундах
`stats_since` `timestamp with time zone`	Время начала сбора статистики по этому оператору
`minmax_stats_since` `timestamp with time zone`	Время начала сбора статистики минимальных/максимальных значений по этому оператору (столбцы `min_plan_time`, `max_plan_time`, `min_exec_time` и `max_exec_time`)

По соображениям безопасности только суперпользователям и ролям с правами роли `pg_read_all_stats` разрешено видеть текст SQL и `queryid` запросов, выполняемых другими пользователями. Однако другие пользователи могут видеть статистику, если это представление установлено в их базу данных.

Планируемые запросы (то есть `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`) и служебные команды объединяются в одну запись в `pg_stat_statements`, когда они имеют идентичные структуры запросов согласно внутреннему вычисленному хешу. Обычно два запроса будут считаться равными при таком сравнении, если они семантически равнозначны, не считая значений констант, фигурирующих в запросе.

Примечание

Нижеприведённые соображения о замене констант и `queryid` применимы, только если включён compute_query_id. Если же для вычисления `queryid` вы используете встроенный модуль, обратитесь к его документации.

Когда значение константы игнорируется в целях сравнения запроса с другими запросами, эта константа заменяется в выводе `pg_stat_statements` обозначением параметра, например, `$1`. В остальном этот вывод содержит текст первого запроса, хеш которого равнялся значению `queryid`, связанному с записью в `pg_stat_statements`.

В `pg_stat_statements` запросы, для которых может быть применена нормализация, могут отображаться со значениями констант, особенно при большом числе отбрасываемых записей. Чтобы снизить вероятность отбрасывания записей, можно задать более высокое значение параметра `pg_stat_statements.max`. Представление `pg_stat_statements_info`, описанное ниже в разделе Подраздел F.45.2, предоставляет статистику об отброшенных записях.

В некоторых случаях запросы с визуально различными текстами могут быть объединены в одну запись `pg_stat_statements`. Обычно это происходит только для семантически равнозначных запросов, но есть небольшая вероятность, что из-за наложений хеша несвязанные запросы могут оказаться объединёнными в одной записи. (Однако это невозможно для запросов, принадлежащих разным пользователям баз данных.)

Так как значение хеша `queryid` вычисляется по представлениям запроса на стадии после разбора, возможна и обратная ситуация: запросы с одинаковым текстом могут оказаться в разных записях, если они получили различные представления по разным причинам, например, из-за изменения `search_path`.

Потребители статистики `pg_stat_statements` могут пожелать использовать в качестве более стабильного и надёжного идентификатора для каждой записи не текст запроса, а `queryid` (возможно, в сочетании с `dbid` и `userid`). Однако важно понимать, что стабильность значения хеша `queryid` гарантируется с ограничениями. Так как этот идентификатор получается из дерева запроса после анализа, его значение будет, помимо прочего, зависеть от внутренних идентификаторов объектов, фигурирующих в этом представлении. С этим связано несколько неинтуитивных следствий. Например, `pg_stat_statements` будет считать два одинаково выглядящих запроса разными, если они обращаются к таблице, которая была удалена, а затем воссоздана между этими запросами. Результат хеширования также чувствителен к различиям в машинной архитектуре и другим особенностям платформы. Более того, не стоит рассчитывать на то, что `queryid` будет оставаться неизменным при обновлении основных версий Postgres Pro.

Можно ожидать, что два сервера, участвующие в репликации на основе воспроизведения физического WAL, будут иметь одинаковые `queryid` для одного запроса. Однако схемы с логической репликацией не гарантируют сохранения идентичности реплик во всех имеющих значение деталях, так что `queryid` не будет полезным идентификатором для накопления показателей стоимости по набору логических реплик. В случае сомнений в том или ином подходе рекомендуется непосредственно протестировать его.

Как правило, можно предположить, что значения `queryid` не меняются между корректирующими релизами Postgres Pro при условии, что экземпляры работают на одной и той же архитектуре и детали метаданных каталога совпадают. Совместимость между корректирующими релизами может быть нарушена только в крайнем случае.

Обозначения параметров, применяемые для замены констант в представляющем запросы тексте, нумеруются, начиная со следующего за последним параметром `$n` в исходном тексте запроса, или с `$1` в отсутствие параметров в нём. Стоит отметить, что в некоторых случаях на эту нумерацию могут влиять скрытые символы параметров. Например, PL/pgSQL применяет такие символы для добавления в запросы значений локальных переменных функций, так что оператор PL/pgSQL вида `SELECT i + 1 INTO j` будет представлен в тексте как `SELECT i + $2`.

Текст, представляющий запрос, сохраняется во внешнем файле на диске и не занимает разделяемую память. Поэтому даже очень объёмный текст запроса может быть сохранён успешно. Однако если в файле накапливается много длинных текстов запросов, он может вырасти до неудобоваримого размера. В качестве решения этой проблемы, `pg_stat_statements` может решить стереть текст запросов, и в результате во всех существующих записях в представлении `pg_stat_statements` в поле `query` окажутся значения NULL, хотя статистика, связанная с каждым `queryid` будет сохранена. Если это происходит и мешает анализу, возможно, стоит уменьшить `pg_stat_statements.max` для предотвращения таких ситуаций.

Показатели `plans` и `calls` не обязательно должны совпадать, так как статистика планирования и выполнения обновляется в конце соответствующей фазы и только при успешном завершении этой фазы. Например, если для оператора успешно выполнилось планирование, но во время выполнения произошла ошибка, изменится только статистика планирования. Если же планирование пропускается по причине использования кешированного плана, увеличивается только счётчик выполнения.

F.45.2. Представление `pg_stat_statements_info`

Статистика самого модуля `pg_stat_statements` собирается и выдаётся через представление с именем `pg_stat_statements_info`. Это представление содержит только одну строку. Столбцы представления показаны в Таблице F.27.

Таблица F.27. Столбцы `pg_stat_statements_info`

Тип столбца Описание
`dealloc bigint` Сколько всего раз в `pg_stat_statements` отбрасывались записи о редко выполняемых операторах, поскольку на обработку поступало больше различных операторов, чем задано в `pg_stat_statements.max`
`stats_reset timestamp with time zone`

Тип столбца Описание
Время последнего сброса всех статистических данных в представлении `pg_stat_statements` .

F.45.3. Функции

`pg_stat_statements_reset(userid Oid, dbid Oid, queryid bigint, minmax_only boolean) returns timestamp with time zone`

Функция `pg_stat_statements_reset` очищает всю статистику, собранную к этому времени модулем `pg_stat_statements` для заданного пользователя (`userid`), базы данных (`dbid`) и запроса (`queryid`). В случае отсутствия одного из параметров для него подразумевается нулевое значение (неприменимое ограничение) и очищается статистика, соответствующая другим параметрам. Если никакой параметр не задан или все параметры имеют нулевое значение (неприменимое), очищается вся статистика. Вместе с очисткой всей статистики в `pg_stat_statements` также сбрасывается статистика в `pg_stat_statements_info`. Когда для параметра `minmax_only` установлено значение `true`, сбрасываются значения только минимального и максимального времени планирования и выполнения оператора (т. е. значения столбцов `min_plan_time`, `max_plan_time`, `min_exec_time` и `max_exec_time`). Значение `minmax_only` по умолчанию — `false`. Время последнего сброса минимальных/максимальных значений можно увидеть в столбце `minmax_stats_since` представления `pg_stat_statements` или в столбце `stats_reset` представления `pg_stat_statements_info`, если соответствующий сброс значений был действительно выполнен. По умолчанию эту функцию могут выполнять только суперпользователи. Другим пользователям можно дать доступ к ней, используя `GRANT`.

`pg_stat_statements(showtext boolean) returns setof record`

Представление `pg_stat_statements` реализовано на базе функции, которая тоже называется `pg_stat_statements`. Клиенты могут вызывать функцию `pg_stat_statements` непосредственно, и могут указать `showtext := false` и получить результат без текста запроса (то есть, выходной аргумент (`OUT`), соответствующий столбцу представления `query`, будет содержать NULL). Эта возможность предназначена для поддержки внешних инструментов, для которых желательно избежать издержек, связанных с получением текстов запросов неопределённой длины. Такие инструменты могут кешировать текст первого запроса, который они получат самостоятельно, как это и делает `pg_stat_statements`, а затем запрашивать тексты запросов только при необходимости. Так как сервер сохраняет тексты запросов в файле, этот подход сокращает объём физического ввода-вывода, порождаемого при постоянном обращении к данным `pg_stat_statements`.

F.45.4. Параметры конфигурации

`pg_stat_statements.max` (`integer`)

Параметр `pg_stat_statements.max` задаёт максимальное число операторов, отслеживаемых модулем (то есть, максимальное число строк в представлении `pg_stat_statements`). Когда на обработку поступает больше, чем заданное число различных операторов, информация о редко выполняемых операторах отбрасывается. В представлении `pg_stat_statements_info` можно увидеть, сколько раз это происходило. Значение по умолчанию — 5000. Этот параметр можно задать только при запуске сервера.

`pg_stat_statements.track` (`enum`)

Параметр `pg_stat_statements.track` определяет, какие операторы будут отслеживаться модулем. Со значением `top` отслеживаются операторы верхнего уровня (те, что непосредственно выполняются клиентами), со значением `all` также отслеживаются вложенные операторы (например, операторы, вызываемые внутри функций), а значение `none` полностью отключает сбор статистики по операторам. Значение по умолчанию — `top`. Изменять этот параметр могут только суперпользователи.

`pg_stat_statements.track_utility` (`boolean`)

Параметр `pg_stat_statements.track_utility` определяет, будет ли этот модуль отслеживать служебные команды. Служебными командами считаются команды, отличные от `SELECT`, `INSERT`, `UPDATE`, `DELETE` и `MERGE`. Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pg_stat_statements.track_planning` (`boolean`)

Параметр `pg_stat_statements.track_planning` определяет, будет ли этот модуль отслеживать операции планирования и их длительность. Включение этого параметра может привести к заметному снижению производительности, особенно когда во множестве одновременных сеансов выполняются операторы с одинаковой структурой запросов, в результате чего эти сеансы пытаются одновременно изменить одни и те же записи в `pg_stat_statements`. Значение по умолчанию — `off` (выкл.). Изменить этот параметр могут только суперпользователи.

`pg_stat_statements.save` (`boolean`)

Параметр `pg_stat_statements.save` определяет, должна ли статистика операторов сохраняться после перезагрузки сервера. Если он отключён (имеет значение `off`), статистика не сохраняется при остановке сервера и не перезагружается при запуске. Значение по умолчанию — `on` (вкл.). Этот параметр можно задать только в `postgresql.conf` или в командной строке сервера.

Этому модулю требуется дополнительная разделяемая память в объёме, пропорциональном `pg_stat_statements.max`. Заметьте, что эта память будет занята при загрузке модуля, даже если `pg_stat_statements.track` имеет значение `none`.

Эти параметры должны задаваться в `postgresql.conf`. Обычное использование выглядит так:

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

compute_query_id = on
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.45.5. Пример вывода

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]---+--------------------------------------------------------------------
query           | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2
calls           | 3000
total_exec_time | 25565.855387
rows            | 3000
hit_percent     | 100.0000000000000000
-[ RECORD 2 ]---+--------------------------------------------------------------------
query           | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
calls           | 3000
total_exec_time | 20756.669379
rows            | 3000
hit_percent     | 100.0000000000000000
-[ RECORD 3 ]---+--------------------------------------------------------------------
query           | copy pgbench_accounts from stdin
calls           | 1
```

```
total_exec_time | 291.865911
rows            | 100000
hit_percent     | 100.0000000000000000
-[ RECORD 4 ]---+--------------------------------------------------------------------
query           | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
calls           | 3000
total_exec_time | 271.232977
rows            | 3000
hit_percent     | 98.8454011741682975
-[ RECORD 5 ]---+--------------------------------------------------------------------
query           | alter table pgbench_accounts add primary key (aid)
calls           | 1
total_exec_time | 160.588563
rows            | 0
hit_percent     | 100.0000000000000000


bench=# SELECT pg_stat_statements_reset(0,0,s.queryid) FROM pg_stat_statements AS s
            WHERE s.query = 'UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE
 bid = $2';

bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]---+--------------------------------------------------------------------
query           | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
calls           | 3000
total_exec_time | 20756.669379
rows            | 3000
hit_percent     | 100.0000000000000000
-[ RECORD 2 ]---+--------------------------------------------------------------------
query           | copy pgbench_accounts from stdin
calls           | 1
total_exec_time | 291.865911
rows            | 100000
hit_percent     | 100.0000000000000000
-[ RECORD 3 ]---+--------------------------------------------------------------------
query           | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
calls           | 3000
total_exec_time | 271.232977
rows            | 3000
hit_percent     | 98.8454011741682975
-[ RECORD 4 ]---+--------------------------------------------------------------------
query           | alter table pgbench_accounts add primary key (aid)
calls           | 1
total_exec_time | 160.588563
rows            | 0
hit_percent     | 100.0000000000000000
-[ RECORD 5 ]---+--------------------------------------------------------------------
query           | vacuum analyze pgbench_accounts
calls           | 1
total_exec_time | 136.448116
rows            | 0
hit_percent     | 99.9201915403032721

bench=# SELECT pg_stat_statements_reset(0,0,0);

bench=# SELECT query, calls, total_exec_time, rows, 100.0 * shared_blks_hit /
```

```
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT 5;
-[ RECORD 1 ]---+--------------------------------------------------
---------------------------
query           | SELECT pg_stat_statements_reset(0,0,0)
calls           | 1
total_exec_time | 0.189497
rows            | 1
hit_percent     |
-[ RECORD 2 ]---+--------------------------------------------------
---------------------------
query           | SELECT query, calls, total_exec_time, rows, $1 * shared_blks_hit /
       +
                |               nullif(shared_blks_hit + shared_blks_read, $2) AS
 hit_percent+
                |          FROM pg_stat_statements ORDER BY total_exec_time DESC LIMIT
 $3
calls           | 0
total_exec_time | 0
rows            | 0
hit_percent     |
```

F.45.6. Авторы

Такахиро Итагаки `<itagaki.takahiro@oss.ntt.co.jp>`. Нормализацию запросов добавил Питер Гейган `<peter@2ndquadrant.com>`.

F.46. pgstattuple — получение статистики на уровне кортежей

Модуль `pgstattuple` предоставляет различные функции для получения статистики на уровне кортежей.

Так как эти функции возвращают подробную информацию, относящуюся к уровню страницы, доступ к ним по умолчанию ограничен. Право `EXECUTE` для них имеет только роль `pg_stat_scan_tables`. Разумеется, суперпользователи могут обойти это ограничение. После того как это расширение установлено, можно поменять права доступа к этим функциям командами `GRANT` и разрешить их выполнение другим пользователям. Однако предпочтительнее будет добавить этих пользователей в роль `pg_stat_scan_tables`.

F.46.1. Функции

`pgstattuple(regclass) returns record`

Функция `pgstattuple` возвращает физическую длину отношения, процент «мёртвых» кортежей и другую информацию. Она может быть полезна для принятия решения о необходимости очистки. В аргументе передаётся имя (возможно, дополненное схемой) или OID целевого отношения. Например:

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

Столбцы результата описаны в Таблице F.28.

Таблица F.28. Столбцы результата `pgstattuple`

Столбец	Тип	Описание
`table_len`	`bigint`	Физическая длина отношения в байтах
`tuple_count`	`bigint`	Количество «живых» кортежей
`tuple_len`	`bigint`	Общая длина «живых» кортежей в байтах
`tuple_percent`	`float8`	Процент «живых» кортежей
`dead_tuple_count`	`bigint`	Количество «мёртвых» кортежей
`dead_tuple_len`	`bigint`	Общая длина «мёртвых» кортежей в байтах
`dead_tuple_percent`	`float8`	Процент «мёртвых» кортежей
`free_space`	`bigint`	Общий объём свободного пространства в байтах
`free_percent`	`float8`	Процент свободного пространства

Примечание

Значение `table_len` всегда будет больше суммы `tuple_len`, `dead_tuple_len` и `free_space`. Разница объясняется фиксированными издержками, внутристраничной таблицей указателей на кортежи и пропусками, добавляемыми для выравнивания кортежей.

Функция `pgstattuple` получает блокировку отношения только для чтения. Таким образом, её результаты отражают не мгновенный снимок; на них будут влиять параллельные изменения.

`pgstattuple` считает кортеж «мёртвым», если `HeapTupleSatisfiesDirty` возвращает false.

`pgstattuple(text) returns record`

Эта функция равнозначна функции `pgstattuple(regclass)` за исключением того, что для неё целевое отношение задаётся в текстовом виде. Данная функция оставлена для обратной совместимости, в будущем она может перейти в разряд устаревших.

`pgstatindex(regclass) returns record`

Функция `pgstatindex` возвращает запись с информацией об индексе типа B-дерево. Например:

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 54.27
leaf_fragmentation | 0
```

Столбцы результата:

Столбец	Тип	Описание
`version`	`integer`	Номер версии B-дерева
`tree_level`	`integer`	Уровень корневой страницы в дереве
`index_size`	`bigint`	Общий объём индекса в байтах
`root_block_no`	`bigint`	Расположение страницы корня (0, если её нет)
`internal_pages`	`bigint`	Количество «внутренних» страниц (верхнего уровня)
`leaf_pages`	`bigint`	Количество страниц на уровне листьев
`empty_pages`	`bigint`	Количество пустых страниц
`deleted_pages`	`bigint`	Количество удалённых страниц
`avg_leaf_density`	`float8`	Средняя плотность страниц на уровне листьев
`leaf_fragmentation`	`float8`	Фрагментация на уровне листьев

Выдаваемый размер индекса (`index_size`) обычно вычисляется по формуле `internal_pages + leaf_pages + empty_pages + deleted_pages` плюс одна страница, так как в нём учитывается и метастраница индекса.

Как и `pgstattuple`, эта функция собирает данные страница за страницей и не следует ожидать, что её результат представляет мгновенный снимок всего индекса.

`pgstatindex(text) returns record`

Эта функция равнозначна функции `pgstatindex(regclass)` за исключением того, что для неё целевое отношение задаётся в текстовом виде. Данная функция оставлена для обратной совместимости, в будущем она может перейти в разряд устаревших.

`pgstatginindex(regclass) returns record`

Функция `pgstatginindex` возвращает запись с информацией об индексе типа GIN. Например:

```
test=> SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+--
version        | 1
pending_pages  | 0
pending_tuples | 0
```

Столбцы результата:

Столбец	**Тип**	**Описание**
`version`	`integer`	Номер версии GIN
`pending_pages`	`integer`	Количество страниц в списке ожидающих обработки
`pending_tuples`	`bigint`	Количество кортежей в списке ожидающих обработки

`pgstathashindex(regclass) returns record`

Функция `pgstathashindex` возвращает запись с информацией о хеш-индексе. Например:

```
test=> select * from pgstathashindex('con_hash_index');
-[ RECORD 1 ]--+-----------------
version        | 4
bucket_pages   | 33081
overflow_pages | 0
bitmap_pages   | 1
unused_pages   | 32455
live_items     | 10204006
dead_items     | 0
free_percent   | 61.8005949100872
```

Столбцы результата:

Столбец	**Тип**	**Описание**
`version`	`integer`	Номер версии HASH
`bucket_pages`	`bigint`	Количество страниц групп
`overflow_pages`	`bigint`	Количество страниц переполнения
`bitmap_pages`	`bigint`	Количество страниц битовой карты
`unused_pages`	`bigint`	Количество неиспользованных страниц

Столбец	**Тип**	**Описание**
`live_items`	`bigint`	Количество «живых» кортежей
`dead_tuples`	`bigint`	Количество «мёртвых» кортежей
`free_percent`	`float`	Процент свободного пространства

`pg_relpages(regclass) returns bigint`

Функция `pg_relpages` возвращает число страниц в отношении.

`pg_relpages(text) returns bigint`

Эта функция равнозначна функции `pg_relpages(regclass)` за исключением того, что для неё целевое отношение задаётся в текстовом виде. Данная функция оставлена для обратной совместимости, в будущем она может перейти в разряд устаревших.

`pgstattuple_approx(regclass) returns record`

Функция `pgstattuple_approx` является более быстрой альтернативой `pgstattuple`, возвращающей приблизительные результаты. В качестве аргумента ей передаётся имя или OID целевого отношения. Например:

```
test=> SELECT * FROM pgstattuple_approx('pg_catalog.pg_proc'::regclass);
-[ RECORD 1 ]--------+-------
table_len            | 573440
scanned_percent      | 2
approx_tuple_count   | 2740
approx_tuple_len     | 561210
approx_tuple_percent | 97.87
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 11996
approx_free_percent  | 2.09
```

Выходные столбцы описаны в Таблице F.29.

Тогда как `pgstattuple` всегда производит полное сканирование таблицы и возвращает точное число живых и мёртвых кортежей (и их размер), а также точный объём свободного пространства, функция `pgstattuple_approx` пытается избежать полного сканирования и возвращает точную статистику только по мёртвым кортежам, а количество и объём живых кортежей, как и объём свободного пространства определяет приблизительно.

Она делает это, пропуская страницы, в которых, согласно карте видимости, есть только видимые кортежи (если для страницы установлен соответствующий бит, предполагается, что она не содержит мёртвых кортежей). Для таких страниц эта функция узнаёт объём свободного пространства из карты свободного пространства и предполагает, что остальное пространство на странице занято живыми кортежами.

На страницах, которые нельзя пропустить, она сканирует каждый кортеж, отражает его наличие и размер в соответствующих счётчиках и суммирует свободное пространство на странице. В конце она оценивает приблизительно общее число живых кортежей, исходя из числа просканированных страниц и кортежей (так же, как VACUUM рассчитывает значение pg_class.reltuples).

Таблица F.29. Столбцы результата `pgstattuple_approx`

Столбец	**Тип**	**Описание**
`table_len`	`bigint`	Физическая длина отношения в байтах (точная)

Столбец	**Тип**	**Описание**
`scanned_percent`	`float8`	Просканированный процент таблицы
`approx_tuple_count`	`bigint`	Количество «живых» кортежей (приблизительное)
`approx_tuple_len`	`bigint`	Общая длина «живых» кортежей в байтах (приблизительная)
`approx_tuple_percent`	`float8`	Процент «живых» кортежей
`dead_tuple_count`	`bigint`	Количество «мёртвых» кортежей (точное)
`dead_tuple_len`	`bigint`	Общая длина «мёртвых» кортежей в байтах (точная)
`dead_tuple_percent`	`float8`	Процент «мёртвых» кортежей
`approx_free_space`	`bigint`	Общий объём свободного пространства в байтах (приблизительный)
`approx_free_percent`	`float8`	Процент свободного пространства

В показанном выше выводе показатели свободного пространства могут не соответствовать выводу `pgstattuple` в точности, потому что карта свободного пространства показывает верное значение, но не гарантируется, что оно будет точным до байта.

F.46.2. Авторы

Тацуо Исии, Сатоши Нагаясу и Абхиджит Менон-Сен

F.47. pg_surgery — проведение операций низкого уровня с данными отношений

Модуль `pg_surgery` предоставляет различные функции для проведения операций с повреждёнными отношениями. Эти функции по природе своей небезопасны, и их использование может повредить данные (или усугубить имеющиеся повреждения БД). Например, эти функции могут нарушить соответствие индексов таблице, повредить ограничения `UNIQUE` или `FOREIGN KEY` или даже сделать видимыми кортежи, чтение которых приведёт к аварийной остановке сервера баз данных. Эти функции следует использовать с большой осторожностью и только в крайнем случае.

F.47.1. Функции

`heap_force_kill(regclass, tid[]) returns void`

Функция `heap_force_kill` помечает «использованные» указатели как «мёртвые», не обращаясь к кортежам. Эта функция предназначена для принудительного удаления кортежей, к которым нельзя обратиться. Например:

```
test=> select * from t1 where ctid = '(0, 1)';
ERROR:  could not access status of transaction 4007513275
DETAIL:  Could not open file "pg_xact/0EED": No such file or directory.

test=# select heap_force_kill('t1'::regclass, ARRAY['(0, 1)']::tid[]);
 heap_force_kill
-----------------

(1 row)

test=# select * from t1 where ctid = '(0, 1)';
(0 rows)
```

`heap_force_freeze(regclass, tid[]) returns void`

Функция `heap_force_freeze` помечает кортежи как замороженные, не обращаясь к кортежам. Эта функция предназначена для получения доступа к кортежам, которые оказались недоступными по причине повреждения информации о видимости или не дают успешно выполнить очистку таблицы по причине повреждения этой информации. Например:

```
test=> vacuum t1;
ERROR:  found xmin 507 from before relfrozenxid 515
CONTEXT:  while scanning block 0 of relation "public.t1"

test=# select ctid from t1 where xmin = 507;
 ctid
-------
 (0,3)
(1 row)

test=# select heap_force_freeze('t1'::regclass, ARRAY['(0, 3)']::tid[]);
 heap_force_freeze
-------------------

(1 row)

test=# select ctid from t1 where xmin = 2;
 ctid
-------
 (0,3)
(1 row)
```

F.47.2. Авторы

Ашутош Шарма (Ashutosh Sharma) `<ashu.coek88@gmail.com>`

F.48. pg_trgm — поддержка определения схожести текста на основе триграмм

Модуль `pg_trgm` предоставляет функции и операторы для определения схожести алфавитно-цифровых строк на основе триграмм, а также классы операторов индексов, поддерживающие быстрый поиск схожих строк.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.48.1. Понятия, связанные с триграммами (или триграфами)

Триграмма — это группа трёх последовательных символов, взятых из строки. Мы можем измерить схожесть двух строк, подсчитав число триграмм, которые есть в обеих. Эта простая идея оказывается очень эффективной для измерения схожести слов на многих естественных языках.

> **Примечание**
>
> `pg_trgm`, извлекая триграммы из строк, игнорирует символы, не относящиеся к словам (не алфавитно-цифровые). При выделении триграмм, содержащихся в строке, считается, что перед каждым словом находятся два пробела, а после — один пробел. Например, из строки «`cat`» выделяется набор триграмм: «`  c`», «` ca`», «`cat`» и «`at `». Из строки «`foo|bar`» выделяются триграммы: «`  f`», «` fo`», «`foo`», «`oo `», «`  b`», «` ba`», «`bar`» и «`ar `».

F.48.2. Функции и операторы

Реализованные в модуле `pg_trgm` функции перечислены в Таблице F.30, а операторы — в Таблице F.31.

Таблица F.30. Функции `pg_trgm`

Функция **Описание**
`similarity` (`text`, `text`) → `real` Возвращает число, показывающее, насколько близки два аргумента. Диапазон результатов — от нуля (это значение указывает, что две строки полностью различны) до одного (это значение указывает, что две строки идентичны).
`show_trgm` (`text`) → `text[]` Возвращает массив всех триграмм в заданной строке. (На практике это редко бывает полезно, кроме как для отладки.)
`word_similarity` (`text`, `text`) → `real` Возвращает число, представляющее наибольшую степень схожести между набором триграмм в первой строке и любым непрерывным фрагментом упорядоченного набора триграмм во второй строке. Подробнее об этом рассказывается ниже.
`strict_word_similarity` (`text`, `text`) → `real` Подобна `word_similarity` , но подгоняет границы фрагментов к границам слов. Так как триграммы не пересекают слова, эта функция фактически выдаёт наибольшую степень схожести между первой строкой и любой непрерывной последовательностью слов во второй строке.
`show_limit` () → `real` Возвращает текущий порог схожести, который использует оператор `%`. Это значение задаёт минимальную схожесть между двумя словами, при которой они считаются настолько близкими, что одно может быть, например, ошибочным написанием другого (*Устаревшая* функция; используйте вместо неё `SHOW pg_trgm.similarity_threshold` .)

Функция Описание
`set_limit` (`real`) → `real` Задаёт текущий порог схожести, который использует оператор `%`. Это значение должно быть в диапазоне от 0 до 1 (по умолчанию 0.3). Возвращает то же значение, что было передано на вход. (*Устаревшая* функция; используйте вместо неё `SET pg_trgm.similarity_threshold`.)

Рассмотрим следующий пример:

```
# SELECT word_similarity('word', 'two words');
 word_similarity
-----------------
             0.8
(1 row)
```

Набор триграмм для первой строки: `{"  w"," wo","wor","ord","rd "}`. Во второй строке упорядоченный набор триграмм: `{"  t"," tw","two","wo ","  w"," wo","wor","ord","rds","ds "}`. Наиболее близкий фрагмент упорядоченного множества триграмм во второй строке: `{"  w"," wo","wor","ord"}`, а коэффициент схожести равен `0.8`.

Эта функция возвращает значение, которое можно примерно воспринимать как максимальную оценку схожести первой строки с любой подстрокой второй строки. Данная функция не добавляет пробелы к границам фрагмента, поэтому совпадение с отдельным словом оценивается выше, чем совпадение с частью слова.

При этом `strict_word_similarity` выбирает последовательность слов во второй строке. В показанном выше примере `strict_word_similarity` выберет последовательность из одного слова `'words'`, которой соответствуют триграммы `{"  w"," wo","wor","ord","rds","ds "}`.

```
# SELECT strict_word_similarity('word', 'two words'), similarity('word', 'words');
 strict_word_similarity | similarity
------------------------+------------
               0.571429 |   0.571429
(1 row)
```

Таким образом, функция `strict_word_similarity` полезна для определения схожести целых слов, а `word_similarity` больше подходит для определения схожести частей слов.

Таблица F.31. Операторы `pg_trgm`

Оператор Описание
`text` `%` `text` → `boolean` Возвращает `true`, если схожесть аргументов выше текущего порога, заданного параметром `pg_trgm.similarity_threshold`.
`text` `<%` `text` → `boolean` Возвращает `true`, если схожесть между набором триграмм в первом аргументе и непрерывным фрагментом упорядоченного набора триграмм во втором превышает уровень схожести, устанавливаемый параметром `pg_trgm.word_similarity_threshold`.
`text` `%>` `text` → `boolean` Коммутирующий оператор для `<%`.
`text` `<<%` `text` → `boolean` Возвращает `true`, если во втором аргументе имеется непрерывный фрагмент упорядоченного набора триграмм, соответствующего границам слов, и его схожесть с набором триграмм первого аргумента превышает уровень схожести, устанавливаемый параметром `pg_trgm.strict_word_similarity_threshold`.

Оператор **Описание**
`text %>> text → boolean` Коммутирующий оператор для `<<%`.
`text <-> text → real` Возвращает «расстояние» между аргументами, то есть один минус значение `similarity()` .
`text <<-> text → real` Возвращает «расстояние» между аргументами, то есть один минус значение `word_similarity()` .
`text <->> text → real` Коммутирующий оператор для `<<->`.
`text <<<-> text → real` Возвращает «расстояние» между аргументами, то есть один минус значение `strict_word_similarity()` .
`text <->>> text → real` Коммутирующий оператор для `<<<->`.

F.48.3. Параметры GUC

`pg_trgm.similarity_threshold` (`real`)

Задаёт текущий порог схожести, который использует оператор `%`. Это значение должно быть в диапазоне от 0 до 1 (по умолчанию 0.3).

`pg_trgm.word_similarity_threshold` (`real`)

Задаёт текущий порог схожести слов, который используют операторы `<%` и `%>`. Это значение должно быть в диапазоне от 0 до 1 (по умолчанию 0.6).

`pg_trgm.strict_word_similarity_threshold` (`real`)

Задаёт текущий порог схожести строго слов, который используют операторы `<<%` и `%>>`. Это значение должно быть в диапазоне от 0 до 1 (по умолчанию 0.5).

F.48.4. Поддержка индексов

Модуль `pg_trgm` предоставляет классы операторов индексов GiST и GIN, позволяющие создавать индекс по текстовым столбцам для очень быстрого поиска по критерию схожести. Эти типы индексов поддерживают вышеописанные операторы схожести и дополнительно поддерживают поиск на основе триграмм для запросов с `LIKE`, `ILIKE`, `~`, `~*` и `=`. В стандартной сборке `pg_trgm` регистр при определении схожести не учитывается. Операторы неравенства не поддерживаются. Обратите внимание, что для операторов равенства эти индексы могут быть не так эффективны, как индексы-B-деревья.

Пример:

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops);
```

или

```
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);
```

Класс операторов GiST `gist_trgm_ops` аппроксимирует набор триграмм в виде сигнатуры битовой карты. В его необязательном целочисленном параметре `siglen` можно задать размер сигнатуры в байтах. Параметр может принимать значения от 1 до 2024, по умолчанию он равен 12. При увеличении размера сигнатуры поиск работает точнее (сканируется меньшая область в индексе и меньше страниц кучи), но сам индекс становится больше.

Пример создания такого индекса с длиной сигнатуры 32 байта:

```
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops(siglen=32));
```

На этот момент у вас будет индекс по столбцу `t`, используя который можно осуществлять поиск по схожести. Пример типичного запроса:

```
SELECT t, similarity(t, 'слово') AS sml
  FROM test_trgm
  WHERE t % 'слово'
  ORDER BY sml DESC, t;
```

Он выдаст все значения в текстовом столбце, которые достаточно схожи со словом *word*, в порядке сортировки от наиболее к наименее схожим. Благодаря использованию индекса, эта операция будет быстрой даже с очень большими наборами данных.

Другой вариант предыдущего запроса:

```
SELECT t, t <-> 'слово' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

Он может быть довольно эффективно выполнен с применением индексов GiST, а не GIN. Обычно он выигрышнее первого варианта только когда требуется получить небольшое количество близких совпадений.

Также вы можете использовать индекс по столбцу `t` для оценки схожести слов условных и оценки схожести слов в строгом смысле. Примеры типичных запросов:

```
SELECT t, word_similarity('слово', t) AS sml
  FROM test_trgm
  WHERE 'слово' <% t
  ORDER BY sml DESC, t;
```

и

```
SELECT t, strict_word_similarity('слово', t) AS sml
  FROM test_trgm
  WHERE 'слово' <<% t
  ORDER BY sml DESC, t;
```

В результате будут возвращены все значения в текстовом столбце, для которых найдется непрерывный фрагмент в упорядоченном наборе триграмм, достаточно схожий с набором триграмм строки *слово*. Данные значения будут отсортированы по порядку от наиболее к наименее схожим. Этот индекс позволит ускорить поиск даже с очень большим объёмом данных.

Другие возможные варианты предыдущих запросов:

```
SELECT t, 'слово' <<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

и

```
SELECT t, 'слово' <<<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

Они могут быть довольно эффективно выполнены с применением индексов GiST, а не GIN.

Начиная с PostgreSQL 9.1, эти типы индексов также поддерживают поиск с операторами `LIKE` и `ILIKE`, например:

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

При таком поиске по индексу сначала из искомой строки извлекаются триграммы, а затем они ищутся в индексе. Чем больше триграмм оказывается в искомой строке, тем более эффективным будет поиск по индексу. В отличие от поиска по B-дереву, искомая строка не должна привязываться к левому краю.

Начиная с PostgreSQL 9.3, индексы этих типов также поддерживают поиск по регулярным выражениям (операторы `~` и `~*`), например:

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

При таком поиске из регулярного выражения извлекаются триграммы, а затем они ищутся в индексе. Чем больше триграмм удаётся извлечь из регулярного выражения, тем более эффективным будет поиск по индексу. В отличие от поиска по B-дереву, искомая строка не должна привязываться к левому краю.

Относительно поиска по регулярному выражению или с `LIKE`, имейте в виду, что при отсутствии триграмм в искомом шаблоне поиск сводится к полному сканирования индекса.

Выбор между индексами GiST и GIN зависит от относительных характеристик производительности GiST и GIN, которые здесь не рассматриваются.

F.48.5. Интеграция с текстовым поиском

Сопоставление триграмм — очень полезный приём в сочетании с применением полнотекстового индекса. В частности это может помочь найти слова, написанные неправильно, которые не будут находиться непосредственно механизмом полнотекстового поиска.

В первую очередь нужно построить дополнительную таблицу, содержащую все уникальные слова в документе:

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM documents');
```

Здесь `documents` — это таблица с текстовым полем `bodytext`, по которому мы будем выполнять поиск. Конфигурация `simple` используется с функцией `to_tsvector` вместо конфигурации для определённого языка по той причине, что нам нужен список исходных (необработанных стеммером) слов.

Затем нужно создать индекс триграмм по столбцу со словами:

```
CREATE INDEX words_idx ON words USING GIN (word gin_trgm_ops);
```

Теперь мы можем использовать запрос `SELECT`, подобный показанному в предыдущем примере, и предлагать варианты исправлений слов, введённых пользователем с ошибками. Кроме того, может быть полезно дополнительно проверить, что выбранные слова также имеют длину, примерно равную длине ошибочных слов.

> **Примечание**
>
> Так как таблица `words` была сформирована как отдельная статическая таблица, её нужно периодически обновлять, чтобы она достаточно хорошо соответствовала набору документов. Постоянно поддерживать её в полностью актуальном состоянии обычно не требуется.

F.48.6. Ссылки

Сайт разработки GiST *http://www.sai.msu.su/~megera/postgres/gist/*

Сайт разработки Tsearch2 *http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/*

F.48.7. Авторы

Олег Бартунов `<oleg@sai.msu.su>`, Москва, Московский Государственный Университет, Россия

Фёдор Сигаев `<teodor@sigaev.ru>`, Москва, ООО «Дельта-Софт», Россия

Александр Коротков `<a.korotkov@postgrespro.ru>`, Москва, Postgres Professional, Россия

Документация: Кристофер Кингс-Линн

Разработку этого модуля спонсировало ООО «Дельта-Софт», г. Москва, Россия.

F.49. pg_tsparser — текстовый поиск

`pg_tsparser` — это расширение Postgres Pro для текстового поиска. Это расширение меняет стандартную стратегию разбора текста для слов, включающих:

- подчёркивания
- цифры и буквы, разделённые знаком минуса

В дополнение к отдельным частям слова, возвращаемых по умолчанию, `pg_tsparser` также возвращает всё слово.

F.49.1. Установка и настройка

Расширение `pg_tsparser` входит в состав Postgres Pro. Чтобы задействовать `pg_tsparser`, установив Postgres Pro, создайте расширение `pg_tsparser` в каждой базе данных, где вы планируете его использовать:

```
CREATE EXTENSION pg_tsparser;
```

Активировав `pg_tsparser`, вы можете создавать собственные конфигурации текстового поиска. Вместе с `pg_tsparser` можно использовать любой имеющийся словарь.

Например, так можно создать конфигурацию `english_ts` для английского языка:

```
CREATE TEXT SEARCH CONFIGURATION english_ts (
    PARSER = tsparser
);

COMMENT ON TEXT SEARCH CONFIGURATION english_ts IS 'text search configuration for
 english language';

ALTER TEXT SEARCH CONFIGURATION english_ts
    ADD MAPPING FOR email, file, float, host, hword_numpart, int,
    numhword, numword, sfloat, uint, url, url_path, version
    WITH simple;

ALTER TEXT SEARCH CONFIGURATION english_ts
    ADD MAPPING FOR asciiword, asciihword, hword_asciipart,
    word, hword, hword_part
    WITH english_stem;
```

F.49.2. Примеры

Следующие примеры иллюстрируют различия в результатах поиска, возвращаемых анализатором текста `pg_tsparser` и стандартным анализатором:

```
SELECT to_tsvector('english', 'pg_trgm') as def_parser,
       to_tsvector('english_ts', 'pg_trgm')  as new_parser;
   def_parser    |         new_parser
-----------------+-----------------------------
 'pg':1 'trgm':2 | 'pg':2 'pg_trgm':1 'trgm':3
(1 row)

SELECT to_tsvector('english', '123-abc') as def_parser,
       to_tsvector('english_ts', '123-abc')  as new_parser;
   def_parser    |         new_parser
-----------------+-----------------------------
 '123':1 'abc':2 | '123':2 '123-abc':1 'abc':3
(1 row)
```

```
SELECT to_tsvector('english', 'rel-3.2-A') as def_parser,
       to_tsvector('english_ts', 'rel-3.2-A')  as new_parser;
    def_parser     |          new_parser
-------------------+-------------------------------
 '-3.2':2 'rel':1 | '3.2':3 'rel':2 'rel-3.2-a':1
(1 row)
```

См. также

CREATE TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION

F.49.3. Авторы

Postgres Professional, Москва, Россия

F.50. pg_variables — функции для работы с переменными различных типов

Модуль `pg_variables` содержит функции для работы с переменными различных типов. Созданные переменные существуют в течение текущего пользовательского сеанса.

F.50.1. Установка

Расширение `pg_variables` включено в состав Postgres Pro. Установив Postgres Pro, вы должны выполнить команду CREATE EXTENSION, чтобы подготовить `pg_variables` к работе, следующим образом:

```
CREATE EXTENSION pg_variables;
```

F.50.2. Использование

Модуль `pg_variables` содержит функции, позволяющие создавать переменные скалярных типов, переменные-записи и переменные-массивы, читать их и управлять ими. Синтаксис и описания функций представлены в следующих разделах:

- В Подразделе F.50.3.1 описываются функции для скалярных переменных.
- В Подразделе F.50.3.2 описываются функции для переменных-коллекций записей.
- В Подразделе F.50.3.3 описываются функции для переменных-массивов.
- В Подразделе F.50.3.4 описываются функции для общих переменных-коллекций.
- В Подразделе F.50.3.6 перечисляются функции, позволяющие управлять всеми переменными в текущем сеансе.

Разобранные примеры использования вы можете найти в Подразделе F.50.5.

F.50.2.1. Использование транзакционных переменных

По умолчанию переменные создаются как нетранзакционные. Успешно созданная переменная продолжает существовать на протяжении всего сеанса, вне зависимости от возможных откатов транзакций. Например:

```
SELECT pgv_set('vars', 'int1', 101);
BEGIN;
SELECT pgv_set('vars', 'int2', 102);
ROLLBACK;

SELECT * FROM pgv_list() order by package, name;
 package | name | is_transactional
---------+------+------------------
 vars    | int1 | f
 vars    | int2 | f
```

Если вы хотите использовать переменную с поддержкой транзакций и точек сохранения, передайте дополнительный флаг `is_transactional` в последнем параметре функции, создающей переменную:

```
BEGIN;
SELECT pgv_set('vars', 'trans_int', 101, true);
SAVEPOINT sp1;
SELECT pgv_set('vars', 'trans_int', 102, true);
ROLLBACK TO sp1;
COMMIT;
SELECT pgv_get('vars', 'trans_int', NULL::int);
 pgv_get
---------
     101
```

Вы должны указывать флаг `is_transactional` при каждом изменении значения транзакционной переменной с помощью функций `pgv_set()` и `pgv_insert()`. В противном случае произойдёт ошибка. Другим функциям передавать этот флаг не нужно.

```
SELECT pgv_insert('pack', 'var_record', row(123::int, 'text'::text), true);

SELECT pgv_insert('pack', 'var_record', row(456::int, 'another text'::text));
ERROR:  variable "var_record" already created as TRANSACTIONAL

SELECT pgv_delete('pack', 'var_record', 123::int);
```

Если вызовы функций `pgv_free()` или `pgv_remove()` откатываются, затрагиваемые транзакционные переменные восстанавливаются, в отличие от нетранзакционных, которые удаляются необратимо. Например:

```
SELECT pgv_set('pack', 'var_reg', 123);
SELECT pgv_set('pack', 'var_trans', 456, true);
BEGIN;
SELECT pgv_free();
ROLLBACK;
SELECT * FROM pgv_list();
 package |   name    | is_transactional
---------+-----------+------------------
 pack    | var_trans | t
```

F.50.3. Функции

F.50.3.1. Скалярные переменные

Следующие функции поддерживают скалярные переменные:

Функция	Возвращает
`pgv_set(package text, name text, value anynonarray, is_transactional bool default false)`	`void`
`pgv_get(package text, name text, var_type anynonarray, strict bool default true)`	`anynonarray`

Для функции `pgv_get()` вы должны сначала создать пакет и переменную, воспользовавшись функцией `pgv_set()`. Если указанный пакет или переменная не существуют, происходит ошибка:

```
SELECT pgv_get('vars', 'int1', NULL::int);
ERROR:  unrecognized package "vars"
```

```
SELECT pgv_get('vars', 'int1', NULL::int);
ERROR:  unrecognized variable "int1"
```

Функция `pgv_get()` проверяет тип переменной. Если заданный тип не соответствует типу переменной, выдаётся ошибка:

```
SELECT pgv_get('vars', 'int1', NULL::text);
ERROR:  variable "int1" requires "integer" value
```

F.50.3.2. Коллекции записей

Следующие функции поддерживают коллекции переменных типа запись.

Функция	Возвращает	Описание
`pgv_insert(package text, name text, r record, is_`	`void`	Вставляет запись в переменную-коллекцию для заданного

Функция	Возвращает	Описание
`transactional bool default false)`		пакета. Если пакет или переменная не существуют, они создаются автоматически. Первый столбец записи `r` — первичный ключ. Если запись с таким же первичным ключом уже существует или эта переменная-коллекция имеет другую структуру, выдаётся ошибка.
`pgv_update(package text, name text, r record)`	`boolean`	Изменяет запись с соответствующим первичным ключом (он задаётся в первом столбце `r`). Возвращает `true`, если запись была найдена. Если эта переменная-коллекция имеет другую структуру, выдаётся ошибка.
`pgv_delete(package text, name text, value anynonarray)`	`boolean`	Удаляет запись с соответствующим первичным ключом (он задаётся в первом столбце `r`). Возвращает `true`, если запись была найдена, или `false` в противном случае.
`pgv_select(package text, name text)`	`set of records`	Возвращает записи из переменной-коллекции.
`pgv_select(package text, name text, value anynonarray)`	`record`	Возвращает записи с соответствующими первичными ключами (первичный ключ задаётся в первом столбце `r`).
`pgv_select(package text, name text, value anyarray)`	`set of records`	Возвращает записи из переменных-коллекций с соответствующими первичными ключами (первичный ключ задаётся в первом столбце `r`).

Для использования функций `pgv_update()`, `pgv_delete()` и `pgv_select()` вы должны сначала создать пакет и переменную, воспользовавшись функцией `pgv_insert()`. Тип переменной и тип записи должны быть одинаковыми, иначе выдаётся ошибка.

F.50.3.3. Массивы

Следующие функции поддерживают переменные-массивы:

Функция	Возвращает
`pgv_set(package text, name text, value anyarray, is_transactional bool default false)`	`void`
`pgv_get(package text, name text, var_type anyarray, strict bool default true)`	`anyarray`

Инструкции по использованию этих функций совпадают с приведёнными в Подразделе F.50.3.1 для скалярных переменных.

F.50.3.4. Общие коллекции

Следующие функции поддерживают общие переменные-коллекции:

Функция	**Возвращает**	**Описание**
`pgv_set_elem(package text, name text, key int, value anyelement, is_transactional bool default false)` `pgv_set_elem(package text, name text, key text, value anyelement, is_transactional bool default false)`	`void`	Устанавливает значение для элемента с ключом `key` переменной-коллекции `name` в пакете `package`. Если пакет или переменная не существуют, они создаются автоматически. Внутри одной коллекции разрешены только ключи одного типа. Аргумент `key` может быть типа `int` или `text`. Если элемент с указанным ключом уже существует, его значение изменяется на новое. Параметр `is_transactional` показывает, является ли новая переменная транзакционной, и по умолчанию имеет значение `false`. Таким образом, если переменная уже существует, она должна быть транзакционной/нетранзакционной, в зависимости от значения `is_transactional` , в противном случае возникает ошибка. Если коллекция уже существует и тип его значения не соответствует типу нового значения, также возникает ошибка.
`pgv_get_elem(package text, name text, key int, val_type anyelement)` `pgv_get_elem(package text, name text, key text, val_type anyelement)`	`anyelement`	Возвращает значение элемента с ключом `key` переменной-коллекции `name` в пакете `package`. Если в этой коллекции нет элемента с указанным ключом, возвращается NULL. Аргумент `key` может быть типа `int` или `text`. Если пакет или переменная не существует или указанная переменная не является коллекцией, возникает ошибка. Аргумент `val_type` необходим для правильного определения типа возвращаемого значения.
`pgv_remove_elem(package text, name text, key int)` `pgv_remove_elem(package text, name text, key text)`	`void`	Удаляет элемент с ключом `key` переменной-коллекции `name` из пакета `package`. Если в этой коллекции нет элемента с указанным ключом, ничего не происходит. Аргумент `key` может быть типа `int` или `text`. Если пакет или переменная не существует или указанная пере-

Функция	Возвращает	Описание
		менная не является коллекции, возникает ошибка.

Важно

Коллекции, созданные функциями `pgv_set_elem()` и `pgv_insert()`, считаются несовместимыми.

F.50.3.5. Итераторы

Следующие функции предназначены для использования итераторов, чтобы передвигаться по переменным-коллекциям. Эти функции работают с коллекциями, созданными функциями `pgv_set_elem()` и `pgv_insert()`.

Функция	Возвращает	Описание
`pgv_first(package text, name text, key_type anyelement)`	`anyelement`	Возвращает первый ключ из переменной-коллекции. Коллекции сортируются по ключу в порядке возрастания. Параметр `key_type` необходим для определения возвращаемого значения. Если переданное значение `name` не является именем переменной-коллекции, возникает ошибка.
`pgv_last(package text, name text, key_type anyelement)`	`anyelement`	Возвращает последний ключ из переменной-коллекции. Коллекции сортируются по ключу в порядке возрастания. Параметр `key_type` необходим для определения возвращаемого значения. Если переданное значение `name` не является именем переменной-коллекции, возникает ошибка.
`pgv_next(package text, name text, key anyelement)`	`anyelement`	Возвращает следующий ключ из переменной-коллекции. Переданный ключ `key` может отсутствовать в коллекции. Возвращает NULL, если используется для последнего ключа в коллекции. Коллекции сортируются по ключу в порядке возрастания. Если переданное значение `name` не является именем переменной-коллекции, возникает ошибка.
`pgv_prior(package text, name text, key anyelement)`	`anyelement`	Возвращает предыдущий ключ из переменной-коллекции. Переданный ключ `key` может отсутствовать в коллекции. Возвращает NULL, если используется для первого ключа в коллекции. Коллекции сортируют-

Функция	Возвращает	Описание
		ся по ключу в порядке возрастания. Если переданное значение `name` не является именем переменной-коллекции, возникает ошибка.
`pgv_count(package text, name text)`	`integer`	Возвращает количество элементов в коллекции. Если переданное значение `name` не является именем переменной-коллекции, возникает ошибка.

F.50.3.6. Функции разного назначения

Функция	Возвращает	Описание
`pgv_exists(package text, name text)`	`bool`	Возвращает `true`, если существуют пакет и переменная, или `false` в противном случае.
`pgv_exists(package text)`	`bool`	Возвращает `true`, если существуют пакет и переменная, или `false` в противном случае.
`pgv_remove(package text, name text)`	`void`	Удаляет переменную с заданным именем. Указанный пакет и переменная должны существовать, иначе выдаётся ошибка.
`pgv_remove(package text)`	`void`	Удаляет заданный пакет и все его переменные. Указанный пакет должен существовать, иначе выдаётся ошибка.
`pgv_free()`	`void`	Удаляет все пакеты и переменные.
`pgv_list()`	`table(package text, name text, is_transactional bool)`	Выводит список всех существующих переменных с именем соответствующего пакета и признаком того, что переменная транзакционная.
`pgv_stats()`	`table(package text, allocated_memory bigint)`	Возвращает список созданных пакетов и объём памяти, используемый переменными, в байтах. Если вы используете транзакционные переменные, в этот список также включаются все удалённые пакеты, которые могут быть восстановлены командой `ROLLBACK`. Эта функция поддерживается только в Postgres Pro версии 9.6 и новее.

F.50.3.7. Устаревшие функции

F.50.3.7.1. Целочисленные переменные

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	**Возвращает**
`pgv_set_int(package text, name text, value int, is_transactional bool default false)`	`void`
`pgv_get_int(package text, name text, strict bool default true)`	`int`

F.50.3.7.2. Текстовые переменные

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	**Возвращает**
`pgv_set_text(package text, name text, value text, is_transactional bool default false)`	`void`
`pgv_get_text(package text, name text, strict bool default true)`	`text`

F.50.3.7.3. Числовые переменные

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	**Возвращает**
`pgv_set_numeric(package text, name text, value numeric, is_transactional bool default false)`	`void`
`pgv_get_numeric(package text, name text, strict bool default true)`	`numeric`

F.50.3.7.4. Переменные даты/времени

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	**Возвращает**
`pgv_set_timestamp(package text, name text, value timestamp, is_transactional bool default false)`	`void`
`pgv_get_timestamp(package text, name text, strict bool default true)`	`timestamp`

F.50.3.7.5. Переменные даты/времени с часовым поясом

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	**Возвращает**
`pgv_set_timestamptz(package text, name text, value timestamptz, is_transactional bool default false)`	`void`
`pgv_get_timestamptz(package text, name text, strict bool default true)`	`timestamptz`

F.50.3.7.6. Переменные даты

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	**Возвращает**
`pgv_set_date(package text, name text, value date, is_transactional bool default false)`	`void`
`pgv_get_date(package text, name text, strict bool default true)`	`date`

F.50.3.7.7. Переменные Jsonb

Следующие функции считаются устаревшими. Используйте вместо них универсальные функции для скалярных переменных.

Функция	**Возвращает**
`pgv_set_jsonb(package text, name text, value jsonb, is_transactional bool default false)`	`void`
`pgv_get_jsonb(package text, name text, strict bool default true)`	`jsonb`

F.50.4. Важные замечания

F.50.4.1. Правила сортировки в коллекциях

Коллекции хранятся в порядке возрастания. Для ключей типа `text` потребуется правило сортировки для определения порядка элементов. Если при вставке первого элемента коллекции правило сортировки не указано, используется правило сортировки по умолчанию. В противном случае используется указанное правило сортировки.

F.50.4.2. Курсоры для функций, возвращающих множества

Все функции, возвращающие множества, за исключением `pgv_select(package,variable)`, фиксируют свои результаты при выполнении первой команды `FETCH` из курсора, и на них не влияют дальнейшие действия с данными.

Результаты `pgv_select(package,variable)` принимаются динамически, и на них влияют транзакции/изменения в коллекции. Для функции `pgv_select()`, вызываемой для транзакционной коллекции, снимок коллекции просматривается курсорами при выполнении первой команды `FETCH`, но с учётом изменений, которые были сделаны в этой транзакции и в зафиксированных подтранзакциях.

F.50.5. Примеры

Определение скалярных переменных с помощью функции `pgv_set()` и получение их значений с помощью `pgv_get()`:

```
SELECT pgv_set('vars', 'int1', 101);
SELECT pgv_set('vars', 'int2', 102);
SELECT pgv_set('vars', 'text1', 'text variable'::text);

SELECT pgv_get('vars', 'int1', NULL::int);
 pgv_get
-------------
```

```
         101

SELECT pgv_get('vars', 'int2', NULL::int);
 pgv_get
-------------
         102

SELECT pgv_get('vars', 'text1', NULL::text);
 pgv_get
---------------
 text variable
```

В предположении, что создана следующая таблица `tab`, рассмотрите несколько примеров с переменными-записями:

```
CREATE TABLE tab (id int, t varchar);
INSERT INTO tab VALUES (0, 'str00'), (1, 'str11');
```

Для работы с переменными-записями можно использовать следующие функции:

```
SELECT pgv_insert('vars', 'r1', tab) FROM tab;

SELECT pgv_select('vars', 'r1');
 pgv_select
------------
 (1,str11)
 (0,str00)

SELECT pgv_select('vars', 'r1', 1);
 pgv_select
------------
 (1,str11)

SELECT pgv_select('vars', 'r1', 0);
 pgv_select
------------
 (0,str00)

SELECT pgv_select('vars', 'r1', ARRAY[1, 0]);
 pgv_select
------------
 (1,str11)
 (0,str00)

SELECT pgv_delete('vars', 'r1', 1);

SELECT pgv_select('vars', 'r1');
 pgv_select
------------
 (0,str00)
```

Проанализируйте поведение транзакционной переменной `var_text` при изменении до и после точки сохранения:

```
SELECT pgv_set('pack', 'var_text', 'before transaction block'::text, true);
BEGIN;
SELECT pgv_set('pack', 'var_text', 'before savepoint'::text, true);
SAVEPOINT sp1;
SELECT pgv_set('pack', 'var_text', 'savepoint sp1'::text, true);
SAVEPOINT sp2;
```

```
SELECT pgv_set('pack', 'var_text', 'savepoint sp2'::text, true);
RELEASE sp2;
SELECT pgv_get('pack', 'var_text', NULL::text);
    pgv_get
---------------
 savepoint sp2

ROLLBACK TO sp1;
SELECT pgv_get('pack', 'var_text', NULL::text);
     pgv_get
------------------
 before savepoint

ROLLBACK;
SELECT pgv_get('pack', 'var_text', NULL::text);
         pgv_get
--------------------------
 before transaction block
```

Если вы создадите транзакционную переменную после команд `BEGIN` или `SAVEPOINT` и затем произойдёт откат к предыдущему состоянию, эта переменная будет удалена:

```
BEGIN;
SAVEPOINT sp1;
SAVEPOINT sp2;
SELECT pgv_set('pack', 'var_int', 122, true);
RELEASE SAVEPOINT sp2;
SELECT pgv_get('pack', 'var_int', NULL::int);
pgv_get
---------
     122

ROLLBACK TO sp1;
SELECT pgv_get('pack','var_int', NULL::int);
ERROR:  unrecognized variable "var_int"
COMMIT;
```

Просмотреть имеющиеся пакеты и переменные:

```
SELECT * FROM pgv_list() ORDER BY package, name;
 package |   name   | is_transactional
---------+----------+------------------
 pack    | var_text | t
 vars    | int1     | f
 vars    | int2     | f
 vars    | r1       | f
 vars    | text1    | f
```

Получение объёма памяти, занятой переменными, в байтах:

```
SELECT * FROM pgv_stats() ORDER BY package;
 package | allocated_memory
---------+------------------
 pack    |            16384
 vars    |            32768
```

Удаление избранных переменных или пакетов:

```
SELECT pgv_remove('vars', 'int1');
SELECT pgv_remove('vars');
```

Удаление всех пакетов и переменных:

```
SELECT pgv_free();
```

В этих примерах показано использование переменных-коллекций и функций итераторов:

```
sql
SELECT pgv_set_elem('pack', 'var', 1, 1);
SELECT pgv_set_elem('pack', 'var', 5, 5);
SELECT pgv_set_elem('pack', 'var', 10, 10);

SELECT pgv_first('pack', 'var', NULL::int);
 pgv_first
-----------
         1

SELECT pgv_last('pack', 'var', NULL::int);
 pgv_last
----------
       10

SELECT pgv_next('pack', 'var', pgv_first('pack', 'var', NULL::int));
 pgv_next
----------
        5

SELECT pgv_prior('pack', 'var', pgv_last('pack', 'var', NULL::int));
 pgv_prior
-----------
         5

SELECT pgv_prior('pack', 'var', pgv_first('pack', 'var', NULL::int));
 pgv_prior
-----------

SELECT pgv_next('pack', 'var', pgv_last('pack', 'var', NULL::int));
 pgv_prior
-----------

SELECT pgv_next('pack', 'var', 3);
 pgv_next
----------
        5

SELECT pgv_prior('pack', 'var', 3);
 pgv_prior
-----------
         1

SELECT pgv_get_elem('pack', 'var', pgv_last('pack', 'var', NULL::int), NULL::int);
 pgv_get_elem
--------------
           10

SELECT pgv_remove_elem('pack', 'var', pgv_last('pack', 'var', NULL::int));
 pgv_remove_elem
-----------------
```

```
SELECT pgv_get_elem('pack', 'var', pgv_last('pack', 'var', NULL::int), NULL::int);
 pgv_get_elem
--------------
            5
(1 row)
```

Эти примеры показывают, как правила сортировки влияют на порядок элементов в коллекции, а также как перебирать всю коллекцию с помощью циклов PL/pgSQL:

```
sql
SELECT pgv_set_elem('pack', 'var1', 'а' COLLATE "ru_RU", 'а'::text);
SELECT pgv_set_elem('pack', 'var1', 'д' COLLATE "ru_RU", 'д'::text);
SELECT pgv_set_elem('pack', 'var1', 'е' COLLATE "ru_RU", 'е'::text);
SELECT pgv_set_elem('pack', 'var1', 'ё' COLLATE "ru_RU", 'ё'::text);
SELECT pgv_set_elem('pack', 'var1', 'ж' COLLATE "ru_RU", 'ж'::text);
SELECT pgv_set_elem('pack', 'var1', 'я' COLLATE "ru_RU", 'я'::text);
DO
$$
DECLARE
        iter text;
BEGIN
        iter := pgv_first('pack', 'var1', NULL::text);
        WHILE iter IS NOT NULL LOOP
          RAISE NOTICE '%', pgv_get_elem('pack', 'var1', iter, NULL::text);
          iter := pgv_next('pack', 'var1', iter);
        END LOOP;
END;
$$;
NOTICE:  а
NOTICE:  д
NOTICE:  е
NOTICE:  ё
NOTICE:  ж
NOTICE:  я

SELECT pgv_set_elem('pack', 'var2', 'а' COLLATE "C", 'а'::text);
SELECT pgv_set_elem('pack', 'var2', 'д' COLLATE "C", 'д'::text);
SELECT pgv_set_elem('pack', 'var2', 'е' COLLATE "C", 'е'::text);
SELECT pgv_set_elem('pack', 'var2', 'ё' COLLATE "C", 'ё'::text);
SELECT pgv_set_elem('pack', 'var2', 'ж' COLLATE "C", 'ж'::text);
SELECT pgv_set_elem('pack', 'var2', 'я' COLLATE "C", 'я'::text);
DO
$$
DECLARE
        iter text;
BEGIN
        iter := pgv_first('pack', 'var2', NULL::text);
        WHILE iter IS NOT NULL LOOP
          RAISE NOTICE '%', pgv_get_elem('pack', 'var2', iter, NULL::text);
          iter := pgv_next('pack', 'var2', iter);
        END LOOP;
END;
$$;
NOTICE:  а
NOTICE:  д
NOTICE:  е
NOTICE:  ж
NOTICE:  я
NOTICE:  ё
```

F.50.6. Авторы

Postgres Professional, Москва, Россия

F.51. pg_visibility — информация из карты видимости и вспомогательные функции

Модуль `pg_visibility` даёт возможность исследовать для определённой таблицы карту видимости (Visibility Map, VM) и информацию о видимости на уровне страниц. Он также предоставляет функции для проверки целостности карты видимости и принудительного её пересоздания.

Для хранения информации о видимости на уровне страниц применяются по три различных бита. Бит полной видимости в карте показывает, что каждый кортеж в соответствующей странице отношения является видимым для всех текущих и будущих транзакций. Бит полной заморозки в карте видимости показывает, что все кортежи в данной странице являются замороженными; то есть никакой операции очистки в будущем не придётся обрабатывать эту страницу, пока в ней не будет добавлен, изменён, удалён или заблокирован кортеж. Бит `PD_ALL_VISIBLE` в заголовке страницы имеет то же значение, что и бит полной видимости в карте видимости, но он хранится в самой странице данных, а не в отдельной структуре данных. В обычной ситуации эти два бита будут согласованы, но бит полной видимости в странице иногда может быть установлен, тогда как в карте видимости он оказывается сброшенным при восстановлении после сбоя. Считываемые значения могут также различаться, если они подвергаются изменению в промежутке между обращениями `pg_visibility` к карте видимости и к странице данных. Разумеется, эти биты также могут различаться при событиях, приводящих к разрушению данных.

Функции, выдающие информацию о битах `PD_ALL_VISIBLE`, более дорогостоящие, чем те, что обращаются только к карте видимости, так как они должны читать блоки отношения, а не только карту видимости (которая намного меньше). Дорогостоящими являются также и функции, проверяющие блоки данных отношения.

F.51.1. Функции

`pg_visibility_map(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean) returns record`

Возвращает биты полной видимости и полной заморозки в карте видимости для указанного блока заданного отношения.

`pg_visibility(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns record`

Возвращает биты полной видимости и полной заморозки в карте видимости для указанного блока заданного отношения, а также бит `PD_ALL_VISIBLE` этого блока.

`pg_visibility_map(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean) returns setof record`

Возвращает биты полной видимости и полной заморозки в карте видимости для каждого блока заданного отношения.

`pg_visibility(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns setof record`

Возвращает биты полной видимости и полной заморозки в карте видимости для каждого блока заданного отношения, а также бит `PD_ALL_VISIBLE` каждого блока.

`pg_visibility_map_summary(relation regclass, all_visible OUT bigint, all_frozen OUT bigint) returns record`

Возвращает число полностью видимых страниц и полностью замороженных страниц в отношении, согласно карте видимости.

`pg_check_frozen(relation regclass, t_ctid OUT tid) returns setof tid`

Возвращает идентификаторы TID незамороженных кортежей в страницах, помеченных как полностью замороженные в карте видимости. Если эта функция возвращает непустой набор TID, карта видимости испорчена.

`pg_check_visible(relation regclass, t_ctid OUT tid) returns setof tid`

Возвращает идентификаторы TID не полностью видимых кортежей в страницах, помеченных как полностью видимые в карте видимости. Если эта функция возвращает непустой набор TID, карта видимости испорчена.

`pg_truncate_visibility_map(relation regclass) returns void`

Аннулирует карту видимости для заданного отношения. Эта функция полезна, если вы считаете, что карта видимости для указанного отношения испорчена, и хотите принудительно пересоздать её. Первая же команда `VACUUM`, выполняемая с данным отношением после этой функции, просканирует все страницы в отношении и пересоздаст карту видимости. (Пока это не произойдёт, для запросов карта видимости будет выглядеть как полностью нулевая.)

По умолчанию эти функции разрешено выполнять только суперпользователям и ролям с правами роли `pg_stat_scan_tables`, за исключением `pg_truncate_visibility_map(relation regclass)`, которую могут выполнять только суперпользователи.

F.51.2. Автор

Роберт Хаас `<rhaas@postgresql.org>`

F.52. pg_wait_sampling — периодический сбор статистики по событиям ожидания

`pg_wait_sampling` — это расширение Postgres Pro для периодического сбора статистики по событиям ожидания.

Начиная с версии 9.6, Postgres Pro Standard выдаёт информацию о текущих событиях ожидания для определённых процессов. Однако чтобы получить полезную общую статистику активности сервера, необходимо снимать показания по событиям ожидания неоднократно. Расширение `pg_wait_sampling` автоматизирует снятие показаний по событиям ожидания, запуская специальный рабочий процесс. Когда расширение `pg_wait_sampling` включено, вы можете получать следующую периодически собираемую информацию:

- *История ожиданий* — список событий ожидания для недавних процессов, с отметками времени.
- *Профиль ожиданий* — количество наблюдавшихся событий ожидания для всех процессов (в разрезе типов событий).
- Текущие события ожидания для всех процессов, включая фоновые рабочие процессы.

Используя `pg_wait_sampling`, можно выявить проблемы зависимостей для запросов, выполняющихся дольше, чем ожидается. Вы можете видеть процессы в состоянии ожидания в каждый момент времени и анализировать статистику событий ожидания. Список возможных событий ожидания приведён в Таблице 26.4.

В сочетании с `pg_stat_statements` это расширение может также предоставлять статистику по каждому запросу.

См. также

Просмотр статистики по событиям ожидания

Справка

F.52.1. Установка

Расширение `pg_wait_sampling` входит в состав Postgres Pro Standard и не имеет особых требований.

Чтобы включить `pg_wait_sampling`, выполните следующее:

1. Добавьте `pg_wait_sampling` в переменную `shared_preload_libraries` в файле `postgresql.conf`:

   ```
   shared_preload_libraries = 'pg_wait_sampling'
   ```

   > **Важно**
   >
   > При использовании `pg_wait_sampling` с расширением `pg_stat_statements` поместите `pg_stat_statements` перед `pg_wait_sampling` в переменной `shared_preload_libraries`. Это гарантирует, что расширение `pg_stat_statements` не перезапишет идентификаторы запросов, которые используются `pg_wait_sampling`.

2. Создайте расширение `pg_wait_sampling`, выполнив следующий запрос:

   ```
   CREATE EXTENSION pg_wait_sampling;
   ```

3. Перезапустите сервер. Так как расширению `pg_wait_sampling` требуется дополнительная разделяемая память и фоновый рабочий процесс, вы должны перезапустить сервер после добавления или удаления `pg_wait_sampling`.

После создания расширения `pg_wait_sampling` начинает собирать статистику по событиям ожидания для каждого процесса. Вы можете по своему усмотрению изменить частоту снятия показаний и настроить режимы сбора статистики, используя переменные GUC.

Если вы хотите остановить сбор статистики, удалите расширение `pg_wait_sampling`, уберите `pg_wait_sampling` из переменной `shared_preload_libraries` и перезагрузите сервер.

F.52.2. Использование

F.52.2.1. Просмотр статистики по событиям ожидания

Когда `pg_wait_sampling` работает, показания снимаются по умолчанию каждые 10 мс. Обратиться к собранной статистике вы можете через набор функций и представлений. Если вы замечаете, что некоторые запросы «зависли» или выполняются дольше, чем ожидается, вы можете увидеть все текущие события ожидания в представлении `pg_wait_sampling_current`:

```
SELECT * FROM pg_wait_sampling_current;
```

Возвращаемая статистика охватывает все активные процессы, включая фоновые рабочие процессы. Чтобы отфильтровать это представление по одному процессу, воспользуйтесь функцией `pg_wait_sampling_get_current()` и передайте ей идентификатор интересующего процесса:

```
SELECT * FROM pg_wait_sampling_get_current(pid);
```

Чтобы лучше понимать динамику недавних событий ожидания, можно обратиться к представлению `pg_wait_sampling_history`, показывающему историю событий ожидания для последних процессов:

```
SELECT * FROM pg_wait_sampling_history;
```

Это представление выдаёт показания по событиям ожидания с отметками времени. История ожиданий реализована как кольцевой буфер в памяти, размер которого по умолчанию ограничен 5000 показаниями. Чтобы сохранить историю за больший период времени, вы можете изменить переменную `pg_wait_sampling.history_size` либо подключить клиентское приложение, которое будет периодически считывать историю ожиданий и сохранять её в локальном хранилище.

Для отслеживания событий ожидания в вашей базе данных в течение длительного времени используйте представление `pg_wait_sampling_profile`:

```
SELECT * FROM pg_wait_sampling_profile;
```

Профиль ожиданий сохраняется как хеш-таблица в памяти, накапливающая показатели по отдельным процессам и по событиям ожидания. Вы можете сбросить профиль ожиданий, вызвав функцию `pg_wait_sampling_reset_profile()`. Если вы подключите к вашей базе данных клиента, который будет периодически сбрасывать данные событий ожидания и очищать профиль, вы можете сохранять и анализировать концентрацию событий ожидания во времени.

> **Важно**
>
> Так как статистика по событиям ожидания находится в ОЗУ, важно очищать профиль ожиданий время от времени во избежание переполнения памяти.

Если вас не интересует распределение событий ожидания по процессам, вы можете присвоить переменной `pg_wait_sampling.profile_pid` `false`. В этом случае значение `pid` для всех процессов считается равным нулю и показатели ожиданий для всех процессов аккумулируются вместе.

F.52.3. Справка

F.52.3.1. Переменные GUC

Поведение расширения `pg_wait_sampling` настраивается с помощью переменных GUC, которые хранятся в локальной памяти. Изменить их можно в файле `postgresql.conf` или при помощи команды `ALTER SYSTEM`. Чтобы изменения вступили в силу, перезагрузите конфигурацию сервера. Это можно сделать, например, вызвав функцию `pg_reload_conf()` или выполнив команду `pg_ctl reload`.

`pg_wait_sampling.history_size` (`int4`)

Размер буфера в памяти для истории показаний (максимальное число показаний).

По умолчанию: 5000

`pg_wait_sampling.history_period` (`int4`)

Интервал снятия показаний для истории, в миллисекундах.

По умолчанию: 10

`pg_wait_sampling.profile_period` (`int4`)

Интервал снятия показаний для профиля, в миллисекундах.

По умолчанию: 10

`pg_wait_sampling.profile_pid` (`bool`)

Режим обработки показаний для профиля ожиданий. Если равняется `true`, профиль ожиданий собирается по процессам. Если задать для `pg_wait_sampling.profile_pid` значение `false`, профиль будет собираться по всем процессам сразу. В этом случае значение `pid` для всех процессов принимается равным нулю и соответствующая строка в профиле отражает события ожидания для всех процессов.

По умолчанию: `true`

`pg_wait_sampling.profile_queries` (`enum`)

Режим обработки показаний для профиля ожидания. Если для `pg_wait_sampling.profile_queries` установлено значение `none`, то поле `queryid` в представлениях будет равно нулю. Если для него установлено значение `top`, то записываются только идентификаторы сообщений верхнего уровня. Если установлено значение `all`, то записываются идентификаторы запросов вложенных сообщений. Чтобы собрать профиль ожидания для каждого запроса, убедитесь, что расширение pg_stat_statements настроено, и установите для `pg_wait_sampling.profile_queries` значение `top`. Для версии 14 или выше возможно вычислить идентификаторы запросов, настроив параметр `compute_query_id` и установив для `pg_wait_sampling.profile_queries` значение `top` или `all`. За подробностями обратитесь к Подразделу 18.9.2.

По умолчанию: `top`

`pg_wait_sampling.sample_cpu` (`bool`)

Режим обработки показаний, который определяет, следует ли собирать показания о процессах, обрабатываемых в данный момент процессором. Если для `pg_wait_sampling.sample_cpu` установлено значение `true`, то выборка также включает процессы, которые ничего не ожидают. Столбцы событий ожидания для таких процессов будут иметь значение `NULL`.

По умолчанию: `true`

F.52.3.2. Представления pg_wait_sampling

F.52.3.2.1. Представление pg_wait_sampling_current

Представление `pg_wait_sampling_current` выдаёт информацию о текущих событиях ожидания для всех процессов, включая фоновые рабочие процессы.

Таблица F.32. Представление pg_wait_sampling_current

Имя столбца	Столбец Тип	Описание
`pid`	`int4`	Идентификатор процесса
`event_type`	`text`	Имя типа события ожидания
`event`	`text`	Имя события ожидания
`queryid`	`int8`	Идентификатор запроса

F.52.3.2.2. Представление pg_wait_sampling_history

Представление `pg_wait_sampling_history` содержит историю событий ожидания. Эти данные хранятся в кольцевом буфере в памяти.

Таблица F.33. Представление pg_wait_sampling_history

Имя столбца	Столбец Тип	Описание
`pid`	`int4`	Идентификатор процесса
`ts`	`timestamptz`	Время снятия показания
`event_type`	`text`	Имя типа события ожидания
`event`	`text`	Имя события ожидания
`queryid`	`int8`	Идентификатор запроса

F.52.3.2.3. Представление pg_wait_sampling_profile

Представление `pg_wait_sampling_profile` содержит профиль событий ожидания. Эти данные хранятся в виде хеш-таблицы в памяти.

Таблица F.34. Представление pg_wait_sampling_profile

Имя столбца	Столбец Тип	Описание
`pid`	`int4`	Идентификатор процесса
`event_type`	`text`	Имя типа события ожидания
`event`	`text`	Имя события ожидания
`queryid`	`int8`	Идентификатор запроса
`count`	`int8`	Число наблюдений

F.52.3.3. Функции

`pg_wait_sampling_get_current(`*`pid`*` int4)`

Возвращает представление `pg_wait_sampling_current` со списком текущих событий ожидания. Если вы зададите аргумент *`pid`*, представление будет отфильтровано по процессу с заданным идентификатором.

Аргументы:

- *`pid`* — Необязательный параметр. Идентификатор процесса, для которого будут выводиться текущие события ожидания.

`pg_wait_sampling_reset_profile()`

Сбрасывает профиль ожиданий и очищает память.

F.52.4. Авторы

- Александр Коротков
- Ильдус Курбангалиев `<i.kurbangaliev@gmail.com>`

F.53. pg_walinspect — просмотр журнала предзаписи на низком уровне

Модуль `pg_walinspect` предоставляет SQL-функции для просмотра журнала предзаписи на низком уровне. Он работает с запущенным кластером баз данных Postgres Pro и может быть полезен для целей отладки, анализа, отчётности или обучения. Модуль похож на pg_waldump, но работает через SQL, а не как отдельная утилита.

Все функции этого модуля выдают информацию из WAL, относящуюся к текущей линии времени сервера.

> **Примечание**
>
> Функции `pg_walinspect` зачастую вызываются с аргументом LSN, который задаёт *начало* интересующей записи WAL. Тем не менее некоторые функции, например `pg_logical_emit_message`, возвращают LSN *после* только что добавленной записи.

> **Подсказка**
>
> Все функции `pg_walinspect`, которые показывают информацию о записях, попадающих в определённый диапазон значений LSN, могут принимать *`конечные_lsn`* после текущего LSN сервера. При использовании *`конечного_lsn`* «из будущего» ошибки не возникнет.
>
> Может быть удобным указать значение `FFFFFFFF/FFFFFFFF` (максимально допустимое значение `pg_lsn`) в качестве аргумента *`конечного_lsn`*. Это аналогично варианту указать *`конечный_lsn`*, совпадающий с текущим LSN сервера.

По умолчанию использовать эти функции разрешено только суперпользователям и ролям, включённым в роль `pg_read_server_files`. Суперпользователь может дать доступ другим, воспользовавшись командой `GRANT`.

F.53.1. Функции общего назначения

`pg_get_wal_record_info(in_lsn pg_lsn) returns record`

Получает информацию о записи WAL, которая расположена по *`заданному_lsn`* или после него. Например:

```
postgres=# SELECT * FROM pg_get_wal_record_info('0/E419E28');
-[ RECORD 1 ]----+-------------------------------------------------
start_lsn        | 0/E419E28
end_lsn          | 0/E419E68
prev_lsn         | 0/E419D78
xid              | 0
resource_manager | Heap2
record_type      | VACUUM
record_length    | 58
main_data_length | 2
fpi_length       | 0
description      | nunused: 5, unused: [1, 2, 3, 4, 5]
block_ref        | blkref #0: rel 1663/16385/1249 fork main blk 364
```

Если *`заданный_lsn`* не находится в начале записи WAL, будет показана информация о следующей корректной записи WAL. Если такой записи WAL нет, функция выдаст ошибку.

`pg_get_wal_records_info(start_lsn pg_lsn, end_lsn pg_lsn) returns setof record`

Получает информацию обо всех корректных записях WAL между *`начальным_lsn`* и *`конечным_lsn`*. Возвращает одну строку для каждой записи WAL. Пример использования функции:

```
postgres=# SELECT * FROM pg_get_wal_records_info('0/1E913618', '0/1E913740') LIMIT
 1;
-[ RECORD 1 ]----+----------------------------------------------------------
start_lsn        | 0/1E913618
end_lsn          | 0/1E913650
prev_lsn         | 0/1E9135A0
xid              | 0
resource_manager | Standby
record_type      | RUNNING_XACTS
record_length    | 50
main_data_length | 24
fpi_length       | 0
description      | nextXid 33775 latestCompletedXid 33774 oldestRunningXid 33775
block_ref        |
```

Функция выдаёт ошибку, если *`начальный_lsn`* недоступен.

`pg_get_wal_block_info(start_lsn pg_lsn, end_lsn pg_lsn, show_data boolean DEFAULT true) returns setof record`

Получает информацию о каждой ссылке на блок из всех корректных записей WAL между *`начальным_lsn`* и *`конечным_lsn`* с одной или более ссылок на блок. Возвращает одну строку для ссылки на блок для каждой записи WAL. Например:

```
postgres=# SELECT * FROM pg_get_wal_block_info('0/1230278', '0/12302B8');
-[ RECORD 1 ]-----+-----------------------------------
start_lsn         | 0/1230278
end_lsn           | 0/12302B8
prev_lsn          | 0/122FD40
block_id          | 0
reltablespace     | 1663
reldatabase       | 1
relfilenode       | 2658
relforknumber     | 0
relblocknumber    | 11
xid               | 341
resource_manager  | Btree
record_type       | INSERT_LEAF
record_length     | 64
main_data_length  | 2
block_data_length | 16
block_fpi_length  | 0
block_fpi_info    |
description       | off: 46
block_data        | \x00002a00070010402630000070696400
block_fpi_data    |
```

В этом примере рассмотрена запись WAL, содержащая только одну ссылку на блок, однако во многих записях WAL таких ссылок несколько. Строки, возвращаемые функцией `pg_get_wal_block_info`, будут гарантированно иметь уникальную комбинацию *`начального_lsn`* и *`id_блока`*.

Большая часть показанной здесь информации совпадает с выводом функции `pg_get_wal_records_info`, при условии передачи одинаковых аргументов. Однако `pg_get_wal_block_info` выводит информацию из каждой записи WAL в расширенном виде, добавляя в вывод по одной строке для каждой ссылки на блок, поэтому некоторые детали отслеживаются на уровне ссылки на блок, а не на уровне целой записи. Такая структура полезна при работе с запросами, которые отслеживают изменение отдельных блоков с течением времени. Обратите внимание, что для записей без ссылок на блоки (например, записи WAL для `COMMIT`), строки возвращаться не будут, поэтому функция `pg_get_wal_block_info` действительно может возвращать *меньше* строк, чем функция `pg_get_wal_records_info`.

Параметры `reltablespace`, `reldatabase` и `relfilenode` ссылаются на `pg_tablespace`.`oid`, `pg_database`.`oid` и `pg_class`.`relfilenode` соответственно. Поле `relforknumber` обозначает номер слоя в отношении для ссылки на блок (за подробностями обратитесь к `common/relpath.h`).

Подсказка

Функция `pg_filenode_relation` (см. Таблицу 9.101) позволяет определить, какое отношение было изменено первоначально.

Клиенты могут избежать издержек материализации данных блока, что может существенно ускорить выполнение функции. Когда для *`show_data`* задано значение `false`, значения `block_data` и `block_fpi_data` опускаются (то есть выходные аргументы `block_data` и `block_fpi_data` принимают значение `NULL` для всех возвращаемых строк). Очевидно, что подобная оптимизация осуществима только для запросов, которым на самом деле не требуются данные блоков.

Функция выдаёт ошибку, если *`начальный_lsn`* недоступен.

`pg_get_wal_stats(start_lsn pg_lsn, end_lsn pg_lsn, per_record boolean DEFAULT false) returns setof record`

Выдаёт статистику по всем корректным записям WAL между *`начальным_lsn`* и *`конечным_lsn`*. По умолчанию возвращает одну строку для каждого типа *`менеджера_ресурсов`* (resource_manager). Когда *`по_типу_записи`* имеет значение `true`, то возвращает отдельные строки для разных *`типов_записей`* (record_type). Пример использования функции:

```
postgres=# SELECT * FROM pg_get_wal_stats('0/1E847D00', '0/1E84F500')
             WHERE count > 0 AND
                   "resource_manager/record_type" = 'Transaction';
             LIMIT 1;
-[ RECORD 1 ]----------------+-------------------
resource_manager/record_type | Transaction
count                        | 2
count_percentage             | 8
record_size                  | 875
record_size_percentage       | 41.23468426013195
fpi_size                     | 0
fpi_size_percentage          | 0
combined_size                | 875
combined_size_percentage     | 2.8634072910530795
```

Функция выдаёт ошибку, если *`начальный_lsn`* недоступен.

F.53.2. Автор

Бхарат Рупиредди (Bharath Rupireddy) `<bharath.rupireddyforpostgres@gmail.com>`

F.54. plantuner — поддержка указаний для планировщика, позволяющих отключать или подключать индексы при выполнении запроса

Модуль `plantuner` добавляет поддержку указаний для планировщика, позволяющих отключать или подключать определённые индексы при выполнении запроса.

F.54.1. Объяснение

В некоторых случаях может потребоваться управлять планировщиком, давая ему указания, чтобы оптимизатор не выполнял некоторые части своего алгоритма. В частности, нередко возникают ситуации, когда разработчик хочет временно отключить некоторые индексы, не удаляя их, либо наоборот, использовать определённый индекс принудительно.

Эта версия `plantuner` даёт возможность скрыть определённые индексы от планировщика Postgres Pro, чтобы он их не использовал. С некоторой нагрузкой Postgres Pro бывает слишком пессимистичным в отношении только что созданных таблиц и считает, что в них содержится гораздо больше строк, чем есть на самом деле. Если же переменная GUC `plantuner.fix_empty_table` имеет значение `true`, `plantuner` будет обнулять число страниц/кортежей в таблице, которая не содержит никаких блоков в файле.

F.54.2. Переменные GUC

`plantuner.disable_index` — список индексов, которые не будет видеть планировщик.

`plantuner.enable_index` — список индексов, которые будет видеть планировщик, даже если они скрыты параметром `plantuner.disable_index`.

F.54.3. Пример

Чтобы включить этот модуль, вы можете либо загрузить разделяемую библиотеку `plantuner` в сеансе `psql`, либо указать её в параметре `shared_preload_libraries` в `postgresql.conf`.

```
=# LOAD 'plantuner';
=# create table test(id int);
=# create index id_idx on test(id);
=# create index id_idx2 on test(id);
=# \d test
     Table "public.test"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
Indexes:
    "id_idx" btree (id)
    "id_idx2" btree (id)
=# explain select id from test where id=1;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx2  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)
=# set enable_seqscan=off;
=# set plantuner.disable_index='id_idx2';
=# explain select id from test where id=1;
                              QUERY PLAN
----------------------------------------------------------------------
```

```
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)
=# set plantuner.disable_index='id_idx2,id_idx';
=# explain select id from test where id=1;
                              QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on test  (cost=10000000000.00..10000000040.00 rows=12 width=4)
   Filter: (id = 1)
(2 rows)
=# set plantuner.enable_index='id_idx';
=# explain select id from test where id=1;
                             QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=4.34..15.03 rows=12 width=4)
   Recheck Cond: (id = 1)
   ->  Bitmap Index Scan on id_idx  (cost=0.00..4.34 rows=12 width=0)
         Index Cond: (id = 1)
(4 rows)
```

F.54.4. Авторы

Разработку осуществили Фёдор Сигаев (teodor@sigaev.ru) и Олег Бартунов (oleg@sai.msu.su).

Эту работу спонсировал проект Nomao (http://www.nomao.com).

F.55. postgres_fdw — обращение к данным, находящимся на внешних серверах Postgres Pro

Модуль `postgres_fdw` предоставляет обёртку сторонних данных `postgres_fdw`, используя которую можно обращаться к данным, находящимся на внешних серверах Postgres Pro.

Функциональность этого модуля во многом пересекается с функциональностью старого модуля dblink. Однако `postgres_fdw` предоставляет более прозрачный и стандартизированный синтаксис для обращения к удалённым таблицам и во многих случаях даёт лучшую производительность.

Чтобы подготовиться к обращению к удалённым данным через `postgres_fdw`:

1. Установите расширение `postgres_fdw` с помощью команды CREATE EXTENSION.
2. Создайте объект стороннего сервера, используя CREATE SERVER, который будет представлять удалённую базу данных, к которой вы хотите подключаться. Укажите свойства подключения, кроме `user` и `password`, в параметрах объекта сервера.
3. Создайте сопоставление пользователей, используя CREATE USER MAPPING, для каждого пользователя базы, которому нужен доступ к удалённому серверу. Укажите имя и пароль удалённого пользователя в параметрах `user` и `password` сопоставления.
4. Создайте стороннюю таблицу, используя CREATE FOREIGN TABLE или IMPORT FOREIGN SCHEMA, для каждой удалённой таблицы, к которой вы хотите обращаться. Столбцы сторонней таблицы должны соответствовать столбцам целевой удалённой таблицы. Однако вы можете использовать локально имена таблиц и/или столбцов, отличные от удалённых, если укажете корректные удалённые имена в параметрах объекта сторонней таблицы.

После этого для обращения к данным, хранящимся в нижележащей удалённой таблице, вам нужно только выполнять `SELECT`. Вы также можете изменять данные в удалённой таблице, выполняя `INSERT`, `UPDATE`, `DELETE`, `COPY` или `TRUNCATE`. (Разумеется, удалённый пользователь, указанный в сопоставлении, должен иметь необходимые права для этого.)

Обратите внимание, что в запросах на чтение или изменение данных в удалённых таблицах указание `ONLY` в командах `SELECT`, `UPDATE`, `DELETE` и `TRUNCATE` не действует.

Заметьте, что в настоящее время в `postgres_fdw` не поддерживаются операторы `INSERT` с предложением `ON CONFLICT DO UPDATE`. Однако предложение `ON CONFLICT DO NOTHING` поддерживается, при отсутствии указания для выбора уникального индекса. Заметьте также, что `postgres_fdw` поддерживает перемещение строк, вызванное командами `UPDATE`, выполняемыми для секционированных таблиц. Однако в настоящее время невозможно выполнить изменение, при котором удалённая секция, выбранная для добавления перемещаемой строки, также является целевой секцией для `UPDATE` и должна модифицироваться в другой части той же команды.

Обычно рекомендуется объявлять столбцы сторонней таблицы точно с теми же типами данных и правилами сортировки, если они применимы, как у целевых столбцов удалённой таблицы. Хотя `postgres_fdw` в настоящее время довольно лоялен к преобразованиям типов данных при необходимости, но когда типы или правила сортировки не совпадают, возможны неожиданные семантические аномалии, вследствие того, что удалённый сервер будет обрабатывать условия запроса не так, как локальный сервер.

Заметьте, что сторонняя таблица может быть объявлена с меньшим количеством или с другим порядком столбцов, чем в нижележащей удалённой таблице. Сопоставление столбцов удалённой таблицы осуществляется по имени, а не по позиции.

F.55.1. Параметры обёртки для postgres_fdw

F.55.1.1. Параметры подключения

Для стороннего сервера, настраиваемого с использованием обёртки сторонних данных `postgres_fdw`, можно задать те же параметры, что принимает libpq в строках подключения, как описано в Подразделе 32.1.2, за исключением следующих параметров, которые не допускаются или обрабатывается по-другому:

- `user`, `password` и `sslpassword` (их следует задавать в сопоставлениях пользователей или в файле описания служб)
- `client_encoding` (автоматически принимается равной локальной кодировке сервера)
- `application_name` может задаваться как параметр подключения *и/или* значение postgres_fdw.application_name. Если задаётся и там, и там, `postgres_fdw.application_name` переопределяет значение параметра подключения. В отличие от libpq, `postgres_fdw` допускает использование спецпоследовательностей в `application_name`. За дополнительной информацией обратитесь к postgres_fdw.application_name.
- `fallback_application_name` (всегда `postgres_fdw`)
- `sslkey` и `sslcert` могут задаваться в свойствах соединения и/или сопоставления пользователя. Если они задаются и там, и там, приоритет имеет свойство сопоставления пользователя.

Создавать и изменять сопоставления пользователей, устанавливающие параметр `sslcert` или `sslkey`, могут только суперпользователи.

Обычные пользователи могут подключаться к сторонним серверам с аутентификацией по паролю или с делегированными учётными данными GSSAPI, поэтому в сопоставлениях для обычных пользователей, где требуется аутентификация по паролю, необходимо указывать параметр `password`.

Суперпользователь может отключить проверку пароля на уровне сопоставления пользователя, установив параметр `password_required 'false'`, например:

```
ALTER USER MAPPING FOR some_non_superuser SERVER loopback_nopw
OPTIONS (ADD password_required 'false');
```

Чтобы недоверенные пользователи не могли злоупотреблять правами пользователя, от имени которого работает сервер postgres, и повысить свои привилегии, изменять это свойство в сопоставлении пользователя разрешено только суперпользователю.

Позаботьтесь о том, чтобы через установленное сопоставление пользователь не мог подключиться к другой базе с правами суперпользователя, как в случаях CVE-2007-3278 и CVE-2007-6601. Не задавайте параметр `password_required=false` для роли `public`. Помните, что сопоставляемый пользователь имеет возможность использовать любые клиентские сертификаты, а также записи в файлах `.pgpass`, `.pg_service.conf` и т. п. в домашнем каталоге системного пользователя, от имени которого работает сервер postgres. Они также могут использовать отношения доверия, установленные режимами аутентификации, например `peer` или `ident`.

F.55.1.2. Параметры имени объекта

Эти параметры позволяют управлять тем, как на удалённый сервер Postgres Pro будут передаваться имена, фигурирующие в операторах SQL. Данные параметры нужны, когда сторонняя таблица создаётся с именами, отличными от имён удалённой таблицы.

`schema_name` (`string`)

Этот параметр, который может задаваться для сторонней таблицы, указывает имя схемы для обращения к этой таблице на удалённом сервере. Если данный параметр опускается, применяется схема сторонней таблицы.

`table_name` (`string`)

Этот параметр, который может задаваться для сторонней таблицы, указывает имя таблицы для обращения к этой таблице на удалённом сервере. Если данный параметр опускается, применяется имя сторонней таблицы.

`column_name` (`string`)

Этот параметр, который может задаваться для столбца сторонней таблицы, указывает имя столбца для обращения к этому столбцу на удалённом сервере. Если данный параметр опускается, применяется исходное имя столбца.

F.55.1.3. Параметры оценки стоимости

Модуль `postgres_fdw` получает удалённые данные, выполняя запросы на удалённых серверах, поэтому в идеале ожидаемая стоимость сканирования сторонней таблицы должна равняться стоимости выполнения на удалённом сервере плюс издержки сетевого взаимодействия. Самый надёжный способ получить такие оценки — узнать стоимость у удалённого сервера и добавить некоторую надбавку — но для простых запросов может быть невыгодно передавать дополнительный запрос, только чтобы получить оценку стоимости. Поэтому `postgres_fdw` предоставляет следующие параметры, позволяющие управлять вычислением оценки стоимости:

`use_remote_estimate` (`boolean`)

> Этот параметр, который может задаваться для сторонней таблицы или для стороннего сервера, определяет, будет ли `postgres_fdw` выполнять удалённо команды `EXPLAIN` для получения оценок стоимости. Параметр, заданный для сторонней таблицы, переопределяет параметр сервера, но только для данной таблицы. Значение по умолчанию — `false` (выкл.).

`fdw_startup_cost` (`floating point`)

> Этот параметр, который может задаваться для стороннего сервера, устанавливает значение с плавающей точкой, добавляемое к оценке стоимости запуска для любого сканирования сторонней таблицы на этом сервере. Он выражает дополнительные издержки на установление подключения, разбор и планирование запроса на удалённой стороне и т. д. Значение по умолчанию — `100`.

`fdw_tuple_cost` (`floating point`)

> Этот параметр, который может задаваться для стороннего сервера в виде значения с плавающей точкой, устанавливает дополнительную цену чтения одного кортежа из сторонней таблицы на этом сервере. Это число можно увеличить или уменьшить, чтобы отразить меньшую или большую фактическую скорость сетевого взаимодействия с удалённым сервером. Значение по умолчанию — `0.2`.

Когда поведение `use_remote_estimate` включено, `postgres_fdw` получает количество строк и оценку стоимости с удалённого сервера, а затем добавляет к оценке стоимости `fdw_startup_cost` и `fdw_tuple_cost`. Когда поведение `use_remote_estimate` отключено, `postgres_fdw` рассчитывает число строк и оценку стоимости локально, а затем так же добавляет к этой оценке `fdw_startup_cost` и `fdw_tuple_cost`. Локальная оценка может быть точной только при условии наличия локальной копии статистики удалённых таблиц. Обновить эту статистику для сторонней таблицы можно с помощью команды ANALYZE; при этом удалённая таблица будет просканирована, и по её содержимому будут вычислена и сохранена статистика как для локальной таблицы. Локальное хранение статистики может быть полезно для сокращения издержек планирования для удалённой таблицы — но если удалённая таблица меняется часто, локальная статистика будет быстро устаревать.

Следующий параметр определяет поведение операции `ANALYZE`:

`analyze_sampling` (`string`)

> Этот параметр, который может быть указан для сторонней таблицы или стороннего сервера, определяет логику работы `ANALYZE`: будут ли данные при применении команды для сторонней таблицы выбираться на стороннем сервере или все данные будут считываться и переноситься для выборки локально. Поддерживаются значения `off`, `random`, `system`, `bernoulli` и `auto`. Значение `off` позволяет отключать выборку данных на стороннем сервере, в результате чего все данные переносятся и выборка выполняется локально. Значение `random` позволяет производить удалённую выборку данных с помощью функции `random()`, тогда как при значениях `system` и `bernoulli` используются встроенные методы `TABLESAMPLE` с соответствующими названиями. `random` работает со всеми версиями сторонних серверов, тогда как `TABLESAMPLE` поддерживается только в версии 9.5 и новее. При значении `auto` (по умолчанию) метод выборки выбирается автоматически — сейчас это `bernoulli` или `random` в зависимости от версии стороннего сервера.

F.55.1.4. Параметры удалённого выполнения

По умолчанию ограничения `WHERE`, содержащие встроенные операторы и функции, обрабатываются на удалённом сервере, а ограничения, содержащие вызовы не встроенных функций, проверяются локально после получения строк. Если же расширенные функции доступны на удалённом сервере и можно рассчитывать, что они дадут те же результаты, что и локально, производительность можно увеличить, передавая и такие предложения `WHERE` для удалённого выполнения. Этим поведением позволяет управлять следующий параметр:

`extensions` (`string`)

В этом параметре задаётся список имён расширений Postgres Pro через запятую, которые установлены и имеют совместимые версии и на локальном, и на удалённом сервере. Относящиеся к перечисленным расширениям и при этом постоянные (immutable) функции и операторы могут передаваться на выполнение удалённому серверу. Этот параметр можно задать только для стороннего сервера, но не для таблицы.

При использовании параметра `extensions` *пользователь сам отвечает* за то, чтобы перечисленные расширения существовали и их поведение было одинаковым на локальном и удалённом сервере. В противном случае удалённые запросы могут выдавать ошибки или неожиданные результаты.

`fetch_size` (`integer`)

Этот параметр определяет, сколько строк должна получать `postgres_fdw` в одной операции выборки. Его можно задать для сторонней таблицы или стороннего сервера. Значение по умолчанию — `100` строк.

`batch_size` (`integer`)

Этот параметр определяет, сколько строк должна вставлять `postgres_fdw` в одной операции вставки. Его можно задать для сторонней таблицы или стороннего сервера. Значение по умолчанию — `1` строка.

Обратите внимание, что фактическое количество строк, вставляемых `postgres_fdw` в одной порции, зависит от количества столбцов и значения `batch_size`. Порция передаётся в одном запросе, а протокол libpq, используя который `postgres_fdw` подключается к удалённому серверу, ограничивает количество параметров одного запроса числом 65535. Когда «число столбцов» * `batch_size` превышает этот предел, значение `batch_size` корректируется во избежание ошибки.

Этот параметр также используется при копировании строк в сторонние таблицы. В этом случае фактическое число строк, которые копирует `postgres_fdw` в рамках одной операции, определяется так же, как и при операции вставки. Тем не менее ввиду ограничений реализации команды `COPY` максимальное число копируемых строк не может превышать 1000.

F.55.1.5. Параметры асинхронного выполнения

Обёртка `postgres_fdw` поддерживает асинхронное выполнение, когда для улучшения производительности несколько частей узла `Append` выполняются одновременно, а не последовательно. Асинхронным выполнением управляет следующий параметр:

`async_capable` (`boolean`)

Этот параметр определяет, будет ли `postgres_fdw` допускать одновременное сканирование сторонних таблиц для асинхронного выполнения. Его можно задать для сторонней таблицы или для стороннего сервера. Параметр, определённый на уровне таблицы, переопределяет параметр уровня сервера. Значение по умолчанию — `false`.

Чтобы получаемые со стороннего сервера данные были согласованными, `postgres_fdw` открывает только одно соединение для каждого стороннего сервера и выполняет все запросы к нему последовательно, даже если в запросе задействуются несколько сторонних таблиц (несколько подключений будет установлено, только если для этих таблиц используются разные сопоставления пользователей). Поэтому возможно, что производительность увеличится, если отключить этот параметр и тем самым исключить издержки, связанные с асинхронным выполнением запросов.

Асинхронное выполнение используется, даже когда узел `Append` содержит как подпланы, выполняемые синхронно, так и подпланы, выполняемые асинхронно. В таких случаях, если `postgres_fdw` обрабатывает асинхронные подпланы, кортежи этих подпланов не возвращаются до тех пор, пока хотя бы один синхронный подплан не вернёт все кортежи, поскольку этот подплан выполняется, пока асинхронные подпланы ожидают результатов асинхронных запросов к сторонним серверам. Такое поведение может измениться в будущих версиях.

F.55.1.6. Параметры управления транзакциями

Как описано в разделе «Управление транзакциями», для управления транзакциями и подтранзакциями `postgres_fdw` создаёт соответствующие удалённые транзакции и подтранзакции. Если текущей локальной транзакции соответствует несколько удалённых транзакций, по умолчанию `postgres_fdw` фиксирует или прерывает эти удалённые транзакции последовательно, когда фиксируется или прерывается локальная транзакция. Удалённые подтранзакции `postgres_fdw` по умолчанию фиксирует или прерывает также последовательно. Ускорить эту процедуру позволяют следующие параметры:

`parallel_commit` (`boolean`)

Этот параметр определяет, будет ли `postgres_fdw` при фиксировании локальной транзакции фиксировать все соответствующие ей удалённые транзакции одновременно. Действие этого параметра также распространяется на подтранзакции. Его можно определить только для удалённого сервера, а не на уровне таблицы. Значение по умолчанию — `false`.

`parallel_abort` (`boolean`)

Этот параметр определяет, будет ли `postgres_fdw` при прерывании локальной транзакции прерывать все соответствующие ей удалённые транзакции одновременно. Действие этого параметра также распространяется на подтранзакции. Его можно определить только для удалённого сервера, а не на уровне таблицы. Значение по умолчанию — `false`.

Если текущая локальная транзакция задействует несколько сторонних серверов, для которых включены эти параметры, удалённые транзакции на них фиксируются или прерываются одновременно, когда фиксируется или прерывается локальная транзакция.

Включение этих параметров может негативно повлиять на производительность стороннего сервера с большим количеством удалённых транзакций при фиксации или прерывании локальной транзакции.

F.55.1.7. Параметры изменения данных

По умолчанию все сторонние таблицы, доступные через `postgres_fdw`, считаются допускающими изменения. Это можно переопределить с помощью следующего параметра:

`updatable` (`boolean`)

Этот параметр определяет, будет ли `postgres_fdw` допускать изменения в сторонних таблицах посредством команд `INSERT`, `UPDATE` и `DELETE`. Его можно задать для сторонней таблицы или для стороннего сервера. Параметр, определённый на уровне таблицы, переопределяет параметр уровня сервера. Значение по умолчанию — `true` (изменения разрешены).

Конечно, если удалённая таблица на самом деле не допускает изменения, всё равно произойдёт ошибка. Использование этого параметра прежде всего позволяет выдать ошибку локально, не обращаясь к удалённому серверу. Заметьте, однако, что представление `information_schema` будет показывать, что определённая сторонняя таблица `postgres_fdw` является изменяемой (или нет), согласно значению данного параметра, не проверяя это на удалённом сервере.

F.55.1.8. Параметры операции опустошения таблиц

По умолчанию все сторонние таблицы, доступные через `postgres_fdw`, считаются допускающими опустошение. Это можно переопределить с помощью следующего параметра:

`truncatable` (`boolean`)

Этот параметр определяет, будет ли `postgres_fdw` допускать опустошение сторонних таблиц посредством команды `TRUNCATE`. Его можно задать для сторонней таблицы или для стороннего сервера. Параметр, определённый на уровне таблицы, переопределяет параметр уровня сервера. Значение по умолчанию — `true` (опустошение разрешено).

Конечно, если удалённая таблица на самом деле не допускает опустошение, всё равно произойдёт ошибка. Использование этого параметра прежде всего позволяет выдать ошибку локально, не обращаясь к удалённому серверу.

F.55.1.9. Параметры импорта

Обёртка `postgres_fdw` позволяет импортировать определения сторонних таблиц с применением команды IMPORT FOREIGN SCHEMA. Эта команда создаёт на локальном сервере определения сторонних таблиц, соответствующие таблицам или представлениям, существующим на удалённом сервере. Если импортируемые удалённые таблицы содержат столбцы пользовательских типов данных, на локальном сервере должны быть совместимые типы с теми же именами.

Поведение процедуры импорта можно настроить следующими параметрами (задаваемыми в команде `IMPORT FOREIGN SCHEMA`):

`import_collate` (`boolean`)

Этот параметр устанавливает, будут ли в определения сторонних таблиц, импортируемых с внешнего сервера, включаться характеристики столбцов `COLLATE`. По умолчанию они включаются. Вам может потребоваться отключить его, если на удалённом сервере набор имён правил сортировки отличается от локального, что скорее всего будет иметь место, если серверы работают в разных операционных системах. Тем не менее отключение чревато тем, что правила сортировки в столбцах импортируемых таблиц не совпадут с правилами для нижележащих данных, что приведёт к аномальному поведению запросов.

Даже когда этот параметр равен `true`, импортировать столбцы с правилом сортировки, выбираемым на удалённом сервере по умолчанию, рискованно. Столбцы импортируются со свойством `COLLATE "default"`, а в результате будет применяться правило сортировки, выбираемое по умолчанию на локальном сервере, которое может отличаться от выбираемого на удалённом.

`import_default` (`boolean`)

Этот параметр устанавливает, будут ли в определения сторонних таблиц, импортируемых с внешнего сервера, включаться заданные для столбцов выражения `DEFAULT`. По умолчанию они не включаются. Если вы включите этот параметр, остерегайтесь выражений по умолчанию, которые могут вычисляться на локальном сервере не так, как на удалённом; например, частый источник проблем — `nextval()`. Если в импортируемом выражении используются функции или операторы, несуществующие локально, команда `IMPORT` в целом выдаст сбой.

`import_generated` (`boolean`)

Этот параметр определяет, включаются ли выражения генерируемых столбцов (`GENERATED`) в определения сторонних таблиц, импортируемых со стороннего сервера. Значение по умолчанию — `true`. Выполнение `IMPORT` завершится ошибкой, если в импортируемом генерирующем выражении используются функции или операторы, не существующие локально.

`import_not_null` (boolean)

Этот параметр устанавливает, будут ли в определения сторонних таблиц, импортируемых с внешнего сервера, включаться ограничения столбцов `NOT NULL`. По умолчанию они включаются.

Заметьте, что никакие другие ограничения, кроме `NOT NULL`, из удалённых таблиц импортироваться не будут. Хотя Postgres Pro поддерживает ограничения-проверки для сторонних таблиц, никаких средств для автоматического импорта их нет из-за риска различного вычисления выражения ограничения на локальном и удалённом серверах. Любая такая несогласованность в поведении ограничений-проверок могла бы привести к сложно выявляемым ошибкам в оптимизации запросов. Поэтому, если вы хотите импортировать ограничения-проверки, вы должны сделать это вручную и должны внимательно проверить семантику каждого. Более подробно интерпретация ограничений-проверок для сторонних таблиц описана в CREATE FOREIGN TABLE.

Таблицы или сторонние таблицы, являющиеся секциями некоторой другой таблицы, импортируются, только когда они явным образом указаны в предложении `LIMIT TO`. В противном случае команда IMPORT FOREIGN SCHEMA автоматически исключает их. Так как все данные могут быть доступны через корневую таблицу в иерархии секционирования, должно быть достаточно импортировать только секционированные таблицы, чтобы получить доступ ко всем данным, не создавая дополнительные объекты.

F.55.1.10. Параметры управления соединением

По умолчанию все соединения, устанавливаемые `postgres_fdw` со сторонними серверами, остаются открытыми в текущем сеансе для повторного использования.

`keep_connections` (boolean)

Этот параметр определяет, будет ли `postgres_fdw` оставлять открытыми соединения со сторонними серверами, чтобы последующие запросы также могли использовать эти соединения. Этот параметр можно задать только для стороннего сервера. Значение по умолчанию — `on`. Если задано значение `off`, все соединения с этим сторонним сервером будут закрываться в конце каждой транзакции.

F.55.2. Функции

`postgres_fdw_get_connections(OUT server_name text, OUT valid boolean) returns setof record`

Эта функция возвращает имена всех сторонних серверов, с которыми `postgres_fdw` установил соединение в локальном сеансе. Она также возвращает признак действительного соединения. Этот признак — `false`, если это соединение со сторонним сервером использовалось в текущей локальной транзакции, но объект стороннего сервера или сопоставление пользователя были изменены или удалены (обратите внимание, что если объект стороннего сервера удалён, вместо имени сервера в записи этого соединения будет `NULL`). Такое недействующее соединение будет закрыто в конце транзакции. Для действительных соединений этот признак установлен. Если открытых соединений нет, эта функция не возвращает никакие записи. Пример использования функции:

```
postgres=# SELECT * FROM postgres_fdw_get_connections() ORDER BY 1;
 server_name | valid
-------------+-------
 loopback1   | t
 loopback2   | f
```

`postgres_fdw_disconnect(server_name text) returns boolean`

Эта функция закрывает открытые соединения, установленные `postgres_fdw` с указанным сторонним сервером в локальном сеансе. Обратите внимание, что с одним сервером может быть установлено несколько соединений, использующих различные сопоставления пользователей. Если какие-либо соединения используются в текущей локальной транзакции, они не будут разорваны, о чём сообщат соответствующие предупреждения. Если функция закрывает хотя бы одно соединение, она возвращает `true`; в противном случае `false`. Если сторонний сервер с указанным именем не найден, выводится сообщение об ошибке. Пример использования функции:

```
postgres=# SELECT postgres_fdw_disconnect('loopback1');
 postgres_fdw_disconnect
-------------------------
 t
```

`postgres_fdw_disconnect_all() returns boolean`

Эта функция закрывает все открытые соединения, установленные `postgres_fdw` со сторонними серверами в локальном сеансе. Если какие-либо соединения используются в текущей локальной транзакции, они не будут разорваны, о чем сообщат соответствующие предупреждения. Если функция закрывает хотя бы одно соединение, она возвращает `true`; в противном случае `false`. Пример использования функции:

```
postgres=# SELECT postgres_fdw_disconnect_all();
 postgres_fdw_disconnect_all
-----------------------------
 t
```

F.55.3. Управление соединением

Модуль `postgres_fdw` устанавливает соединение со сторонним сервером при первом запросе, в котором участвует сторонняя таблица, связанная со сторонним сервером. По умолчанию это соединение сохраняется и повторно используется для последующих запросов в том же сеансе. Этим поведением управляет параметр стороннего сервера `keep_connections`. Если для обращения к стороннему серверу задействуются разные пользователи (сопоставления пользователей), отдельное соединение устанавливается для каждого сопоставления пользователей.

При изменении или удалении определения стороннего сервера или сопоставления пользователей, связанные с ними соединения закрываются. Но обратите внимание, что если какие-либо соединения используются в текущей локальной транзакции, то они сохраняются до конца транзакции. Если для запросов, задействующих сторонние таблицы, понадобятся уже закрытые соединения, они будут открытыми заново.

По умолчанию установленное соединение сохраняется до конца локального или соответствующего удалённого сеанса. Чтобы соединения закрывались явно, можно отключить параметр `keep_connections` или использовать функции `postgres_fdw_disconnect` и `postgres_fdw_disconnect_all`. В частности, эти функции полезны тем, что позволяют освободить подключения на стороннем сервере, закрыв уже не нужные.

F.55.4. Управление транзакциями

В процессе выполнения запроса, в котором участвуют какие-либо удалённые таблицы на стороннем сервере, `postgres_fdw` открывает транзакцию на удалённом сервере, если такая транзакция ещё не была открыта для текущей локальной транзакции. Эта удалённая транзакция фиксируется или прерывается, когда фиксируется или прерывается локальная транзакция. Подобным образом реализуется и управление точками сохранения.

Для удалённой транзакции выбирается режим изоляции `SERIALIZABLE`, когда локальная транзакция открыта в режиме `SERIALIZABLE`; в противном случае применяется режим `REPEATABLE READ`. Этот выбор гарантирует, что если запрос сканирует несколько таблиц на удалённом сервере, он будет получать согласованные данные одного снимка для всех сканирований. Как следствие, последовательные запросы в одной транзакции будут видеть одни данные удалённого сервера, даже если на нём параллельно происходят изменения, вызванные другими действиями. Это поведение ожидаемо для локальной транзакции в режимах `SERIALIZABLE` и `REPEATABLE READ`, но для локальной транзакции в режиме `READ COMMITTED` оно может быть неожиданным. В будущих выпусках Postgres Pro эти правила могут быть изменены.

Заметьте, что подготовку удалённой транзакции для двухфазной фиксации `postgres_fdw` в настоящее время не поддерживает.

F.55.5. Оптимизация удалённых запросов

Обёртка `postgres_fdw` пытается оптимизировать удалённые запросы, уменьшая объём обмена данными со сторонними серверами. Для этого она может передавать на выполнение удалённому серверу предложения `WHERE` и не получать столбцы таблицы, не требующиеся для текущего запроса. Чтобы уменьшить риск некорректного выполнения запросов, предложения `WHERE` передаются удалённому серверу, только если в них используются типы данных, операторы и функции, встроенные в ядро или относящиеся к расширениям, перечисленным в параметре `extensions`. Операторы и функции в таких предложениях также должны быть постоянными (`IMMUTABLE`). Для запросов `UPDATE` или `DELETE` обёртка `postgres_fdw` пытается оптимизировать выполнение, передавая весь запрос на удалённый сервер, если в запросе нет предложения `WHERE`, которое нельзя было бы передать, не выполняется локальное соединение, в целевой таблице отсутствуют хранимые генерируемые столбцы и локальные триггеры `BEFORE`/`AFTER` уровня строки, а в родительских представлениях нет ограничения `CHECK OPTION`. Кроме того, в запросах `UPDATE` выражения, присваиваемые целевым столбцам, должны задействовать только встроенные типы данных и постоянные (`IMMUTABLE`) операторы и функции, чтобы уменьшить риск неверного выполнения запроса.

Когда обёртка `postgres_fdw` обнаруживает соединение сторонних таблиц на одном стороннем сервере, она передаёт всё соединение этому серверу, если только по какой-то причине не решит, что будет эффективнее выбирать строки из каждой таблицы по отдельности, или сопоставляемые при обращении к таблицам пользователи оказываются разными. При передаче предложений `JOIN` принимаются те же меры предосторожности, что были описаны выше для предложений `WHERE`.

Запрос, фактически отправляемый удалённому серверу для выполнения, можно изучить с помощью команды `EXPLAIN VERBOSE`.

F.55.6. Окружение удалённого выполнения запросов

В удалённых сеансах, установленных обёрткой `postgres_fdw`, в параметре search_path задаётся только `pg_catalog`, так что без указания схемы видны только встроенные объекты. Это не проблема для запросов, которые генерирует сама `postgres_fdw`, так как она всегда добавляет такие указания. Однако это может быть опасно для функций, которые выполняются на удалённом сервере при срабатывании триггеров или правил для удалённых таблиц. Например, если удалённая таблица на самом деле представляет собой представление, любые функции, используемые в этом представлении, будут выполняться с таким ограниченным путём поиска. Поэтому рекомендуется в таких функциях дополнять схемой все имена либо добавлять параметры `SET search_path` (см. CREATE FUNCTION) в определения таких функций, чтобы установить ожидаемый ими путь поиска в окружении.

Обёртка `postgres_fdw` подобным образом устанавливает для удалённого сеанса различные параметры:

- TimeZone — `UTC`
- DateStyle — `ISO`
- IntervalStyle — `postgres`
- extra_float_digits принимает значение `3` для удалённых серверов версии 9.0 и новее либо `2` для более старых версий

С ними проблемы менее вероятны, чем с `search_path`, но если они возникнут, их можно урегулировать, установив нужные значения с помощью `SET`.

Это поведение *не* рекомендуется переопределять, устанавливая значения этих параметров на уровне сеанса; это скорее всего приведёт к поломке `postgres_fdw`.

F.55.7. Совместимость с разными версиями

Модуль `postgres_fdw` может применяться с удалёнными серверами версий, начиная с PostgreSQL 8.3. Способность только чтения данных доступна, начиная с 8.1. Однако при этом есть ограничение, вызванное тем, что `postgres_fdw` полагает, что постоянные встроенные функции и операторы могут безопасно передаваться на удалённый сервер для выполнения, если они фигурируют в предложении `WHERE` для сторонней таблицы. Таким образом, встроенная функция, добавленная в более новой версии, чем на удалённом сервере, может быть отправлена на выполнение, что в результате приведёт к ошибке «функция не существует» или подобной. Отказы такого типа можно предотвратить, переписав запрос, например, поместив ссылку на стороннюю таблицу во вложенный `SELECT` с `OFFSET 0` в качестве защиты от оптимизации, и применив проблематичную функцию или оператор снаружи этого вложенного `SELECT`.

F.55.8. События ожидания

`postgres_fdw` может показывать следующие события ожидания в рамках типа `Extension`:

`PostgresFdwCleanupResult`

Ожидание прерывания транзакции на удалённом сервере.

`PostgresFdwConnect`

Ожидание установления подключения к удалённому серверу.

`PostgresFdwGetResult`

Ожидание получения результатов запроса от удалённого сервера.

F.55.9. Параметры конфигурации

`postgres_fdw.application_name` (`string`)

Задаёт значение конфигурационного параметра application_name, используемого при соединении `postgres_fdw` со сторонним сервером. Этот параметр переопределяет свойство `application_name`, заданное для объекта стороннего сервера. Обратите внимание, что изменение этого параметра не затронет существующие подключение, а повлияет только на вновь устанавливаемые.

В `postgres_fdw.application_name` можно задать любую строку произвольной длины, содержащую не только ASCII-символы. Однако обратите внимание, что когда она передаётся стороннему серверу в качестве параметра `application_name`, она усекается до `NAMEDATALEN` символов, а символы не из набора ASCII заменяются шестнадцатеричными спецсимволами в стиле C. За дополнительной информацией обратитесь к application_name.

Символы `%` в этой строке обозначают начало спецпоследовательностей, заменяемых описанной далее информацией. Неизвестные управляющие последовательности игнорируются. Все остальные символы переносятся в имя приложения без изменений. Обратите внимание, что использовать знаки плюс/минус и число между знаком `%` и спецсимволом (чтобы обозначить выравнивание или дополнение) нельзя.

Спецпоследовательность	**Назначение**
`%a`	Имя приложения на локальном сервере
`%c`	ID сеанса на локальном сервере (подробнее см. log_line_prefix)
`%C`	Имя кластера на локальном сервере (подробнее см. cluster_name)
`%u`	Имя пользователя на локальном сервере
`%d`	Имя базы данных на локальном сервере
`%p`	PID обслуживающего процесса на локальном сервере
`%%`	Буквально %

Например, если пользователь `local_user` устанавливает подключение из базы данных `local_db` к базе данных `foreign_db` от имени `foreign_user`, значение `'db=%d, user=%u'` заменяется на `'db=local_db, user=local_user'`.

F.55.10. Примеры

Ниже приведён пример создания сторонней таблицы с применением `postgres_fdw`. Сначала установите расширение:

```
CREATE EXTENSION postgres_fdw;
```

Затем создайте сторонний сервер с помощью команды CREATE SERVER. В данном примере мы хотим подключиться к серверу Postgres Pro, работающему по адресу `192.83.123.89`, порт `5432`. База данных, к которой устанавливается подключение, на удалённом сервере называется `foreign_db`:

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

Для определения роли, которая будет задействована на удалённом сервере, с помощью CREATE USER MAPPING задаётся сопоставление пользователей:

```
CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');
```

Теперь можно создать стороннюю таблицу, применив команду CREATE FOREIGN TABLE. В этом примере мы хотим обратиться к таблице `some_schema.some_table` на удалённом сервере. Локальным именем этой таблицы будет `foreign_table`:

```
CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

Важно, чтобы типы данных и другие свойства столбцов, объявленных в `CREATE FOREIGN TABLE`, соответствовали фактической удалённой таблице. Также должны соответствовать имена столбцов, если только вы не добавите параметры `column_name` для отдельных столбцов, задающие их реальные имена в удалённой таблице. Во многих случаях использовать `IMPORT FOREIGN SCHEMA` предпочтительнее, чем конструировать определения сторонних таблиц вручную.

F.55.11. Автор

Шигеру Ханада `<shigeru.hanada@gmail.com>`

F.56. ptrack — механизм резервного копирования базы Postgres Pro на уровне блоков

PTRACK — это механизм, предназначенный для резервного копирования базы Postgres Pro на уровне блоков. Когда PTRACK включён, программы резервного копирования, в частности pg_probackup, могут использовать его API для получения информации об изменённых блоках, создавая инкрементальные копии. Копируя только блоки, изменённые со времени создания предыдущей копии, можно значительно ускорить процесс создания и минимизировать объём резервных копий.

PTRACK спроектирован так, что он может допускать неточности в положительную сторону, но ошибки в отрицательную сторону исключены: это означает, что все изменения в каталоге данных, не считая изменений вспомогательных битов, гарантированно отмечаются в карте PTRACK, хотя изменёнными могут также считаться и некоторые на самом деле не модифицированные блоки.

F.56.1. Настройка PTRACK

Установив Postgres Pro Standard, выполните следующие действия:

1. Добавьте `ptrack` в параметр shared_preload_libraries в файле `postgresql.conf`:

   ```
   shared_preload_libraries = 'ptrack'
   ```

2. Задайте для параметра ptrack.map_size положительное целое значение.

   Для оптимальной производительности рекомендуется задавать `ptrack.map_size` равным `N / 1024`, где *N* — объём кластера Postgres Pro в мегабайтах. Если этот параметр будет иметь меньшее значение, это увеличит вероятность наложения информации разных блоков в карте PTRACK, что повлечёт ложные положительные результаты при определении изменённых блоков. Вследствие этого в инкрементальную копию могут попасть и неизменённые блоки, от чего её размер может несколько увеличиться.

   Большее значение `ptrack.map_size` не окажет негативного влияния на работу PTRACK, но имейте в виду, что для обеспечения надёжности PTRACK использует два дополнительных временных файла такого же размера, поэтому в общей сложности для PTRACK потребуется `ptrack.map_size * 3` мегабайт дискового пространства. Даже если объём кластера (Postgres Pro *N*) превышает 1 `TB`, значения больше 1024 стоит использовать, только если эффективность и, как следствие, оптимизация размера резервных копий, перевешивает издержки, связанные с увеличением карты PTRACK.

3. Проверьте значение параметра wal_level. Используя PTRACK, необходимо установить для этого параметра значение `replica` или выше. В противном случае вы могли бы потерять изменения в случае аварийной остановки, так как некоторые команды не записывают информацию, если уровень `wal_level` равен `minimal`, а файлы карты PTRACK сбрасываются на диск только во время контрольных точек.

4. Перезапустите экземпляр Postgres Pro Standard, чтобы изменения вступили в силу, а затем создайте расширение PTRACK:

   ```
   CREATE EXTENSION ptrack;
   ```

   В результате этой операции будут созданы функции PTRACK, необходимые для обращения к данным PTRACK.

После запуска сервера механизм PTRACK начинает отслеживать все изменения страниц в кластере Postgres Pro и создаёт файл `ptrack.map`, содержащий последние значения LSN для изменившихся страниц.

> **Примечание**
>
> Параметр `ptrack.map_size` можно задать только при запуске сервера. В случае изменения этого параметра ранее созданная карта PTRACK очищается, и отслеживание изменённых

> блоков начинается заново. Во избежание потери изменений рекомендуется сделать полную резервную копию сразу после присваивания ему другого значения.

F.56.2. Параметры конфигурации PTRACK

`ptrack.map_size` (`integer`)

Задаёт размер файла карты PTRACK и объём выделяемой для него общей памяти в мегабайтах. В карте PTRACK хранятся наибольшие значения LSN для всех страниц кластера Postgres Pro, изменённых с момента включения PTRACK. Для этого параметра не рекомендуется устанавливать значение, превышающее 1 Гбайт. Значение 0 отключает PTRACK и производит очистку всех служебных файлов.

Параметр `ptrack.map_size` можно задать только при запуске сервера. В случае изменения этого параметра ранее созданная карта PTRACK очищается, и отслеживание изменённых блоков начинается заново. Во избежание потери изменений рекомендуется сделать полную резервную копию сразу после присваивания ему другого значения.

По умолчанию: 0

F.56.3. Функции PTRACK

`ptrack_init_lsn() returns pg_lsn`

Возвращает LSN, соответствующий последней инициализации карты PTRACK.

`ptrack_get_pagemapset(`*`начальный_lsn`*` pg_lsn) returns setof record`

Возвращает список файлов данных, изменённых с момента, указанного в параметре *`start_lsn`*, а также количество и битовую карту изменённых страниц для каждого файла.

Например:

```
postgres=# SELECT * FROM ptrack_get_pagemapset('0/185C8C0');
        path         | pagecount |                pagemap
---------------------+-----------+----------------------------------------
 base/16384/1255     |         3 | \x001000000000500000000000
 base/16384/2674     |         3 | \x0000000900010000000000000000
 base/16384/2691     |         1 | \x00004000000000000000000000
 base/16384/2608     |         1 | \x000000000000000400000000000000000000
 base/16384/2690     |         1 | \x000400000000000000000000
```

`ptrack_get_change_stat(`*`начальный_lsn`*` pg_lsn) returns record`

Возвращает статистику изменений (количество файлов, количество изменённых страниц и их общий размер в мегабайтах) с момента, указанного в параметре *`start_lsn`*.

Например:

```
postgres=# SELECT * FROM ptrack_get_change_stat('0/285C8C8');
 files | pages |        size, MB
-------+-------+------------------------
    20 |    25 | 0.1953125000000000000
```

`ptrack_version() returns text`

Выдаёт версию PTRACK.

F.57. rum — метод доступа `RUM` для работы с индексами

F.57.1. Введение

Модуль **rum** предоставляет метод доступа `RUM` для работы с индексами. Он основан на коде методов доступа `GIN`.

Индексы `GIN` позволяют выполнять быстрый полнотекстовый поиск, используя типы `tsvector` и `tsquery`. Однако могут возникнуть некоторые проблемы с производительностью из-за того, что информация о позициях лексем и другая дополнительная информация не сохраняются.

Индекс `RUM` решает эти проблемы, сохраняя дополнительную информацию в дереве идентификаторов и обладает следующими преимуществами перед `GIN`:

- Ускорение ранжирования. Для ранжирования необходима информация о позициях лексем. После сканирования индекса `RUM` нет необходимости в сканировании собственно данных, так как индекс сохраняет позиции лексем.
- Ускорение поиска фраз. Это преимущество связано с предыдущим, так как для поиска фраз также необходима информация о позициях лексем.
- Ускорение упорядочивания по меткам времени. Вместе с лексемами индекс `RUM` сохраняет дополнительную информацию, поэтому не требуется сканирование собственно данных.
- Возможность производить поиск «сначала в глубину», а значит сразу выдавать первые результаты.

Недостаток `RUM` состоит в том, что он строится и изменяется медленнее, чем `GIN`, потому что `RUM` хранит помимо ключей дополнительную информацию и использует унифицированные записи WAL.

F.57.2. Установка

`rum` — это обычное расширение Postgres Pro Standard без каких-либо особых предварительных требований.

Процедура установки выглядит следующим образом:

```
$ psql имя_бд -c "CREATE EXTENSION rum"
```

F.57.3. Общие операторы

Реализованные в модуле `rum` операторы перечислены в Таблице F.35:

Таблица F.35. Операторы `rum`

Оператор	Возвращает	Описание
`tsvector <=> tsquery`	`float4`	Возвращает расстояние между значениями `tsvector` и `tsquery`.
`timestamp <=> timestamp`	`float8`	Возвращает расстояние между двумя значениями `timestamp`.
`timestamp <=| timestamp`	`float8`	Возвращает расстояние только для возрастающих значений `timestamp`.
`timestamp |=> timestamp`	`float8`	Возвращает расстояние только для убывающих значений `timestamp`.

Примечание

`rum` вводит собственную функцию ранжирования, которая выполняется внутри оператора `<=>` и вычисляет оценку (инвертированное расстояние), используя заданный метод реализации. Она представляет собой сочетание функций `ts_rank` и `ts_rank_cd` (за подробностями обратитесь к Разделу 9.13). В то время, как `ts_rank` не поддерживает логические операции, а `ts_rank_cd` плохо работает с запросами `OR`, доступная в `rum` функция ранжирования лишена этих недостатков.

F.57.4. Классы операторов

Расширение `rum` предоставляет следующие классы операторов:

`rum_tsvector_ops`

Сохраняет лексемы `tsvector` с информацией о позициях. Поддерживает упорядочивание с оператором `<=>` и поиск по префиксу.

`rum_tsvector_hash_ops`

Сохраняет хеш лексем `tsvector` с информацией о позициях. Поддерживает упорядочивание с оператором `<=>`, но *не* поддерживает поиск по префиксу.

`rum_tsvector_addon_ops`

Сохраняет лексемы `tsvector` с дополнительными данными любых типов, которые принимает `RUM`.

Примечание

Чтобы использовать классы операторов `rum_tsvector_addon_ops`, при создании индекса `RUM` командой `CREATE INDEX` задайте параметры хранения attach и to в предложении `WITH`.

`rum_tsvector_hash_addon_ops`

Сохраняет лексемы `tsvector` с дополнительными данными любых типов, которые принимает `RUM`. Не поддерживает поиск по префиксу.

`rum_tsquery_ops`

Сохраняет ветви дерева запроса в дополнительной информации.

`rum_anyarray_ops`

Сохраняет элементы массива `anyarray` и длину массива. Поддерживает упорядочивание с оператором <=>.

Индексируемые операторы: `&& @> <@ = %`

`rum_anyarray_addon_ops`

Сохраняет элементы `anyarray` с дополнительными данными любых типов, которые принимает `RUM`.

`rum_`*`type`*`_ops`

Сохраняет лексемы соответствующего типа с информацией о позициях. В качестве *типа* в имени класса должно подставляться одно из следующих имён типов: `int2`, `int4`, `int8`, `float4`, `float8`, `money`, `oid`, `timestamp`, `timestamptz`, `time`, `timetz`, `date`, `interval`, `macaddr`, `inet`, `cidr`, `text`, `varchar`, `char`, `bytea`, `bit`, `varbit`, `numeric`.

Класс операторов `rum_`*`тип`*`_ops` поддерживает упорядочивание с операторами `<=>`, `<=|` и `|=>`. Его можно использовать совместно с классами операторов `rum_tsvector_addon_ops`, `rum_tsvector_hash_addon_ops` и `rum_anyarray_addon_ops`.

Поддержка индексируемых операторов зависит от типа данных:

- Операторы `< <= = >= > <=> <=| |=>` поддерживаются для типов `int2`, `int4`, `int8`, `float4`, `float8`, `money`, `oid`, `timestamp`, `timestamptz`.
- Операторы `< <= = >= >` поддерживаются для типов `time`, `timetz`, `date`, `interval`, `macaddr`, `inet`, `cidr`, `text`, `varchar`, `char`, `bytea`, `bit`, `varbit`, `numeric`.

> **Примечание**
>
> Следующие классы операторов теперь считаются устаревшими: `rum_tsvector_timestamp_ops`, `rum_tsvector_timestamptz_ops`, `rum_tsvector_hash_timestamp_ops`, `rum_tsvector_hash_timestamptz_ops`.

F.57.5. Примеры

F.57.5.1. Пример с rum_tsvector_ops

Предположим, что у нас есть таблица:

```
CREATE TABLE test_rum(t text, a tsvector);

CREATE TRIGGER tsvectorupdate
BEFORE UPDATE OR INSERT ON test_rum
FOR EACH ROW EXECUTE PROCEDURE tsvector_update_trigger('a', 'pg_catalog.english', 't');

INSERT INTO test_rum(t) VALUES ('The situation is most beautiful');
INSERT INTO test_rum(t) VALUES ('It is a beautiful');
INSERT INTO test_rum(t) VALUES ('It looks like a beautiful place');
```

Затем мы можем создать новый индекс:

```
CREATE INDEX rumidx ON test_rum USING rum (a rum_tsvector_ops);
```

И выполнять следующие запросы:

```
SELECT t, a <=> to_tsquery('english', 'beautiful | place') AS rank
    FROM test_rum
    WHERE a @@ to_tsquery('english', 'beautiful | place')
    ORDER BY a <=> to_tsquery('english', 'beautiful | place');
                t                 |   rank
---------------------------------+-----------
 The situation is most beautiful | 0.0303964
 It is a beautiful               | 0.0303964
 It looks like a beautiful place | 0.0607927
(3 rows)

SELECT t, a <=> to_tsquery('english', 'place | situation') AS rank
    FROM test_rum
    WHERE a @@ to_tsquery('english', 'place | situation')
    ORDER BY a <=> to_tsquery('english', 'place | situation');
                t                 |   rank
---------------------------------+-----------
 The situation is most beautiful | 0.0303964
 It looks like a beautiful place | 0.0303964
(2 rows)
```

F.57.5.2. Пример с rum_tsvector_addon_ops

Предположим, что у нас есть таблица:

```
CREATE TABLE tsts (id int, t tsvector, d timestamp);

\copy tsts from 'rum/data/tsts.data'

CREATE INDEX tsts_idx ON tsts USING rum (t rum_tsvector_addon_ops, d)
    WITH (attach = 'd', to = 't');
```

С ним мы можем выполнять подобные запросы:

```
EXPLAIN (costs off)
    SELECT id, d, d <=> '2016-05-16 14:21:25' FROM tsts WHERE t @@ 'wr&qh' ORDER BY d
 <=> '2016-05-16 14:21:25' LIMIT 5;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Limit
   ->  Index Scan using tsts_idx on tsts
         Index Cond: (t @@ '''wr'' & ''qh'''::tsquery)
         Order By: (d <=> 'Mon May 16 14:21:25 2016'::timestamp without time zone)
(4 rows)

SELECT id, d, d <=> '2016-05-16 14:21:25' FROM tsts WHERE t @@ 'wr&qh' ORDER BY d <=>
 '2016-05-16 14:21:25' LIMIT 5;
 id  |                d                |   ?column?
-----+---------------------------------+---------------
 355 | Mon May 16 14:21:22.326724 2016 |      2.673276
 354 | Mon May 16 13:21:22.326724 2016 |   3602.673276
 371 | Tue May 17 06:21:22.326724 2016 |  57597.326724
 406 | Wed May 18 17:21:22.326724 2016 | 183597.326724
 415 | Thu May 19 02:21:22.326724 2016 | 215997.326724
(5 rows)
```

F.57.5.3. Пример с rum_tsquery_ops

Предположим, что у нас таблица:

```
CREATE TABLE query (q tsquery, tag text);

INSERT INTO query VALUES ('supernova & star', 'sn'),
    ('black', 'color'),
    ('big & bang & black & hole', 'bang'),
    ('spiral & galaxy', 'shape'),
    ('black & hole', 'color');

CREATE INDEX query_idx ON query USING rum(q);
```

Мы можем выполнить следующий быстрый запрос:

```
SELECT * FROM query
    WHERE to_tsvector('black holes never exists before we think about them') @@ q;
        q         |  tag
------------------+-------
 'black'          | color
 'black' & 'hole' | color
(2 rows)
```

F.57.5.4. Пример с rum_anyarray_ops

Предположим, что у нас есть таблица:

```
CREATE TABLE test_array (i int2[]);

INSERT INTO test_array VALUES ('{}'), ('{0}'), ('{1,2,3,4}'), ('{1,2,3}'), ('{1,2}'),
 ('{1}');

CREATE INDEX idx_array ON test_array USING rum (i rum_anyarray_ops);
```

Теперь мы можем выполнить следующий запрос, используя сканирование индекса:

```
SET enable_seqscan TO off;

EXPLAIN (COSTS OFF) SELECT * FROM test_array WHERE i && '{1}' ORDER BY i <=> '{1}' ASC;
                QUERY PLAN
------------------------------------------
 Index Scan using idx_array on test_array
   Index Cond: (i && '{1}'::smallint[])
   Order By: (i <=> '{1}'::smallint[])
(3 rows)

SELECT * FROM test_array WHERE i && '{1}' ORDER BY i <=> '{1}' ASC;
     i
-----------
 {1}
 {1,2}
 {1,2,3}
 {1,2,3,4}
(4 rows)
```

F.57.6. Авторы

Александр Коротков

Олег Бартунов

Фёдор Сигаев

F.58. seg — тип данных для отрезков или интервалов чисел с плавающей точкой

Этот модуль реализует тип данных `seg` для представления отрезков или интервалов чисел с плавающей точкой. Тип `seg` может выражать отсутствие уверенности в границах интервала, что позволяет применять его для представления лабораторных измерений.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.58.1. Обоснование

Геометрия измерений обычно более сложна, чем точка в числовом континууме. Измерение обычно представляет собой отрезок этого континуума с нечёткими границами. Измеряемые показатели выражаются интервалами вследствие неопределённости и случайности, а также того, что измеряемое значение может отражать некоторое условие, например, диапазон температур стабильности протеина.

Руководствуясь только здравым смыслом, кажется более удобным хранить такие данные в виде интервалов, а не в виде двух отдельных чисел. На практике это оказывается даже эффективнее в большинстве приложений.

Более того, вследствие нечёткости границ использование традиционных числовых типов данных приводит к определённой потере информации. Рассмотрим такой пример: ваш инструмент выдаёт 6.50 и вы вводите это значение в базу данных. Что вы получите, прочитав это значение из базы? Смотрите:

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

В мире измерений, 6.50 — не то же самое, что 6.5. И разница между этими измерениями иногда бывает критической. Экспериментаторы обычно записывают (и публикуют) цифры, которые заслуживают доверия. Запись 6.50 на самом деле представляет неточный интервал, содержащийся внутри большего и ещё более неточного интервала, 6.5, и единственное, что у них может быть общего, это их центральные точки. Поэтому мы определённо не хотим, чтобы такие разные элементы данных выглядели одинаково.

Вывод? Удобно иметь специальный тип данных, в котором можно сохранить границы интервала с произвольной переменной точностью. В данном случае точность переменная в том смысле, что для каждого элемента данных она может записываться индивидуально.

Проверьте это:

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
6.25 .. 6.50
(1 row)
```

F.58.2. Синтаксис

Внешнее представление интервала образуется одним или двумя числами с плавающей точкой, соединёнными оператором диапазона (`..` или `...`). Кроме того, интервал можно задать центральной точкой плюс/минус отклонение. Также этот тип позволяет сохранить дополнительные индикаторы достоверности (<, > или ~). (Однако индикаторы достоверности игнорируются всеми встроенными операторами.) Допустимые представления показаны в Таблице F.36; некоторые примеры приведены в Таблице F.37.

В Таблице F.36 символы *`x`*, *`y`* и *`delta`* обозначают числа с плавающей точкой. Перед значениями *`x`* и *`y`*, но не *`delta`*, может быть добавлен индикатор достоверности.

Таблица F.36. Внешнее представление `seg`

`x`	Одно значение (интервал нулевой длины)
`x` `..` *`y`*	Интервал от *`x`* до *`y`*
`x` `(+-)` *`delta`*	Интервал от *`x`* - *`delta`* до *`x`* + *`delta`*
`x` `..`	Открытый интервал с нижней границей *`x`*
`..` *`x`*	Открытый интервал с верхней границей *`x`*

Таблица F.37. Примеры допустимых вводимых значений `seg`

`5.0`	Создаёт сегмент нулевой длины (или точку, если хотите)
`~5.0`	Создаёт сегмент нулевой длины и записывает `~` в данные. Знак `~` игнорируется при операциях с `seg`, но сохраняется как комментарий.
`<5.0`	Создаёт точку с координатой 5.0. Знак `<` игнорируется, но сохраняется как комментарий.
`>5.0`	Создаёт точку с координатой 5.0. Знак `>` игнорируется, но сохраняется как комментарий.
`5(+-)0.3`	Создаёт интервал `4.7 .. 5.3`. Заметьте, что запись `(+-)` не сохраняется.
`50 ..`	Всё, что больше или равно 50
`.. 0`	Всё, что меньше или равно 0
`1.5e-2 .. 2E-2`	Создаёт интервал `0.015 .. 0.02`
`1 ... 2`	То же, что и `1...2`, либо `1 .. 2`, либо `1..2` (пробелы вокруг оператора диапазона игнорируются)

Так как оператор `...` часто используется в источниках данных, он принимается в качестве альтернативного написания оператора `..`. К сожалению, это порождает неоднозначность при разборе: неясно, какая верхняя граница имеется в виду в записи `0...23` — `23` или `0.23`. Для разрешения этой неоднозначности во входных числах `seg` перед десятичной точкой всегда должна быть минимум одна цифра.

В качестве меры предосторожности, `seg` не принимает интервалы с нижней границей, превышающей верхнюю, например: `5 .. 2`.

F.58.3. Точность

Значения `seg` хранятся внутри как пары 32-битных чисел с плавающей точкой. Это значит, что числа с более чем 7 значащими цифрами будут усекаться.

Числа, содержащие 7 и меньше значащих цифр, сохраняют изначальную точность. То есть, если запрос возвращает 0.00, вы можете быть уверены, что конечные нули не являются артефактами форматирования: они отражают точность исходных данных. Количество ведущих нулей не влияет на точность: значение 0.0067 будет считаться имеющим только две значащих цифры.

F.58.4. Использование

Модуль `seg` включает класс операторов индекса GiST для значений `seg`. Операторы, поддерживаемые этим классом операторов, перечислены в Таблице F.38.

Таблица F.38. Операторы seg для GiST

Оператор Описание
`seg << seg → boolean` Первый `seg` полностью находится левее второго? [a, b] << [c, d] — true, если b < c.
`seg >> seg → boolean` Первый `seg` полностью находится правее второго? [a, b] >> [c, d] — true, если a > d.
`seg &< seg → boolean` Первый `seg` не простирается правее второго? [a, b] &< [c, d] — true, если b <= d.
`seg &> seg → boolean` Первый `seg` не простирается левее второго? [a, b] &> [c, d] — true, если a >= c.
`seg = seg → boolean` Два отрезка `seg` равны?
`seg && seg → boolean` Два отрезка `seg` пересекаются?
`seg @> seg → boolean` Первый `seg` содержит второй?
`seg <@ seg → boolean` Первый `seg` содержится во втором?

Также для типа `seg` поддерживаются стандартные операторы сравнения, показанные в Таблица 9.1. Эти операторы сначала сравнивают (a) с (c), и если они равны, сравнивают (b) с (d). Результат сравнения позволяет упорядочить значения образом, подходящим для большинства случаев, что полезно, если вы хотите применять ORDER BY с этим типом.

F.58.5. Примечания

Примеры использования можно увидеть в регрессионном тесте `sql/seg.sql`.

Механизм, преобразующий `(+-)` в обычные диапазоны, не вполне точно определяет число значащих цифр для границ. Например, он добавляет дополнительную цифру к нижней границе, если результирующий интервал включает степень десяти:

```
postgres=> select '10(+-)1'::seg as seg;
      seg
---------
9.0 .. 11             -- должно быть: 9 .. 11
```

Производительность индекса-R-дерева может значительно зависеть от начального порядка вводимых значений. Может быть очень полезно отсортировать входную таблицу по столбцу `seg`; пример можно найти в скрипте `sort-segments.pl`.

F.58.6. Благодарности

Первый автор: Джин Селков мл. `<selkovjr@mcs.anl.gov>`, Аргоннская национальная лаборатория, Отдел математики и компьютерных наук

Я очень благодарен в первую очередь профессору Джо Геллерштейну (*https://dsf.berkeley.edu/jmh/*) за пояснение сути GiST (*http://gist.cs.berkeley.edu/*). Я также признателен всем разработчикам Postgres в настоящем и прошлом за возможность создать свой собственный мир и спокойно жить в нём. Ещё я хотел бы выразить признательность Аргоннской лаборатории и Министерству энергетики США за годы постоянной поддержки моих исследований в области баз данных.

F.59. sepgsql — проверка безопасности на базе политик SELinux для мандатного управления доступом (MAC, Mandatory Access Control) с метками

Загружаемый модуль `sepgsql` поддерживает мандатное управление доступом (MAC, Mandatory Access Control) с метками, построенное на базе политик безопасности SELinux.

Предупреждение

Текущая реализация имеет существенные ограничения и контролирует не все действия. См. Подраздел F.59.7.

F.59.1. Обзор

Этот модуль интегрируется в SELinux и обеспечивает дополнительный уровень проверок безопасности, расширяющий и дополняющий обычные средства Postgres Pro. С точки зрения SELinux, данный модуль позволяет Postgres Pro выполнять роль менеджера объектов в пространстве пользователя. При выполнении запроса DML обращение к каждой таблице или функции в нём будет контролироваться согласно системной политике безопасности. Эта проверка дополняет штатную проверку разрешений SQL, которую производит Postgres Pro.

Механизм SELinux принимает решения о разрешении доступа на основе меток безопасности, представляемых строками вида `system_u:object_r:sepgsql_table_t:s0`. В каждом решении учитываются две метки: метка субъекта, пытающегося выполнить действие, и метка объекта, над которым должно совершаться это действие. Так как эти метки могут применяться к объекту любого вида, решения о разрешении доступа к объектам внутри базы данных могут подчиняться (и с этим модулем фактически подчиняются) общим критериям, применяемым к объектам любого другого типа, например, к файлам. Эта схема позволяет организовать централизованную политику безопасности для защиты информационных активов, не зависящую от того, как именно хранятся эти активы.

Назначить метку безопасности объекту баз данных позволяет команда `SECURITY LABEL`.

F.59.2. Установка

Модуль `sepgsql` может работать только в Linux 2.6.28 и новее с включённым SELinux. На остальных платформах он не поддерживается. Вам также понадобится libselinux 2.1.10 или новее и selinux-policy 3.9.13 или новее (хотя в некоторых дистрибутивах необходимые правила могут быть адаптированы к политике старой версии).

Команда `sestatus` позволяет проверить состояние SELinux. Типичный её вывод выглядит так:

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
```

Если SELinux отключён или не установлен, его необходимо привести в рабочее состояние, прежде чем устанавливать этот модуль.

Чтобы собрать этот модуль, укажите параметр `--with-selinux` (если используете make и autoconf) или `Dselinux` (если используете Meson). Убедитесь, что в момент сборки установлен RPM-пакет `libselinux-devel`.

Чтобы использовать этот модуль, вы должны включить `sepgsql` в shared_preload_libraries в `postgresql.conf`. Этот модуль не будет корректно работать, если загрузить его каким-либо другим способом. Загрузив его, нужно выполнить `sepgsql.sql` в каждой базе данных. Этот скрипт установит функции, необходимые для управления метками безопасности, и назначит начальные метки безопасности.

Следующий пример показывает, как инициализировать новый кластер баз данных и установить в него функции и метки безопасности `sepgsql`. Измените пути в соответствии с размещением вашей инсталляции:

```
$ export PGDATA=/path/to/data/directory
$ initdb
$ vi $PGDATA/postgresql.conf
  изменить
    #shared_preload_libraries = ''                # (после изменения требуется
 перезапуск)
  на
    shared_preload_libraries = 'sepgsql'          # (после изменения требуется
 перезапуск)
$ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME \
      </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
  done
```

Заметьте, что вы можете увидеть следующие уведомления, в зависимости от конкретных установленных версий libselinux и selinux-policy:

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 33 has invalid object type
 db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 36 has invalid object type
 db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 37 has invalid object type
 db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 38 has invalid object type
 db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 39 has invalid object type
 db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 40 has invalid object type
 db_language
```

Эти сообщения не критичны и их можно игнорировать.

Если процесс установки завершается без ошибок, вы можете запустить сервер обычным образом.

F.59.3. Регрессионные тесты

Природа SELinux такова, что для проведения регрессионных тестов `sepgsql` требуются дополнительные действия по настройке и некоторые из них должен выполнять root. Регрессионные тесты не будут запускаться обычной командой `make check` или `make installcheck`; вы должны настроить конфигурацию и затем вызвать тестовый скрипт вручную. Тесты должны запускаться в каталоге `contrib/sepgsql` настроенного дерева сборки Postgres Pro. Хотя им требуется дерево сборки, эти тесты рассчитаны на использование установленного сервера, то есть они примерно соответствуют `make installcheck`, но не `make check`.

Сначала установите `sepgsql` в рабочую базу данных по инструкциям, приведённым в Подразделе F.59.2. Заметьте, что для этого текущий пользователь операционной системы должен подключаться к базе данных как суперпользователь без аутентификации по паролю.

На втором шаге соберите и установите пакет политики для регрессионного теста. Политика `sepgsql-regtest` представляет собой политику особого назначения, предоставляющую набор правил, включаемых во время регрессионных тестов. Её следует скомпилировать из исходного файла `sepgsql-regtest.te`, что можно сделать командой `make` со скриптом Makefile, поставляемым с

SELinux. Вам нужно будет найти нужный Makefile в своей системе; путь, показанный ниже, приведён только в качестве примера. (Этот Makefile обычно распространяется в RPM-пакете `selinux-policy-devel` или `selinux-policy`.) Скомпилировав пакет политики, его нужно установить с помощью команды `semodule`, которая загружает переданные ей пакеты в ядро. Если пакет установлен корректно, команда `semodule -l` должна вывести `sepgsql-regtest` в списке доступных пакетов политик:

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

На третьем шаге включите параметр `sepgsql_regression_test_mode`. По соображениям безопасности, правила в `sepgsql-regtest` по умолчанию неактивны; параметр `sepgsql_regression_test_mode` активирует правила, необходимые для проведения регрессионных тестов. Включить этот параметр можно командой `setsebool`:

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

На четвёртом шаге убедитесь в том, что ваша оболочка работает в домене `unconfined_t`:

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

Если необходимо сменить рабочий домен, в подробностях это описывается в Подразделе F.59.8.

Наконец, запустите скрипт регрессионного теста:

```
$ ./test_sepgsql
```

Этот скрипт попытается проверить, все ли шаги по настройке конфигурации выполнены корректно, а затем запустит регрессионные тесты для модуля `sepgsql`.

Завершив тесты, рекомендуется отключить параметр `sepgsql_regression_test_mode`:

```
$ sudo setsebool sepgsql_regression_test_mode off
```

Другой, возможно, более предпочтительный вариант — удалить политику `sepgsql-regtest` полностью:

```
$ sudo semodule -r sepgsql-regtest
```

F.59.4. Параметры GUC

`sepgsql.permissive` (`boolean`)

Этот параметр переводит `sepgsql` в разрешительный режим, вне зависимости от режима системы. По умолчанию он имеет значение off (отключён). Задать этот параметр можно только в `postgresql.conf` или в командной строке при запуске сервера.

Когда этот параметр включён, `sepgsql` действует в разрешительном режиме, даже если SELinux в целом находится в ограничительном режиме. Этот параметр полезен в первую очередь для тестирования.

`sepgsql.debug_audit` (`boolean`)

Этот параметр включает вывод сообщений аудита вне зависимости от параметров системной политики. По умолчанию он отключён (имеет значение off), что означает, что сообщения будут выводиться согласно параметрам системы.

Политики безопасности SELinux также содержит правила, определяющие, будут ли фиксироваться в журнале определённые события. По умолчанию фиксируются нарушения доступа, а успешный доступ — нет.

Этот параметр принудительно включает фиксирование в журнале всех возможных событий, вне зависимости от системной политики.

F.59.5. Функциональные возможности

F.59.5.1. Управляемые классы объектов

Модель безопасности SELinux описывает все правила доступа в виде отношений между сущностью субъекта (обычно, это клиент базы данных) и сущностью объекта (например, объектом базы данных), каждая из которых определяется меткой безопасности. Если осуществляется попытка доступа к непомеченному объекту, он обрабатывается как объект, имеющий метку `unlabeled_t`.

В настоящее время `sepgsql` позволяет назначать метки безопасности схемам, таблицам, столбцам, последовательностям, представлениям и функциям. Когда `sepgsql` активен, метки безопасности автоматически назначаются поддерживаемым объектам базы в момент создания. Такая метка называется меткой безопасности по умолчанию и устанавливается согласно политике безопасности системы, которая учитывает метку создателя, метку, назначенную родительскому объекту создаваемого объекта и, возможно, имя создаваемого объекта.

Новый объект базы, как правило, наследует метку безопасности, назначенную родительскому объекту, если только в политике безопасности не заданы специальные правила, называемые правилами перехода типов (в этом случае может быть назначена другая метка). Для схем родительским объектом является текущая база данных; для таблиц, последовательностей, представлений и функций — схема, содержащая эти объекты; для столбцов — таблица.

F.59.5.2. Разрешения для DML

Для таблиц, задействованных в запросе в качестве целевых, проверяются разрешения `db_table:select`, `db_table:insert`, `db_table:update` или `db_table:delete` в зависимости от типа оператора; кроме того, для всех таблиц, содержащих столбцы, фигурирующие в предложении `WHERE` или `RETURNING`, или служащих источником данных для `UPDATE` и т. п., также проверяется разрешение `db_table:select`.

Для всех задействованных столбцов также проверяются разрешения на уровне столбцов. Разрешение `db_column:select` проверяется не только для столбцов, которые считываются оператором `SELECT`, но и для тех, к которым обращаются другие операторы DML; `db_column:update` или `db_column:insert` также проверяется для столбцов, изменяемых операторами `UPDATE` или `INSERT`.

Например, рассмотрим запрос:

```
UPDATE t1 SET x = 2, y = func1(y) WHERE z = 100;
```

В данном случае `db_column:update` будет проверяться для столбца `t1.x`, так как он изменяется, `db_column:{select update}` будет проверяться для `t1.y`, так как он и считывается, и изменяется, а `db_column:select` — для столбца `t1.z`, так как он только считывается. На уровне таблицы также будет проверяться разрешение `db_table:{select update}`.

Для последовательностей проверяется разрешение `db_sequence:get_value`, когда имеет место обращение к объекту последовательности в `SELECT`; заметьте, однако, что в настоящее время разрешения на выполнение связанных функций, таких как, `lastval()`, не проверяются.

Для представлений проверяется `db_view:expand`, а затем все другие соответствующие разрешения для объектов, развёрнутых из определения представления, в индивидуальном порядке.

Для функций проверяется `db_procedure:{execute}`, когда пользователь пытается выполнить функцию в составе запроса, либо при вызове по быстрому пути. Если эта функция является доверенной процедурой, также проверяется разрешение `db_procedure:{entrypoint}`, чтобы удостовериться, что эта функция может быть точкой входа в доверенную процедуру.

При обращении к любому объекту схемы необходимо иметь разрешение `db_schema:search` для содержащей его схемы. Когда имя целевого объекта не дополняется схемой, схемы, для которых данное разрешение отсутствует, не будут просматриваться (то же происходит, если у пользователя нет права `USAGE` для этой схемы). Когда схема указывается явно, пользователь получит ошибку, если он не имеет требуемого разрешения для доступа к указанной схеме.

Клиенту должен быть разрешён доступ ко всем задействованным в запросе таблицам и столбцам, даже если они проявились в нём в результате разворачивания представлений, так что правила применяются согласованно вне зависимости от варианта обращения к содержимому таблиц.

Стандартная система привилегий позволяет суперпользователям баз данных изменять системные каталоги с помощью команд DML и обращаться к таблицам TOAST или модифицировать их. Когда модуль `sepgsql` активен, эти операции запрещаются.

F.59.5.3. Разрешения для DDL

SELinux определяет набор разрешений для управления стандартными операциями для каждого типа объекта: создание, изменение определения, удаление и смена метки безопасности. В дополнение к ним для некоторых типов объектов предусмотрены специальные разрешения для управления их специфическими операциями, как например, добавление или удаление объектов в определённой схеме.

Для создания нового объекта базы данных требуется разрешение `create`. SELinux разрешает или запрещает выполнение этой операции в зависимости от метки безопасности клиента и предполагаемой метки безопасности нового объекта. В некоторых случаях требуются дополнительные разрешения:

- `CREATE DATABASE` дополнительно требует разрешения `getattr` в исходной или шаблонной базе данных.
- Создание объекта схемы дополнительно требует разрешения `add_name` в родительской схеме.
- Создание таблицы дополнительно требует разрешения на создание каждой отдельного столбца таблицы, как если бы каждый столбец таблицы был отдельным объектом верхнего уровня.
- Создание функции с атрибутом `LEAKPROOF` дополнительно требует разрешения `install`. (Это разрешение также проверяется, когда атрибут `LEAKPROOF` устанавливается для существующей функции.)

Когда выполняется команда `DROP`, для удаляемого объекта будет проверяться разрешение `drop`. Разрешения будут также проверяться и для объектов, удаляемых косвенно, вследствие указания `CASCADE`. Для удаления объектов, содержащихся в определённой схеме, (таблиц, представления, последовательностей и процедур) дополнительно нужно иметь разрешение `remove_name` в этой схеме.

Когда выполняется команда `ALTER`, для каждого модифицируемого объекта проверяется разрешение `setattr`, кроме подчинённых объектов, таких как индексы или триггеры таблиц (на них распространяются разрешения родительского объекта). В некоторых случаях требуются дополнительные разрешения:

- При перемещении объекта в новую схему дополнительно требуется разрешение `remove_name` в старой схеме и `add_name` в новой.
- Для установки атрибута `LEAKPROOF` для функции требуется разрешение `install`.
- Для использования `SECURITY LABEL` дополнительно требуется разрешение `relabelfrom` для объекта с его старой меткой безопасности и `relabelto` для этого объекта с новой меткой безопасности. (В случаях, когда установлено несколько поставщиков меток и пользователь пытается задать метку, неподконтрольную SELinux, должно проверяться только разрешение `setattr`. В настоящее время этого не происходит из-за ограничений реализации.)

F.59.5.4. Доверенные процедуры

Доверенные процедуры похожи на функции, определяющие контекст безопасности, или команды setuid. В SELinux реализована возможность запускать доверенный код с меткой безопасности, отличной от метки клиента, как правило, для предоставления чётко контролируемого доступа к важным данным (при этом например, могут отсеиваться строки или хранимые значения могут выводиться с меньшей точностью). Будет ли функция вызываться как доверенная процедура, определяется её меткой безопасности и политикой операционной системы. Например:

```
postgres=# CREATE TABLE customer (
               cid     int primary key,
               cname   text,
               credit  text
           );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
               IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
             AS 'SELECT regexp_replace(credit, ''-[0-9]+$'', ''-xxxx'', ''g'')
                        FROM customer WHERE cid = $1'
           LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
               IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

Показанные выше операции должен выполнять пользователь с правами администратора.

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname  |     show_credit
-----+--------+---------------------
   1 | taro   | 1111-2222-3333-xxxx
   2 | hanako | 5555-6666-7777-xxxx
(2 rows)
```

В данном случае обычный пользователь не может обращаться к `customer.credit` напрямую, но доверенная процедура `show_credit` позволяет ему получить номера кредитных карт клиентов, в которых будут скрыты некоторые цифры.

F.59.5.5. Динамические переключения домена

Возможность динамического перехода из домена в домен SELinux позволяет переводить метку безопасности клиентского процесса, клиентский домен в новый контекст, если это допускается политикой безопасности. Для этого клиент должен иметь разрешение `setcurrent`, а также разрешение `dyntransition` для перехода из старого в новый домен.

Динамические переключения домена следует тщательно продумывать, так как таким образом пользователи могут менять свои метки, а значит и привилегии, по собственному желанию, а не (как в случае с доверенными процедурами) по правилам, диктуемым системой. Таким образом, разрешение `dyntransition` считается безопасным, только когда применяется для переключения в домен с более ограниченным набором привилегий, чем текущий. Например:

```
regression=# select sepgsql_getcon();
                    sepgsql_getcon
-------------------------------------------------------
 unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)
```

```
regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c4');
 sepgsql_setcon
----------------
 t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c1023');
ERROR:  SELinux: security policy violation
```

В показанном выше примере мы смогли переключиться из более широкого диапазона MCS `c1.c1023` в более узкий `c1.c4`, но переключение в обратную сторону было запрещено.

Сочетание динамического переключения домена с доверенными процедурами позволяет получить интересное решение, подходящее для реализации жизненного цикла процессов с пулом соединений. Даже если вашему менеджеру пула соединений не разрешается запускать многие команды SQL, вы можете разрешить ему сменить метку безопасности клиента, вызвав функцию `sepgsql_setcon()` из доверенной процедуры; для этого может передаваться удостоверение для авторизации запроса на смену метки клиента. После этого сеанс получит привилегии целевого пользователя, а не пользователя пула соединений. Позднее менеджер пула может отменить смену контекста безопасности, вызвав `sepgsql_setcon()` с аргументом `NULL`, так же из доверенной процедуры с необходимыми проверками разрешений. Идея этого подхода в том, что только этой доверенной процедуре будет разрешено менять действующую метку безопасности и только в том случае, когда ей передаётся правильное удостоверение. Разумеется, чтобы это решение было безопасным, хранилище удостоверений (таблица, определение процедуры или что-то другое) не должно быть общедоступным.

F.59.5.6. Разное

Выполнение команды `LOAD` в активном режиме запрещается, так как любой загруженный модуль может легко обойти ограничения политики безопасности.

F.59.6. Функции sepgsql

В Таблице F.39 перечислены все доступные функции.

Таблица F.39. Функции sepgsql

Функция Описание
`sepgsql_getcon () → text` Возвращает клиентский домен, текущую метку безопасности клиента.
`sepgsql_setcon ( text ) → boolean` Переключает домен клиента текущего сеанса в новый домен, если это допускает политика безопасности. Эта функция также принимает в аргументе `NULL` как запрос на переход в начальный домен клиента.
`sepgsql_mcstrans_in ( text ) → text` Переводит заданный диапазон MLS/MCS из полной записи в низкоуровневый формат, если работает демон mcstrans.
`sepgsql_mcstrans_out ( text ) → text` Переводит заданный диапазон MLS/MCS из низкоуровневого формата в полную запись, если работает демон mcstrans.
`sepgsql_restorecon ( text ) → boolean` Устанавливает начальные метки безопасности для всех объектов в текущей базе данных. В аргументе может передаваться `NULL` или имя файла со спецификациями контекстов, который будет применяться вместо стандартного системного.

F.59.7. Ограничения

Разрешения для языка определения данных (DDL, Data Definition Language)

Вследствие ограничений реализации, для некоторых операций DDL разрешения не проверяются.

Разрешения для языка управления данными (DCL, Data Control Language)

Вследствие ограничений реализации, для операций DCL разрешения не проверяются.

Управление доступом на уровне строк

Postgres Pro поддерживает ограничение доступа на уровне строк, а `sepgsql` — нет.

Скрытые каналы

Модуль `sepgsql` не пытается скрыть существование определённого объекта, даже если пользователю не разрешено обращаться к нему. Например, возможно догадаться о существовании невидимого объекта по конфликтам первичного ключа, нарушениям внешних ключей и т. д., даже когда нельзя получить содержимое этого объекта. Существование совершенно секретной таблицы невозможно скрыть; надеяться можно только на то, что будет защищено её содержимое.

F.59.8. Внешние ресурсы

SE-PostgreSQL Introduction, Введение в SE-PostgreSQL

На этой вики-странице даётся краткий обзор этого решения и рассказывается об архитектуре и конструкции безопасности, администрировании и ожидаемых в будущем возможностях.

SELinux User's and Administrator's Guide, Руководство пользователя и администратора SELinux

В этом документе представлен широкий спектр знаний по администрированию SELinux в ОС. В первую очередь он ориентирован на системы Red Hat, но его область применения не ограничена ими.

Fedora SELinux FAQ, Часто задаваемые вопросы по SELinux в ОС Fedora

В этом документе даются ответы на часто задаваемые вопросы по SELinux. В первую очередь он ориентирован на ОС Fedora, но его область применения не ограничена ей.

F.59.9. Автор

КайГай Кохэй `<kaigai@ak.jp.nec.com>`

F.60. shared_ispell — разделяемый словарь ispell

Модуль `shared_ispell` предоставляет разделяемый словарь ispell, то есть словарь, расположенный в общем сегменте памяти. В традиционной реализации ispell каждый сеанс инициализирует и загружает собственный экземпляр словаря, и на это уходит много ресурсов процессора и памяти.

Это расширение выделяет область памяти в разделяемом сегменте (вы должны выбрать его размер заранее) и загружает в неё словарь при первом использовании.

F.60.1. Функции

Реализованные в модуле `shared_ispell` функции перечислены в Таблице F.40.

Таблица F.40. Функции `shared_ispell`

Функция	Возвращает	Описание
`shared_ispell_reset()`	`void`	Сбрасывает словари (например, для повторной загрузки изменённых файлов с диска). Сеансы, в которых словари уже использовались, будут должны переинициализировать их.
`shared_ispell_mem_used()`	`int`	Возвращает объём памяти, используемой загруженными словарями в разделяемом сегменте (в байтах).
`shared_ispell_mem_available()`	`int`	Возвращает объём доступной памяти в разделяемом сегменте.
`shared_ispell_dicts()`	`setof(dict_name varchar, affix_name varchar, words int, affixes int, bytes int)`	Возвращает список словарей, загруженных в разделяемый сегмент.
`shared_ispell_stoplists()`	`setof(stop_name varchar, words int, bytes int)`	Возвращает список наборов стоп-слов, загруженных в разделяемый сегмент.

F.60.2. Параметры GUC

`shared_ispell.max_size` (`int`)

Определяет максимальный размер разделяемого сегмента. Это жёсткий предел; разделяемый сегмент нельзя расширить, поэтому нужно установить такой размер, чтобы в этом сегменте уместились все словари и при этом он не занимал лишний объём.

F.60.3. Использование словаря

Этот модуль должен выделить область памяти в разделяемом сегменте. Поэтому добавьте в файл конфигурации следующие параметры (или отредактируйте текущие значения):

```
# загружаемые библиотеки
shared_preload_libraries = 'shared_ispell'

# объём разделяемой памяти
shared_ispell.max_size = 32MB
```

Чтобы понять, сколько памяти вам действительно нужно, задайте сначала большое значение (например, 200 МБ) и загрузите все нужные вам словари. Затем воспользуйтесь функцией `shared_ispell_mem_used()` и определите, сколько памяти на самом деле используется (и скорректируйте соответствующую переменную GUC `shared_ispell.max_size`).

Не задавайте в точности полученное значение, оставьте небольшой резерв, чтобы вы могли перезагружать словари, не изменяя предел GUC max_size (что потребует перезапуска БД). Объёма в районе 512 КБ будет вполне достаточно.

Это расширение определяет шаблон `shared_ispell`, используя который можно определять собственные словари текстового поиска. Например, вы можете сделать следующее:

```
CREATE TEXT SEARCH DICTIONARY english_shared (
    TEMPLATE = shared_ispell,
    DictFile = en_us,
    AffFile = en_us,
    StopWords = english
);

CREATE TEXT SEARCH CONFIGURATION public.english_shared
    ( COPY = pg_catalog.simple );

ALTER TEXT SEARCH CONFIGURATION english_shared
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                word, hword, hword_part
    WITH english_shared, english_stem;
```

Проверить полученную конфигурацию можно так:

```
SELECT * FROM ts_debug('english_shared', 'abilities');
   alias   |   description   |   token   |         dictionaries          |   dictionary
   |  lexemes
-----------+-----------------+-----------+-------------------------------
+----------------+-----------
 asciiword | Word, all ASCII | abilities | {english_shared,english_stem} |
 english_shared | {ability}
(1 row)
```

Вы также можете изменить существующую конфигурацию текстового поиска. Например, если у вас есть конфигурация `public.english`, вы можете подключить к ней шаблон `shared_ispell`:

```
ALTER TEXT SEARCH CONFIGURATION public.english
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                word, hword, hword_part
    WITH english_shared, english_stem;
```

F.60.4. Автор

Томаш Вондра `<tomas.vondra@2ndquadrant.com>`, Прага, Чешская республика

F.61. spi — особенности/примеры использования интерфейса программирования сервера

Модуль spi предоставляет несколько рабочих примеров использования Интерфейса программирования сервера (Server Programming Interface, SPI) и триггеров. Хотя эти функции имеют некоторую ценность сами по себе, они ещё более полезны как заготовки, которые можно приспособить под собственные нужды. Эти функции достаточно общие, чтобы работать с любой таблицей, но вы должны явно указать имена таблицы и полей (как описано ниже) при создании триггера.

Каждая группа функций, описанная ниже, представлена в виде отдельно устанавливаемого расширения.

F.61.1. refint — функции для реализации ссылочной целостности

Функции `check_primary_key()` и `check_foreign_key()` применяются для проверки ограничений внешних ключей. (Эта функциональность уже давно вытеснена встроенным механизмом внешних ключей, но этот модуль всё ещё полезен в качестве примера.)

Функция `check_primary_key()` проверяет ссылающуюся таблицу. Чтобы воспользоваться ей, создайте триггер `BEFORE INSERT OR UPDATE` с этой функцией для таблицы, ссылающейся на другую. Укажите в аргументах триггера: имена столбцов ссылающейся таблицы, образующих внешний ключ, имя целевой таблицы и имена столбцов в ней, образующих первичный/уникальный ключ. Чтобы контролировать несколько внешних ключей, создайте триггер для каждой такой ссылки.

Функция `check_foreign_key()` проверяет целевую таблицу. Чтобы использовать её, создайте триггер `BEFORE DELETE OR UPDATE` с этой функцией для таблицы, на которую ссылаются другие. Укажите в аргументах триггера: число ссылающихся таблиц, для которых функция должна выполнить проверки, действие в случае обнаружения ссылающегося ключа (`cascade` — удалить ссылающуюся строку, `restrict` — прервать транзакцию, `setnull` — установить в ссылающихся полях значения NULL), имена столбцов целевой таблицы, образующих первичный/уникальный ключ, а затем имена таблиц и столбцов (в количестве, задаваемом первым аргументом). Заметьте, что поля первичных/уникальных столбцов должны иметь пометку NOT NULL и по ним должен быть создан уникальный индекс.

Примеры приведены в `refint.example`.

F.61.2. autoinc — функции для автоувеличения полей

Функция `autoinc()` реализует код триггера, сохраняющего следующее значение последовательности в целочисленном поле. Это в некоторой степени пересекается со встроенной функциональностью столбца «serial», но есть и отличия: `autoinc()` препятствует попыткам вставить другое значение поля при добавлении строк и может увеличивать значение поля при изменениях.

Чтобы использовать её, создайте триггер `BEFORE INSERT` (или `BEFORE INSERT OR UPDATE`) с этой функцией. Передайте триггеру два аргумента: имя целочисленного столбца, который будет меняться, и имя объекта последовательности, который будет поставлять значения. (Вообще вы можете задать любое число пар таких имён, если хотите поддерживать несколько автоувеличивающихся столбцов.)

Пример приведён в `autoinc.example`.

F.61.3. insert_username — функции для отслеживания пользователя, вносящего изменения

Функция `insert_username()` реализует код триггера, сохраняющего имя текущего пользователя в текстовом поле. Это может быть полезно для отслеживания пользователя, изменившего конкретную строку таблицы последним.

Чтобы использовать её, создайте триггер `BEFORE INSERT` и/или `UPDATE` с этой функцией. Передайте триггеру один аргумент: имя целевого текстового столбца.

Пример приведён в `insert_username.example`.

F.61.4. moddatetime — функции для отслеживания времени последнего изменения

Функция `moddatetime()` реализует код триггера, сохраняющего текущее время в поле типа `timestamp`. Это может быть полезно для отслеживания времени последней модификации конкретной строки таблицы.

Чтобы использовать её, создайте триггер `BEFORE UPDATE` с этой функцией. Передайте триггеру один аргумент: имя целевого столбца. Столбец должен иметь тип `timestamp` или `timestamp with time zone`.

Пример приведён в `moddatetime.example`.

F.62. sslinfo — получение информации об SSL-сертификате клиента

Модуль `sslinfo` выдаёт информацию о SSL-сертификате, который был представлен текущим клиентом при подключении к Postgres Pro. Этот модуль бесполезен (большинство функций возвратят NULL), если для текущего подключения не задействуется SSL.

Часть информации, выдаваемой этим модулем, можно получить через встроенное системное представление `pg_stat_ssl`.

Это расширение не будет собираться, если конфигурация была произведена без ключа `--with-ssl=openssl`.

F.62.1. Предоставляемые функции

`ssl_is_used() returns boolean`

Возвращает true, если текущее подключение использует SSL, и false в противном случае.

`ssl_version() returns text`

Возвращает имя протокола, по которому организовано SSL-подключение (например TLSv1.0, TLSv1.1, TLSv1.2 или TLSv1.3).

`ssl_cipher() returns text`

Возвращает имя шифра, используемого для SSL-подключения (например, DHE-RSA-AES256-SHA).

`ssl_client_cert_present() returns boolean`

Возвращает true, если текущий клиент предоставил серверу действительный клиентский SSL-сертификат, и false в противном случае. (Сервер может требовать, а может и не требовать предоставления клиентского сертификата.)

`ssl_client_serial() returns numeric`

Возвращает серийный номер текущего клиентского сертификата. Сочетание серийного номера сертификата с выдавшим его центром сертификации гарантирует однозначную идентификацию сертификата (но не его владелец — владелец должен регулярно менять свои ключи и получать сертификаты в центре сертификации).

Поэтому, если вы используете собственный ЦС и настроили сервер, чтобы он принимал сертификаты только от этого ЦС, серийный номер будет наиболее надёжным (хотя не очень запоминающимся) ключом идентификации пользователя.

`ssl_client_dn() returns text`

Возвращает полное имя субъекта из текущего клиентского сертификата, преобразуя символьные данные в кодировку текущей базы данных. Предполагается, что если в именах в сертификатах используются символы вне таблицы ASCII, то ваша база данных может представить эти символы. Если в вашей базе используется кодировка SQL_ASCII, символы вне ASCII в имени будут представлены последовательностями UTF-8.

Результат выглядит примерно так: `/CN=Somebody /C=Some country/O=Some organization`.

`ssl_issuer_dn() returns text`

Возвращает полное имя издателя текущего клиентского сертификата, преобразуя символьные данные в кодировку текущей базы данных. Преобразования кодировки осуществляются так же, как и в `ssl_client_dn`.

Сочетание возвращаемого значения этой функции с серийным номером сертификата однозначно идентифицирует сертификат.

Эта функция полезна, только если в установленном на сервере файле с сертификатами ЦС содержатся сертификаты нескольких ЦС или если один ЦС выдаёт сертификаты для промежуточных центров сертификации.

`ssl_client_dn_field(fieldname text) returns text`

Эта функция возвращает значение указанного поля данных субъекта сертификата, либо NULL, если это поле отсутствует. Имена полей задаются строковыми константами, которые затем преобразуются в идентификаторы объектов ASN1, используя базу данных объектов OpenSSL. Принимаются следующие значения:

```
commonName (или CN)
surname (или SN)
name
givenName (или GN)
countryName (или C)
localityName (или L)
stateOrProvinceName (или ST)
organizationName (или O)
organizationalUnitName (или OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

Все эти поля являются необязательными, за исключением `commonName`. Какие из них будут включены в сертификат, а какие нет, зависит полностью от политики вашего ЦС. Значение этих полей, однако, строго определено стандартами X.500 и X.509, так что их нельзя интерпретировать произвольным образом.

`ssl_issuer_field(fieldname text) returns text`

То же, что `ssl_client_dn_field`, но для издателя, а не для субъекта сертификата.

`ssl_extension_info() returns setof record`

Предоставляет информацию о расширениях клиентского сертификата: имя расширения, значение расширения и является ли это расширение критическим.

F.62.2. Автор

Виктор Вагнер `<vitus@cryptocom.ru>`, ООО «Криптоком»

Дмитрий Воронин `<carriingfate92@yandex.ru>`

Электронный адрес группы разработчиков OpenSSL в Криптокоме: `<openssl@cryptocom.ru>`

F.63. tablefunc — функции, возвращающие таблицы (`crosstab` и не только)

Модуль `tablefunc` содержит ряд функций, возвращающих таблицы (то есть, множества строк). Эти функции полезны и сами по себе, и как примеры написания на C функций, возвращающих наборы строк.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.63.1. Предоставляемые функции

Функции, предоставляемые модулем `tablefunc`, перечислены в Таблице F.41.

Таблица F.41. Функции `tablefunc`

Функция Описание
`normal_rand` (*`numvals`* `integer`, *`mean`* `float8`, *`stddev`* `float8`) → `setof float8` Выдаёт набор случайных значений, имеющих нормальное распределение.
`crosstab` (*`sql`* `text`) → `setof record` Выдаёт «повёрнутую таблицу», содержащую имена строк плюс *`N`* столбцов значений, где *`N`* определяется видом строк, заданным в вызывающем запросе.
`crosstab`*`N`* (*`sql`* `text`) → `setof table_crosstab_` *`N`* Выдаёт «повёрнутую таблицу», содержащую имена строк плюс *`N`* столбцов значений. Функции `crosstab2`, `crosstab3` и `crosstab4` предопределены, но вы можете создать дополнительные функции `crosstab`*`N`*, как описано ниже.
`crosstab` (*`source_sql`* `text`, *`category_sql`* `text`) → `setof record` Выдаёт «повёрнутую таблицу» со столбцами значений, заданными вторым запросом.
`crosstab` (*`sql`* `text`, *`N`* `integer`) → `setof record` Устаревшая версия `crosstab(text)` . Параметр *`N`* теперь игнорируется, так как число столбцов значений всегда определяется вызывающим запросом.
`connectby` (*`relname`* `text`, *`keyid_fld`* `text`, *`parent_keyid_fld`* `text` [, *`orderby_fld`* `text`], *`start_with`* `text`, *`max_depth`* `integer` [, *`branch_delim`* `text`]) → `setof record` Выдаёт представление иерархической древовидной структуры.

F.63.1.1. `normal_rand`

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8
```

Функция `normal_rand` выдаёт набор случайных значений, имеющих нормальное распределение (распределение Гаусса).

Параметр *`numvals`* задаёт количество значений, которое выдаст эта функция. Параметр *`mean`* задаёт медиану нормального распределения, а *`stddev`* — стандартное отклонение.

Например, этот вызов запрашивает 1000 значений с медианой 5 и стандартным отклонением 3:

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
     normal_rand
----------------------
     1.56556322244898
     9.10040991424657
     5.36957140345079
   -0.369151492880995
    0.283600703686639
```

```
       .
       .
       .
     4.82992125404908
     9.71308014517282
     2.49639286969028
(1000 rows)
```

F.63.1.2. `crosstab(text)`

```
crosstab(text sql)
crosstab(text sql, int N)
```

Функция `crosstab` применяется для формирования «повёрнутых» отображений, в которых данные идут вдоль строк, а не сверху вниз. Например, мы можем иметь такие данные:

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

и хотим видеть их так:

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

Функция `crosstab` принимает в текстовом параметре SQL-запрос, выдающий исходные данные первым способом, и выдаёт таблицу, отформатированную вторым способом.

В параметре *`sql`* передаётся SQL-запрос, выдающий исходный набор данных. Этот запрос должен возвращать один столбец `row_name`, один столбец `category` и один столбец `value`. Параметр *`N`* является устаревшим и игнорируется, если передаётся при вызове (раньше он должен был соответствовать количеству выходных столбцов значений, но теперь это количество определяется вызывающим запросом).

Например, заданный запрос может выдавать такой результат:

```
 row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

Функция `crosstab` объявлена как возвращающая `setof record`, так что фактические имена и типы столбцов должны определяться в предложении `FROM` вызывающего оператора `SELECT`, например так:

```
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2 text);
```

Этот запрос выдаст примерно такой результат:

```
           <== столбцы значений ==>
 row_name   category_1   category_2
----------+------------+------------
```

```
  row1         val1          val2
  row2         val5          val6
```

Предложение `FROM` должно определять результат со столбцом `row_name` (того же типа данных, что у первого результирующего столбца SQL-запроса), за которым следуют N столбцов значений (все того же типа данных, что и третий результирующий столбец SQL-запроса). Количество выходных столбцов значений может быть произвольным и имена выходных столбцов определяете вы сами.

Функция `crosstab` выдаёт одну выходную строку для каждой последовательной группы с одним значением `row_name`. Она заполняет столбцы значений слева направо полями `value` из этих строк. Если в группе оказывается меньше строк, чем выходных столбцов значений, дополнительные столбцы принимают значения NULL; если же строк оказывается больше, лишние строки игнорируются.

На практике в SQL-запросе всегда должно указываться `ORDER BY 1,2`, чтобы входные строки были отсортированы должным образом, то есть, чтобы данные с одинаковым значением `row_name` собирались вместе и корректно упорядочивались в строке. Заметьте, что сама `crosstab` не учитывает второй столбец результата запроса; он присутствует только для того, чтобы определять порядок, в котором значения третьего столбца будут следовать в строке.

Полный пример:

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
 test2    | val6       | val7       |
(2 rows)
```

Вы можете в любом случае обойтись без написания предложения `FROM`, определяющего выходные столбцы, создав собственную функцию crosstab, в определении которой будет зашит желательный тип выходной строки. Это описывается в следующем разделе. Также имеется возможность включить требуемое предложение `FROM` в определение представления.

Примечание

Также изучите команду `\crosstabview` в psql, реализующую функциональность, подобную `crosstab()`.

F.63.1.3. `crosstabN(text)`

`crosstabN(text sql)`

Функции `crosstabN` являются примерами того, как можно создать собственные обёртки универсальной функции `crosstab`, чтобы не приходилось выписывать имена и типы столбцов в вызывающем запросе `SELECT`. Модуль `tablefunc` включает функции `crosstab2`, `crosstab3` и `crosstab4`, определяющие типы выходных строк так:

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
        .
        .
        .
    category_N TEXT
);
```

Таким образом, эти функции могут применяться непосредственно, когда входной запрос выдаёт столбцы `row_name` и `value` типа `text` и вы хотите получить на выходе 2, 3 или 4 столбца значений. В остальном эти функции ведут себя в точности так же, как и универсальная функция `crosstab`.

Так, пример, приведённый в предыдущем разделе, можно переписать и в таком виде:

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2');
```

Эти функции представлены в основном в демонстрационных целях. Вы можете создать собственные типы возвращаемых данных и реализовать функции на базе нижележащей функции `crosstab()`. Это можно сделать двумя способами:

- Создать составной тип, описывающий желаемые выходные столбцы, примерно как это делается в примерах в `contrib/tablefunc/tablefunc--1.0.sql`. Затем нужно выбрать уникальное имя для функции, принимающей один параметр `text` и возвращающей `setof имя_вашего_типа`, и связать его с той же нижележащей функцией `crosstab` на C. Например, если ваш источник данных выдаёт имена строк типа `text` и значения типа `float8`, и вы хотите получить 5 столбцов значений:

  ```
  CREATE TYPE my_crosstab_float8_5_cols AS (
      my_row_name text,
      my_category_1 float8,
      my_category_2 float8,
      my_category_3 float8,
      my_category_4 float8,
      my_category_5 float8
  );

  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
      RETURNS setof my_crosstab_float8_5_cols
      AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
  ```

- Использовать выходные параметры (`OUT`), чтобы явно определить возвращаемый тип. Тот же пример можно реализовать и таким способом:

  ```
  CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
      IN text,
      OUT my_row_name text,
      OUT my_category_1 float8,
      OUT my_category_2 float8,
  ```

```
    OUT my_category_3 float8,
    OUT my_category_4 float8,
    OUT my_category_5 float8)
  RETURNS setof record
  AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

F.63.1.4. `crosstab(text, text)`

```
crosstab(text source_sql, text category_sql)
```

Основное ограничение формы `crosstab` с одним параметром состоит в том, что она воспринимает все значения в группе одинаково и вставляет очередное значение в первый свободный столбец. Если вы хотите, чтобы столбцы значений соответствовали определённым категориям данных и некоторые группы могли содержать данные не для всех категорий, этот подход не будет работать. Форма `crosstab` с двумя параметрами решает эту задачу, принимая явный список категорий, соответствующих выходным столбцам.

В параметре *`source_sql`* передаётся SQL-оператор, выдающий исходный набор данных. Этот оператор должен выдавать строки со столбцом `row_name`, столбцом `category` и столбцом `value`. Также он может выдать один или несколько «дополнительных» столбцов. Столбец `row_name` должен быть первым, а столбцы `category` и `value` — последними двумя, именно в этом порядке. Все столбцы между `row_name` и `category` воспринимаются как «дополнительные». Ожидается, что «дополнительные» столбцы будут содержать одинаковые значения для всех строк с одним значением `row_name`.

Например, *`source_sql`* может выдать такой набор данных:

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

 row_name    extra_col   cat    value
----------+------------+-----+---------
  row1         extra1    cat1    val1
  row1         extra1    cat2    val2
  row1         extra1    cat4    val4
  row2         extra2    cat1    val5
  row2         extra2    cat2    val6
  row2         extra2    cat3    val7
  row2         extra2    cat4    val8
```

В параметре *`category_sql`* передаётся оператор SQL, выдающий набор категорий. Этот оператор должен возвращать всего один столбец. Он должен выдать минимум одну строку; в противном случае произойдёт ошибка. Кроме того, выдаваемые им значения не должны повторяться, иначе так же произойдёт ошибка. В качестве *`category_sql`* можно передать, например, такой запрос:

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

Функция `crosstab` объявлена как возвращающая тип `setof record`, так что фактические имена и типы выходных столбцов должны определяться в предложении `FROM` вызывающего оператора `SELECT`, например так:

```
SELECT * FROM crosstab('...', '...')
    AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

При этом будет получен примерно такой результат:

```
                  <==  столбцы значений   ==>
```

```
row_name   extra   cat1   cat2   cat3   cat4
---------+-------+------+------+------+------
  row1     extra1  val1   val2          val4
  row2     extra2  val5   val6   val7   val8
```

В предложении `FROM` должно определяться нужное количество выходных столбцов соответствующих типов данных. Если запрос *`source_sql`* выдаёт *`N`* столбцов, первые *`N`*-2 из них должны соответствовать первым *`N`*-2 выходным столбцам. Оставшиеся выходные столбцы должны иметь тип последнего столбца результата *`source_sql`* и их должно быть столько, сколько строк оказалось в результате запроса *`category_sql`*.

Функция `crosstab` выдаёт одну выходную строку для каждой последовательной группы входных строк с одним значением `row_name`. Выходной столбец `row_name` плюс все «дополнительные» столбцы копируются из первой строки группы. Выходные столбцы значений заполняются содержимым полей `value` из строк с соответствующими значениями `category`. Если в поле `category` оказывается значение, отсутствующее в результате запроса *`category_sql`*, содержимое поля `value` в этой строке игнорируется. Выходные столбцы, для которых соответствующая категория не представлена ни в одной из входных строк группы, принимают значения NULL.

На практике в запросе *`source_sql`* всегда нужно указывать `ORDER BY 1`, чтобы все значения с одним `row_name` гарантированно выводились вместе. Порядок же категорий внутри группы не важен. Кроме того, важно, чтобы порядок значений, выдаваемых запросом *`category_sql`*, соответствовал заданному порядку выходных столбцов.

Два законченных примера:

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov  | Dec
------+------+------+-----+-----+-----+-----+-----+-----+-----+-----+------+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |     | 1500 | 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |     |      |
(2 rows)
```

```
CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
```

```
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
       rowid text,
       rowdt timestamp,
       temperature int4,
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |      test_startdate
  | volts
-------+--------------------------+-------------+-------------
+--------------------------+--------
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |
  | 2.6987
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01 00:00:00
 2003 | 3.1234
(2 rows)
```

Вы можете создать предопределённые функции, чтобы не выписывать имена и типы результирующих столбцов в каждом запросе. Примеры приведены в предыдущем разделе. Нижележащая функция C для этой формы `crosstab` называется `crosstab_hash`.

F.63.1.5. connectby

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

Функция `connectby` выдаёт отображение данных, содержащихся в таблице, в иерархическом виде. Таблица должна содержать поле ключа, однозначно идентифицирующее строки, и поле ключа родителя, ссылающееся на родителя строки (если он есть). Функция `connectby` может вывести вложенное дерево, начиная с любой строки.

Параметры описаны в Таблице F.42.

Таблица F.42. Параметры connectby

Параметр	Описание
relname	Имя исходного отношения
keyid_fld	Имя поля ключа
parent_keyid_fld	Имя поля, содержащего ключ родителя
orderby_fld	Имя поля, по которому сортируются потомки (необязательно)
start_with	Значение ключа отправной строки

Параметр	Описание
max_depth	Максимальная глубина, на которую можно погрузиться, либо ноль для неограниченного погружения
branch_delim	Строка, разделяющая ключи в выводе ветви (необязательно)

Поля ключа и ключа родителя могут быть любого типа, но должны иметь общий тип. Заметьте, что значение *start_with* должно задаваться текстовой строкой, вне зависимости от типа поля ключа.

Функция `connectby` объявлена как возвращающая `setof record`, так что фактические имена и типы выходных столбцов должны определяться в предложении `FROM` вызывающего оператора `SELECT`, например так:

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0,
 '~')
    AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

Первые два выходных столбца используются для вывода ключа текущей строки и ключа её родителя; их тип должен соответствовать типу поля ключа. Третий выходной столбец задаёт глубину в дереве и должен иметь тип `integer`. Если передаётся параметр *branch_delim*, в следующем столбце выводятся ветви, и этот столбец должен иметь тип `text`. Наконец, если передаётся параметр *orderby_fld*, в последнем столбце выводятся последовательные числа, и он должен иметь тип `integer`.

В столбце «branch» показывается путь по ключам, приведший к текущей строке. Ключи разделяются заданной строкой *branch_delim*. Если выводить ветви не требуется, опустите параметр *branch_delim* и столбец branch в списке выходных столбцов.

Если порядок потомков одного родителя имеет значение, добавьте параметр *orderby_fld*, указывающий поле для упорядочивания потомков. Это поле может иметь любой тип, допускающий сортировку. Список выходных столбцов должен включать последним столбцом целочисленный столбец с последовательными значениями, если и только если передаётся параметр *orderby_fld*.

Параметры, представляющие имена таблицы и полей, копируются как есть в SQL-запросы, которые `connectby` генерирует внутри. Таким образом, их нужно заключить в двойные кавычки, если они содержат буквы в разном регистре или специальные символы. Также может понадобиться дополнить имя таблицы схемой.

С большими таблицами производительность будет неудовлетворительной, если не создать индекс по полю с ключом родителя.

Важно, чтобы строка *branch_delim* не фигурировала в значениях ключа, иначе `connectby` может некорректно сообщить об ошибке бесконечной вложенности. Заметьте, что если параметр *branch_delim* не задаётся, для выявления зацикленности применяется символ `~`.

Пример:

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);
```

```
-- с ветвями без orderby_fld (порядок результатов не гарантируется)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+---------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)

-- без ветвей и без orderby_fld (порядок результатов не гарантируется)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)

-- с ветвями и с orderby_fld (заметьте, что row5 идёт перед row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0,
 '~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos int);
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)

-- без ветвей, с orderby_fld (заметьте, что row5 идёт перед row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.63.2. Автор

Джо Конвей (Joe Conway)

F.64. tcn — триггерная функция, сообщающая приёмникам уведомлений об изменениях в содержимом таблицы

Модуль `tcn` предлагает триггерную функцию, уведомляющую приёмники уведомлений об изменениях в любой таблице, к которой привязан триггер. Она должна использоваться в качестве триггера `AFTER` вида `FOR EACH ROW`.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

Этой функции в операторе `CREATE TRIGGER` передаётся только один параметр, и он не является обязательным. Если этот параметр присутствует, он задаёт имя канала для уведомлений. В случае его отсутствия именем канала будет `tcn`.

Сообщение уведомления включает имя таблицы, букву, обозначающую тип выполняемой операции и пары имя столбца/значение для столбца первичного ключа. Каждая часть сообщения отделяется от следующей запятой. Для упрощения разбора сообщения регулярными выражениями имена таблицы и столбцов всегда заключаются в двойные кавычки, а значения данных — в апострофы. Внутренние кавычки и апострофы дублируются.

Далее приведена краткая демонстрация использования расширения.

```
test=# create table tcndata
test-#   (
test(#     a int not null,
test(#     b date not null,
test(#     c text,
test(#     primary key (a, b)
test(#   );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test-#   after insert or update or delete on tcndata
test-#   for each row execute function triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test-#                            (1, date '2012-12-23', 'another'),
test-#                            (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-22'"
 received from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-23'"
 received from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='2',"b"='2012-12-23'"
 received from server process with PID 22770.
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-22'"
 received from server process with PID 22770.
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-23'"
 received from server process with PID 22770.
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload ""tcndata",D,"a"='1',"b"='2012-12-22'"
 received from server process with PID 22770.
```

F.65. test_decoding — пример модуля на базе SQL для логического декодирования WAL

Модуль `test_decoding` представляет пример модуля вывода логического декодирования. Он не делает ничего особенно полезного, но может послужить отправной точкой для разработки собственного модуля вывода.

Модуль `test_decoding` получает WAL через механизм логического декодирования и переводит его в текстовое представление выполняемых операций.

Типичный вывод этого модуля, работающего через интерфейс логического декодирования SQL, может выглядеть так:

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL, 'include-xids', '0');
   lsn     | xid |                       data
-----------+-----+--------------------------------------------------
 0/16D30F8 | 691 | BEGIN
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:2 data[text]:'arg'
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:3 data[text]:'demo'
 0/16D32A0 | 691 | COMMIT
 0/16D32D8 | 692 | BEGIN
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:2
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:3
 0/16D3398 | 692 | COMMIT
(8 rows)
```

Также можно получить информацию об изменениях текущей транзакции — типичный вывод может выглядеть так:

```
postgres[33712]=#* SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL,
 'stream-changes', '1');
    lsn    | xid |                       data
-----------+-----+--------------------------------------------------
 0/16B21F8 | 503 | opening a streamed block for transaction TXN 503
 0/16B21F8 | 503 | streaming change for TXN 503
 0/16B2300 | 503 | streaming change for TXN 503
 0/16B2408 | 503 | streaming change for TXN 503
 0/16BEBA0 | 503 | closing a streamed block for transaction TXN 503
 0/16B21F8 | 503 | opening a streamed block for transaction TXN 503
 0/16BECA8 | 503 | streaming change for TXN 503
 0/16BEDB0 | 503 | streaming change for TXN 503
 0/16BEEB8 | 503 | streaming change for TXN 503
 0/16BEBA0 | 503 | closing a streamed block for transaction TXN 503
(10 rows)
```

F.66. tsm_system_rows — метод извлечения выборки `SYSTEM_ROWS` для `TABLESAMPLE`

Модуль `tsm_system_rows` предоставляет метод извлечения выборки `SYSTEM_ROWS`, который можно использовать в предложении `TABLESAMPLE` команды `SELECT`.

Этот метод извлечения выборки принимает один целочисленный аргумент, задающий максимальное число выбираемых строк. Результирующая выборка будет содержать в точности столько строк, если только в таблице не оказывается меньше заданного числа строк (в этом случае выдаётся вся таблица).

Как и встроенный метод извлечения выборки `SYSTEM`, `SYSTEM_ROWS` производит выборку на уровне блоков, так что выборка будет не полностью случайной, а может подвергаться эффектам кластеризации, особенно когда запрашивается небольшое число строк.

`SYSTEM_ROWS` не поддерживает предложение `REPEATABLE`.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.66.1. Примеры

Пример получения выборки из таблицы с применением метода `SYSTEM_ROWS`. Сначала нужно установить расширение:

```
CREATE EXTENSION tsm_system_rows;
```

Затем вы можете использовать его в команде `SELECT`, например так:

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_ROWS(100);
```

Эта команда выдаст выборку из 100 строк из таблицы `my_table` (а если в таблице не окажется 100 видимых строк, будут возвращены все строки).

F.67. tsm_system_time — метод извлечения выборки `SYSTEM_TIME` для `TABLESAMPLE`

Модуль `tsm_system_time` предоставляет метод извлечения выборки `SYSTEM_TIME`, который можно использовать в предложении `TABLESAMPLE` команды `SELECT`.

Этот метод извлечения выборки принимает в единственном аргументе число с плавающей точкой, задающее максимальное время (в миллисекундах), которое отводится на чтение таблицы. Это даёт возможность непосредственно управлять длительностью выполнения запроса, ценой того, что размер выборки оказывается трудно предсказуемым. Результирующая выборка будет содержать столько строк, сколько удастся прочитать за отведённое время, если только быстрее не будет прочитана вся таблица.

Как и встроенный метод извлечения выборки `SYSTEM`, `SYSTEM_TIME` производит выборку на уровне блоков, так что выборка будет не полностью случайной, а может подвергаться эффектам кластеризации, особенно когда выбирается небольшое число строк.

`SYSTEM_TIME` не поддерживает предложение `REPEATABLE`.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.67.1. Примеры

Пример получения выборки из таблицы с применением метода `SYSTEM_TIME`. Сначала нужно установить расширение:

```
CREATE EXTENSION tsm_system_time;
```

Затем вы можете использовать его в команде `SELECT`, например так:

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_TIME(1000);
```

Эта команда выдаст настолько большую выборку из `my_table`, насколько много строк она успеет прочитать за 1 секунду (1000 миллисекунд). Разумеется, если за 1 секунду удастся прочитать всю таблицу, будут возвращены все её строки.

F.68. unaccent — словарь текстового поиска, который убирает диакритические знаки

Модуль `unaccent` представляет словарь текстового поиска, который убирает надстрочные (диакритические) знаки из лексем. Это фильтрующий словарь, что значит, что выводимые им данные всегда передаются следующему словарю (если он есть), в отличие от нормальных словарей. Применяя его, можно выполнить полнотекстовый поиск без учёта ударений (диакритики).

Текущую реализацию `unaccent` нельзя использовать в качестве нормализующего словаря для словаря `thesaurus`.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.68.1. Конфигурирование

Словарь `unaccent` принимает следующие параметры:

- Параметр `RULES` задаёт базовое имя файла со списком правил преобразования. Этот файл должен находиться в каталоге `$SHAREDIR/tsearch_data/` (где под `$SHAREDIR` понимается каталог с общими данными инсталляции Postgres Pro). Имя файла должно заканчиваться расширением `.rules` (которое не нужно указывать в параметре `RULES`).

Файл правил имеет следующий формат:

- Каждая строка представляет одно правило перевода, состоящее из символа с диакритикой, за которым следует символ без диакритики. Первый символ переводится во второй. Например,

  ```
  À        A
  Á        A
  Â        A
  Ã        A
  Ä        A
  Å        A
  Æ        AE
  ```

  Эти два символа должны разделяться пробельным символом, а любые начальные или конечные пробельные символы в этой строке игнорируются.

- В качестве альтернативного варианта, если в строке задан всего один символ, вхождения этого символа будут удаляться; это полезно для языков, в которых диакритика представляется отдельными символами.

- На самом деле роль «символа» может играть любая строка, не содержащая пробельные символы, так что словари `unaccent` могут быть полезны и для другого рода замены подстрок, а не только для удаления диакритик.

- Для некоторых символов, например числовых, в правиле перевода требуются пробелы. В этом случае переведённые знаки можно обрамлять двойными кавычками. Двойную кавычку при добавлении к переведённому символу нужно экранировать второй двойной кавычкой. Например:

  ```
  ¼       " 1/4"
  ½       " 1/2"
  ¾       " 3/4"
  “        """"
  ”        """"
  ```

- Как и другие файлы конфигурации текстового поиска в Postgres Pro, файл правил должен иметь кодировку UTF-8. При загрузке данные из него будут автоматически преобразованы в кодировку текущей базы данных. Все строки в нём, содержащие непереводимые символы, просто игнорируются, так что файлы правил могут содержать правила, неприменимые в текущей кодировке.

Более полный набор правил, непосредственно пригодный для большинства европейских языков, можно найти в файле `unaccent.rules`, который помещается в `$SHAREDIR/tsearch_data/`, когда устанавливается модуль `unaccent`. Этот файл правил переводит буквы с диакритикой в те же буквы без диакритики, а также разворачивает лигатуры в равнозначные последовательности простых символов (например, Æ в AE).

F.68.2. Использование

При установке расширения `unaccent` в базе создаётся шаблон текстового поиска `unaccent` и словарь `unaccent` на его основе. Для словаря `unaccent` по умолчанию определяется параметр `RULES='unaccent'`, благодаря чему его можно сразу использовать со стандартным файлом `unaccent.rules`. При желании вы можете изменить этот параметр, например, так

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

или создать новые словари на основе этого шаблона.

Протестировать этот словарь можно так:

```
mydb=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
(1 row)
```

Следующий пример показывает, как вставить словарь `unaccent` в конфигурацию текстового поиска:

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
------------------------
 <b>Hôtel</b> de la Mer
(1 row)
```

F.68.3. Функции

Функция `unaccent()` удаляет надстрочные (диакритические) знаки из заданной строки. По сути она представляет собой обёртку вокруг словарей в стиле `unaccent`, но может вызываться и вне обычного контекста текстового поиска.

```
unaccent([словарь regdictionary,] строка text) returns text
```

Если аргумент *`словарь`* опущен, будет использоваться словарь с именем `unaccent`, находящийся в той же схеме, что и сама функция `unaccent()`.

Пример:

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

F.69. uuid-ossp — генератор UUID

Модуль `uuid-ossp` предоставляет функции для генерирования универсальных уникальных идентификаторов (UUID) по одному из нескольких стандартных алгоритмов. В нём также есть функции, выдающие специальные UUID-константы. Этот модуль необходим только в случае особых требований, которым не удовлетворяет функциональность в ядре Postgres Pro. Встроенные в ядро способы генерирования UUID описаны в Разделе 9.14.

Данный модуль считается «доверенным», то есть его могут устанавливать обычные пользователи, имеющие право `CREATE` в текущей базе данных.

F.69.1. Функции `uuid-ossp`

В Таблице F.43 показаны функции, предназначенные для генерации UUID. Четыре алгоритма для генерации UUID, обозначаемые номерами версий 1, 3, 4 и 5, описаны в стандартах ITU-T Rec. X.667, ISO/IEC 9834-8:2005 и *RFC 4122*. (Алгоритма версии 2 нет.) Каждый из этих алгоритмов предназначен для различных сфер применения.

Таблица F.43. Функции для генерирования UUID

Функция Описание
`uuid_generate_v1` () → `uuid` Генерирует UUID версии 1. Такой UUID включает в себя MAC-адрес компьютера и текущее время. Заметьте, что UUID такого типа раскрывают «личность» компьютера, создавшего идентификатор, и время этой операции, что может быть неприемлемым для определённых приложений, где важна конфиденциальность.
`uuid_generate_v1mc` () → `uuid` Генерирует UUID версии 1, но вместо реального MAC-адреса компьютера используется случайный групповой MAC-адрес.

`uuid_generate_v3` (*namespace* `uuid`, *name* `text`) → `uuid`
Генерирует UUID версии 3 для заданного пространства имён UUID и указанного имени. Пространство имён должно задаваться одной из специальных констант, которые выдаются функциями `uuid_ns_*()`, перечисленными в Таблице F.44. (Хотя теоретически это может быть любой UUID.) Имя задаёт идентификатор в выбранном пространстве имён.
Например:
`SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org');`
Из параметра name будет получен MD5-хеш, так что из сгенерированного UUID нельзя будет восстановить имя. В генерируемых таким алгоритмом UUID нет элемента случайности или зависимости от окружения, так что они могут быть воспроизведены.

`uuid_generate_v4` () → `uuid` Генерирует UUID версии 4, который полностью определяется случайными числами.
`uuid_generate_v5` (*namespace* `uuid`, *name* `text`) → `uuid` Генерирует UUID версии 5, который похож на версию 3, но хеш рассчитывается по алгоритму SHA-1. Версия 5 предпочтительнее версии 3, так как SHA-1 считается более безопасным, чем MD5.

Таблица F.44. Функции, возвращающие UUID-константы

Функция Описание
`uuid_nil` () → `uuid` Выдаёт «нулевой» UUID, который не считается действительным UUID.

Функция **Описание**
`uuid_ns_dns` () → `uuid` Выдаёт константу, обозначающую пространство имён DNS для UUID.
`uuid_ns_url` () → `uuid` Выдаёт константу, обозначающую пространство имён URL для UUID.
`uuid_ns_oid` () → `uuid` Выдаёт константу, обозначающую пространство имён идентификаторов объектов ISO (OID, ISO Object Identifier) для UUID. (Здесь имеются в виду идентификаторы объектов ASN.1, которые никак не связаны с OID, применяемыми в Postgres Pro.)
`uuid_ns_x500` () → `uuid` Выдаёт константу, обозначающую пространство имён с уникальными именами X.500 для UUID.

F.69.2. Сборка `uuid-ossp`

В прошлом этот модуль зависел от библиотеки OSSP UUID, что отразилось в его имени. Хотя библиотеку OSSP UUID всё ещё можно найти по адресу *http://www.ossp.org/pkg/lib/uuid/*, она плохо поддерживается и её становится всё сложнее портировать на новые платформы. Поэтому модуль `uuid-ossp` теперь на некоторых платформах можно собирать без библиотеки OSSP. Во FreeBSD и некоторых других ОС на базе BSD подходящие функции формирования UUID включены в системную библиотеку `libc`. В Linux, macOS и некоторых других платформах подходящие функции предоставляются библиотекой `libuuid`, которая изначально пришла из проекта `e2fsprogs` (хотя в современных дистрибутивах Linux она является частью пакета `util-linux-ng`). Вызывая `configure`, передайте ключ `--with-uuid=bsd`, чтобы использовать функции BSD, либо `--with-uuid=e2fs`, чтобы использовать `libuuid` из `e2fsprogs`, либо ключ `--with-uuid=ossp`, чтобы использовать библиотеку OSSP UUID. В конкретной системе может быть установлено сразу несколько библиотек, поэтому `configure` не выбирает библиотеку автоматически.

F.69.3. Автор

Питер Эйзентраут `<peter_e@gmx.net>`

F.70. xml2 — функции для выполнения запросов XPath и преобразований XSLT

Модуль `xml2` предоставляет функции для выполнения запросов XPath и преобразований XSLT.

F.70.1. Уведомление об актуальности

Начиная с PostgreSQL 8.3, функциональность, связанная с XML, основана на стандарте SQL/XML и включена в ядро сервера. Эта функциональность охватывает проверку синтаксиса XML и запросы XPath, что в частности делает и этот модуль, но он имеет абсолютно несовместимый API. Этот модуль планируется удалить в будущей версии Postgres Pro в пользу нового стандартного API, так что мы рекомендуем вам попробовать перевести свои приложения на новый API. Если вы обнаружите, что какая-то функциональность этого модуля не представлена новым API в подходящей форме, пожалуйста, напишите о вашем затруднении в `<pgsql-hackers@lists.postgresql.org>`, чтобы этот недостаток был рассмотрен и, возможно, устранён.

F.70.2. Описание функций

Функции, предоставляемые этим модулем, перечислены в Таблице F.45. Эти функции позволяют выполнять простой разбор XML и запросы XPath.

Таблица F.45. Функции `xml2`

Функция **Описание**
`xml_valid` (*`document`* `text`) → `boolean` Разбирает текст переданного документа и возвращает true, если это правильно сформированный XML. (Замечание: это псевдоним стандартной функции Postgres Pro `xml_is_well_formed()`. Имя `xml_valid()` технически некорректно, так как понятия правильности формата (well-formed) и допустимости (valid) в XML различаются.)
`xpath_string` (*`document`* `text`, *`query`* `text`) → `text` Обрабатывает запрос XPath для переданного документа и приводит результат к типу `text`.
`xpath_number` (*`document`* `text`, *`query`* `text`) → `real` Обрабатывает запрос XPath для переданного документа и приводит результат к типу `real`.
`xpath_bool` (*`document`* `text`, *`query`* `text`) → `boolean` Обрабатывает запрос XPath для переданного документа и приводит результат к типу `boolean`.

`xpath_nodeset` (*`document`* `text`, *`query`* `text`, *`toptag`* `text`, *`itemtag`* `text`) → `text`
Обрабатывает запрос для документа и помещает результат внутрь XML-тегов. Если результат содержит несколько значений, она выдаст:
`<toptag>`
`<itemtag>Значение 1, которое может быть фрагментом XML</itemtag>`
`<itemtag>Значение 2....</itemtag>`
`</toptag>`
Если *`toptag`* или *`toptag`* — пустая строка, соответствующий тег опускается.

`xpath_nodeset` (*`document`* `text`, *`query`* `text`, *`itemtag`* `text`) → `text` Подобна `xpath_nodeset(document, query, toptag, itemtag)`, но выводит результат без *`toptag`*.
`xpath_nodeset` (*`document`* `text`, *`query`* `text`) → `text` Подобна `xpath_nodeset(document, query, toptag, itemtag)`, но выводит результат без обоих тегов.

Функция Описание
`xpath_list ( document text, query text, separator text ) → text` Обрабатывает запрос XPath для документа и возвращает несколько значений, вставляя между ними заданный разделитель (*separator*), например: `Значение 1,Значение 2, Значение 3`, если *separator* содержит знак `,` .
`xpath_list ( document text, query text ) → text` Это обёртка предыдущей функции, устанавливающая в качестве разделителя знак `,` .

F.70.3. xpath_table

```
xpath_table(text key, text document, text relation, text xpaths, text criteria) returns
 setof record
```

Табличная функция `xpath_table` выполняет набор запросов XPath для каждого из набора документов и возвращает результаты в виде таблицы. В первом столбце результата возвращается первичный ключ из таблицы документов, так что результат оказывается готовым к применению в соединениях. Параметры функции описаны в Таблице F.46.

Таблица F.46. Параметры `xpath_table`

Параметр	Описание
key	имя «ключевого» поля — содержимое этого поля просто окажется в первом столбце выходной таблицы, то есть оно указывает на запись, из которой была получена определённая выходная строка (см. замечание о нескольких значениях ниже)
document	имя поля, содержащего XML-документ
relation	имя таблицы (или представления), содержащей документы
xpaths	одно или несколько выражений XPath, разделённых символом `|`
criteria	содержимое предложения WHERE. Оно не может быть пустым, так что если вам нужно обработать все строки в отношении, напишите `true` или `1=1`

Эти параметры (за исключением строк XPath) просто подставляются в обычный оператор SQL SELECT, так что у вас есть определённая гибкость — оператор выглядит так:

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

поэтому в этих параметрах можно передать *всё*, что будет корректно воспринято в этих позициях. Этот SELECT должен возвращать ровно два столбца (что и будет иметь место, если только вы не перечислите несколько полей в параметрах key или document). Будьте осторожны — при таком примитивном подходе обязательно нужно проверять все значения, получаемые от пользователя, во избежание атак с инъекцией SQL.

Эта функция предназначена для использования в выражении `FROM`, с предложением `AS`, задающим выходные столбцы; например:

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title text);
```

Предложение `AS` определяет имена и типы столбцов в выходной таблице. Первым определяется «ключевое» поле, а за ним поля, соответствующие запросам XPath. Если запросов XPath больше, чем столбцов в результате, лишние запросы будут игнорироваться. Если же результирующих столбцов больше, чем запросов XPath, дополнительные столбцы принимают значение NULL.

Заметьте, что в этом примере столбец результата `page_count` определён как целочисленный. Данная функция внутри имеет дело со строковыми значениями, так что, когда вы указываете, что в результате нужно получить целое число, она берёт текстовое представление результата XPath и, применяя функции ввода Postgres Pro, преобразует её в целое число (или в тот тип, который указан в предложении `AS`). Если она не сможет сделать это, произойдёт ошибка — например, если результат пустой — так что если вы допускаете возможность таких проблем с данными, возможно, будет лучше просто оставить для столбца тип `text`.

Вызывающий оператор `SELECT` не обязательно должен быть простым `SELECT *` — он может обращаться к выходным столбцам по именам и соединять их с другими таблицами. Эта функция формирует виртуальную таблицу, с которой вы можете выполнять любые операции, какие пожелаете (например, агрегировать, соединять, сортировать данные и т. д.). Поэтому возможен и такой запрос:

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,''/article/@date'') > ''2003-03-20'' ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

в качестве более сложного примера. Разумеется, для удобства вы можете завернуть весь этот запрос в представление.

F.70.3.1. Результаты с набором значений

Функция `xpath_table` рассчитана на то, что результатом каждого запроса XPath может быть набор данных, так что количество возвращённых этой функцией строк может не совпадать с количеством входных документов. В первой строке возвращается первый результат каждого запроса, во второй — второй результат и т. д. Если один из запросов возвращает меньше значений, чем другие, вместо недостающих значений будет возвращаться NULL.

В некоторых случаях пользователь знает, что некоторый запрос XPath будет возвращать только один результат (возможно, уникальный идентификатор документа) — если он используется рядом с запросом XPath, возвращающим несколько результатов, результат с одним значением будет выведен только в первой выходной строке. Чтобы исправить это, можно воспользоваться полем ключа и соединить результат с более простым запросом XPath. Например:

```
CREATE TABLE test (
    id int PRIMARY KEY,
    xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',
              '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int, val3 int)
```

```
WHERE id = 1 ORDER BY doc_num, line_num

 id | doc_num | line_num | val1 | val2 | val3
----+---------+----------+------+------+------
  1 | C1      | L1       |    1 |    2 |    3
  1 |         | L2       |   11 |   22 |   33
```

Чтобы получить `doc_num` в каждой строке, можно вызывать `xpath_table` дважды и соединить результаты:

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
              '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
    AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
    AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;

 id | line_num | val1 | val2 | val3 | doc_num
----+----------+------+------+------+---------
  1 | L1       |    1 |    2 |    3 | C1
  1 | L2       |   11 |   22 |   33 | C1
(2 rows)
```

F.70.4. Функции XSLT

Если установлена libxslt, доступны следующие функции:

F.70.4.1. `xslt_process`

`xslt_process(text document, text stylesheet, text paramlist) returns text`

Эта функция применяет стиль XSL к документу и возвращает результат преобразования. В `paramlist` передаётся список присваиваний значений параметрам, которые будут использоваться в преобразовании, в форме `a=1,b=2`. Учтите, что разбор параметров выполнен очень просто: значения параметров не могут содержать запятые!

Есть также версия `xslt_process` с двумя аргументами, которая не передаёт никакие параметры преобразованию.

F.70.5. Автор

Джон Грей `<jgray@azuli.co.uk>`

Разработку этого модуля спонсировала компания Torchbox Ltd. (www.torchbox.com). Этот модуль выпускается под той же лицензией BSD, что и Postgres Pro.

Приложение G. Модули и расширения Postgres Pro, поставляемые в виде отдельных пакетов

В этом приложении содержится информация о модулях и расширениях Postgres Pro, которые доступны в Postgres Pro Standard как отдельные пакеты. Они перечислены в Подразделе 16.1.4.1, а в документации каждого модуля объясняется, какой пакет выбрать для его установки. В Приложении F описаны прочие модули и расширения, включённые в дистрибутив Postgres Pro Standard. Расширение pg_portal_modify также поставляется как отдельный пакет (за подробностями обратитесь к Подразделу 16.1.4.1).

G.1. pgpro_pwr — отчёты о нагрузке

Модуль pgpro_pwr предназначен для выявления наиболее ресурсоёмких операций в базе данных. (Корень pwr, произносится как «пауэр» (power), — это сокращение от Postgres Pro Workload Reporting, Отчётность по нагрузке Postgres Pro.) Данный модуль основывается на представлениях Сборщика статистики и расширении pgpro_stats или pg_stat_statements.

Примечание

Хотя pgpro_pwr может работать с расширением pg_stat_statements, по возможности рекомендуется использовать расширение pgpro_stats, так как оно выдаёт планы операторов, информацию о событиях ожидания и статистику распределения нагрузки для баз данных, ролей, клиентских узлов и приложений.

Ниже предполагается, что используется pgpro_stats, если иное не отмечено явно.

Если у вас нет возможности использовать pgpro_stats в нужной базе, но имеется расширение *pg_stat_kcache*, pgpro_pwr может обрабатывать выдаваемую pg_stat_kcache информацию об использовании командами ресурсов процессора и о нагрузке на уровне файловой системы (*rusage*).

pgpro_pwr получает сводную статистику ожидания от расширения pg_wait_sampling. Когда используется pg_wait_sampling, pgpro_pwr сбрасывает профиль выборки ожидания для каждой выборки.

Суть pgpro_pwr заключается в обработке накапливаемых статистических выборок. В каждой выборке содержится статистическая информация о наиболее активных объектах и запросах, добавленная с момента получения предыдущей выборки, или, говоря кратко, *после предыдущей выборки*. Впоследствии эта информация используется при построении отчётов.

pgpro_pwr предоставляет функции для получения выборок. Аккумулируемые регулярные выборки позволяют строить отчёты о нагрузке базы данных за прошедшее время.

pgpro_pwr позволяет создавать выборки явным образом во время массовой обработки данных, тестирования нагрузки и т. д.

При каждом получении выборки вызывается функция `pgpro_stats_statements_reset()` (она описана в pgpro_stats), чтобы статистика выполнения операторов не потерялась, когда количество операторов превысит `pgpro_stats.max` (см. Подраздел G.2.7.1). Кроме этого, в отчёте будет содержаться раздел, в котором можно узнать, не превышает ли количество операторов 90% от значения `pgpro_stats.max`.

Расширение pgpro_pwr, установленное на одном сервере Postgres Pro, может собирать статистику и с других серверов. Таким образом, на ведущем сервере можно собирать статистику нагрузки также с серверов горячего резерва. Чтобы это реализовать, необходимо указать имена всех серверов и строки подключения, а также обеспечить возможность подключения pgpro_pwr ко всем серверам.

G.1.1. Архитектура pgpro_pwr

Данное расширение состоит из следующих частей:

- *Репозиторий истории* — хранилище собранных данных. В нём находится несколько таблиц расширения.

> **Примечание**
>
> Помимо прочего, в таблицах pgpro_pwr хранится текст запросов в открытом виде, который может содержать конфиденциальную информацию. Поэтому в целях безопасности ограничьте доступ к репозиторию должным образом.

- *Механизм управления выборками* включает функции, позволяющие делать выборки и поддерживать актуальное состояние репозитория, удаляя устаревшие данные.
- *Механизм отчётов* включает функции построения отчётов по данным, находящемся в репозитории истории.
- *Административные функции*, предназначенные для создания серверов и выборочных линий, а также для управления ими.

G.1.2. Предварительные требования

Следующие требования приведены для варианта установки pgpro_pwr в *целевом* кластере, то есть в кластере, нагрузка которого будет анализироваться в первую очередь, хотя это расширение может собирать данные производительности и с других кластеров.

G.1.2.1. Для базы данных pgpro_pwr

Расширение pgpro_pwr зависит от языка PL/pgSQL и расширения dblink.

G.1.2.2. Для целевого сервера

Целевой сервер должен разрешать подключения ко всем базам данных с сервера, на котором работает pgpro_pwr. Вам надо будет задать строку подключения для этого сервера с указанием определённой базы данных. Эта база данных имеет особое значение, так как pgpro_pwr будет обращаться к установленному в ней расширению pgpro_stats или pg_stat_statements. Однако заметьте, что pgpro_pwr будет подключаться и ко всем остальным базам на этом сервере.

Для получения более полной статистики можно дополнительно:

- Установить и настроить расширение pgpro_stats в вышеупомянутой базе данных, если вы хотите видеть в отчётах статистику по операторам. Полнота и точность собираемой статистики может зависеть от следующих параметров:
  - `pgpro_stats.max`

    При небольшом значении этого параметра некоторая статистика может теряться между выборками. Если объём попадающей в выборку статистики близок к значению `pgpro_stats.max`, вы увидите предупреждение об этом в отчёте.
  - `pgpro_stats.track`

    Оптимальным является значение по умолчанию, '`top`' (заметьте, что значение '`all`' повлияет на точность полей `%Total` (% всего) в разделах отчёта, посвящённым SQL-операторам).
- Задать следующие параметры Сборщика статистики Postgres Pro:

```
track_activities = on
track_counts = on
track_io_timing = on
track_wal_io_timing = on    # Начиная с PostgreSQL 14
track_functions = all/pl
```

G.1.3. Установка и подготовка

pgpro_pwr поставляется вместе с Postgres Pro Standard в виде отдельного пакета `pgpro-pwr-std-17` (подробные инструкции по установке приведены в Главе 16).

> **Примечание**
>
> Расширение pgpro_pwr создаёт множество объектов в базе данных, поэтому рекомендуется устанавливать его в отдельную схему.

Хотя pgpro_pwr вполне можно использовать с правами суперпользователя, наличие таких прав не является обязательным. Поэтому вы можете выбрать одну из следующих процедур установки в зависимости от ваших требований безопасности и конфигурации или разработать собственную:

G.1.3.1. Упрощённая установка

Эта процедура предназначена для случаев, когда pgpro_pwr устанавливается непосредственно на целевой сервер и будет отслеживать его нагрузку от имени суперпользователя.

Создайте схему для установки pgpro_pwr, а затем создайте расширение:

```
CREATE SCHEMA profile;
CREATE EXTENSION pgpro_pwr SCHEMA profile;
```

G.1.3.2. Развёрнутая установка

Эта процедура охватывает случаи, когда планируется использовать pgpro_pwr для отслеживания нагрузки на одном или нескольких серверах с соблюдением принципа наименьших привилегий.

G.1.3.2.1. В базе данных целевого сервера

Создайте на целевом сервере пользователя для pgpro_pwr:

```
CREATE USER pwr_collector PASSWORD 'collector_pwd';
```

Убедитесь в том, что пользователь имеет возможность подключаться к любой базе данных кластера (по умолчанию это так) и что `pg_hba.conf` разрешает такие подключения с сервера, где размещён pgpro_pwr. Также включите этого пользователя в роль `pwr_collector` и дайте ему право `EXECUTE` для следующих функций:

```
GRANT pg_read_all_stats TO pwr_collector;
GRANT EXECUTE ON FUNCTION pgpro_stats_statements_reset TO pwr_collector;
GRANT EXECUTE ON FUNCTION pgpro_stats_totals_reset(text,bigint) TO pwr_collector;
```

Также следует проверить наличие права `SELECT` для представления `pgpro_stats_archiver`:

```
GRANT SELECT ON pgpro_stats_archiver TO pwr_collector;
```

G.1.3.2.2. В базе данных pgpro_pwr

Создайте непривилегированного пользователя:

```
CREATE USER pwr_user;
```

Этот пользователь будет владельцем схемы расширения, и от его имени будут собираться статистические выборки.

Создайте схему для установки pgpro_pwr:

```
CREATE SCHEMA profile AUTHORIZATION pwr_user;
```

Дайте пользователю право `USAGE` для схемы, где располагается расширение dblink:

```
GRANT USAGE ON SCHEMA public TO pwr_user;
```

Создайте расширение от имени `pwr_user`:

```
\c - pwr_user
CREATE EXTENSION pgpro_pwr SCHEMA profile;
```

Определите параметры подключения к целевому серверу для pgpro_pwr. Например:

```
SELECT profile.create_server('target_server_name','host=192.168.1.100 dbname=postgres
 port=5432');
```

Указанная строка подключения будет использоваться в вызове `dblink_connect()` при выполнении функции `take_sample()`.

Примечание

Строки подключения хранятся в таблице pgpro_pwr в открытом виде, поэтому следует ограничивать доступ к таблицам расширения pgpro_pwr.

G.1.3.3. Настройка ролей pgpro_pwr

В работе pgpro_pwr можно выделить до трёх ролей:

- Роль *владелец pgpro_pwr* является владельцем расширения pgpro_pwr.
- *Роль сбора статистики* используется pgpro_pwr для подключения к базам данных и сбора статистики.
- *Роль создания отчётов* используется для создания отчётов.

Если все действия с pgpro_pwr выполняются от имени суперпользователя `postgres`, можно пропустить большую часть настройки, описанной ниже.

G.1.3.3.1. Владелец pgpro_pwr

Эту роль можно использовать для выполнения всех действий, связанных с pgpro_pwr. Она будет иметь доступ к строкам подключения к серверу, которые могут содержать пароли. Эту роль следует использовать для вызова функции `take_sample()`. Пользователю с такой ролью необходимо расширение dblink.

Рассмотрим пример, в котором каждое расширение установлено в отдельной схеме:

```
\c postgres postgres
CREATE SCHEMA dblink;
CREATE EXTENSION dblink SCHEMA dblink;
CREATE USER pwr_usr with password 'pwr_pwd';
GRANT USAGE ON SCHEMA dblink TO pwr_usr;
CREATE SCHEMA profile AUTHORIZATION pwr_usr;
\c postgres pwr_usr
CREATE EXTENSION pgpro_pwr SCHEMA profile;
```

G.1.3.3.2. Роль сбора статистики

Эта роль должна использоваться расширением pgpro_pwr для подключения к базам данных и сбора статистики. Непривилегированные пользователи не могут открывать соединения, используя dblink, без пароля, поэтому необходимо указать пароль в строке подключения для каждого сервера. Эта роль должна иметь доступ ко всем расширениям, собирающим статистику, а также иметь возможность выполнять сброс статистики для таких расширений.

Рассмотрим пример. Если для сбора статистики используется расширение pgpro_stats, настройте роль сбора статистики следующим образом:

```
\c postgres postgres
CREATE SCHEMA pgps;
CREATE EXTENSION pgpro_stats SCHEMA pgps;
CREATE USER pwr_collector with password 'collector_pwd';
GRANT pg_read_all_stats TO pwr_collector;
GRANT USAGE ON SCHEMA pgps TO pwr_collector;
GRANT EXECUTE ON FUNCTION pgps.pgpro_stats_statements_reset TO pwr_collector;
```

Если для сбора статистики используется расширение pg_stat_statements, настройте её, как указано ниже:

```
\c postgres postgres
CREATE SCHEMA pgss;
CREATE SCHEMA pgsk;
CREATE SCHEMA pgws;
CREATE EXTENSION pg_stat_statements SCHEMA pgss;
CREATE EXTENSION pg_stat_kcache SCHEMA pgsk;
CREATE EXTENSION pg_wait_sampling SCHEMA pgws;
CREATE USER pwr_collector with password 'collector_pwd';
GRANT pg_read_all_stats TO pwr_collector;
GRANT USAGE ON SCHEMA pgss TO pwr_collector;
GRANT USAGE ON SCHEMA pgsk TO pwr_collector;
GRANT USAGE ON SCHEMA pgws TO pwr_collector;
GRANT EXECUTE ON FUNCTION pgss.pg_stat_statements_reset TO pwr_collector;
GRANT EXECUTE ON FUNCTION pgsk.pg_stat_kcache_reset TO pwr_collector;
GRANT EXECUTE ON FUNCTION pgws.pg_wait_sampling_reset_profile TO pwr_collector;
```

Теперь следует задать строку подключения, указывающую на базу данных с установленными расширениями статистики:

```
\c postgres pwr_usr
SELECT profile.set_server_connstr('local','dbname=postgres port=5432 host=localhost
user=pwr_collector password=collector_pwd');
```

В файле `pg_hba.conf` следует указать, что для пользователя `pwr_collector` требуется аутентификация по паролю.

Очевидно, что роль сбора статистики должна быть правильно настроена на всех серверах, с которых расширение pgpro_pwr собирает статистику.

Теперь можно вызывать `take_sample()`, используя роль `pwr_usr`:

```
\c postgres pwr_usr
SELECT * FROM take_sample();
```

Затем необходимо настроить планировщик задач (в нашем примере это команда `crontab` пользователя `postgres`):

```
*/30 * * * *   psql -U pwr_usr -d postgres -c 'SELECT profile.take_sample()' > /dev/
null 2>&1
```

Обратите внимание, что для хранения паролей можно использовать файл паролей Postgres Pro.

G.1.3.3.3. Роль создания отчётов

Любой пользователь может собирать отчёты pgpro_pwr. Минимальные права, необходимые для создания отчётов pgpro_pwr, предоставляются роли `public`. Однако полный отчёт с текстами запросов доступен только члену роли `pg_read_all_stats`. В любом случае роль создания отчётов не имеет доступа к строкам подключения к серверу, поэтому она не может получить пароли серверов.

G.1.3.4. Настройка параметров расширения

Вы можете определить в `postgresql.conf` следующие параметры pgpro_pwr:

`pgpro_pwr.max` (`integer`)

Количество выбираемых первых объектов (операторов, отношений и т. д.), которое будет выдаваться в каждой отсортированной по некоторому критерию таблице отчёта. Этот параметр влияет на размер выборки.

Значение по умолчанию — `20`.

`pgpro_pwr.max_sample_age` (`integer`)

Срок хранения выборки (в днях). Выборки старее `pgpro_pwr.max_sample_age` дней автоматически удаляются при очередном вызове `take_sample()`.

Значение по умолчанию — `7`.

`pgpro_pwr.max_query_length` (`integer`)

Максимальная длина выводимого запроса в отчётах. Все запросы в отчётах будут сокращены до количества символов, указанного в `pgpro_pwr.max_query_length`.

Значение по умолчанию — `20 000` символов.

`pgpro_pwr.track_sample_timings` (`boolean`)

Включает сбор подробной информации о времени выполнения внутренних процедур pgpro_pwr. Этот параметр полезен для диагностики в случае медленного выполнения функций получения выборок. Собранные показатели можно будет просмотреть в представлении `v_sample_timings`.

Значение по умолчанию — `off` (выкл.).

`pgpro_pwr.statements_reset` (`boolean`)

Управляет сбросом статистики pgpro_stats/pg_stat_statements во время получения выборки. Позволяет не сбрасывать статистику во время получения выборки благодаря использованию новых методов. При отключении этого параметра pgpro_pwr будет отслеживать вытеснения операторов, используя значения поля `calls`. Однако этот метод не полностью предотвращает потерю статистики. Расширения pg_stat_statements версии 1.11 и pgpro_stats версии 1.8 имеют возможности для учёта точного времени наблюдения за выражением, которые могут уменьшить потенциальную потерю данных. Когда этот параметр отключён, его можно временно включить в сеансе, чтобы периодически выполнять сброс pgpro_stats/pg_stat_statements.

Значение по умолчанию — `on` (вкл.).

`pgpro_pwr.relsize_collect_mode` (`text`)

Задаёт режим сбора сведений о размерах отношений. Возможные значения:

- `off` — сбор сведений о размерах отношений основан на каталоге `pg_class`. Хотя полученные таким образом данные являются приблизительными, их сбор почти не потребляет ресурсов.
- `on` — для каждой выборки собираются точные размеры отношений с помощью функции `pg_relation_size()`. Такой сбор требует блокировки таблицы и довольно ресурсоёмок.
- `schedule` — точные размеры отношений собираются в окне анализа размеров, определённом для каждого сервера.

Значение по умолчанию — `off` (выкл.).

G.1.4. Управление серверами

После установки pgpro_pwr создаёт по умолчанию активное определение сервера с именем *local*, соответствующее текущему кластеру. Активные определения серверов обрабатываются без явного указания при получении выборок (см. описание `take_sample()`). Неактивный сервер считается *исключённым*.

Строка локального сервера по умолчанию содержит только параметры `dbname` и `port`. Значения этих параметров берутся из свойств подключения, которое было установлено в момент создания расширения. Заданную ранее строку подключения сервера можно изменить, воспользовавшись функцией `set_server_connstr()`.

G.1.4.1. Функции управления серверами

Для управления серверами предназначены следующие функции pgpro_pwr:

`create_server(`*`server`*` name, `*`connstr`*` text, `*`enabled`*` boolean DEFAULT TRUE, `*`max_sample_age`*` integer DEFAULT NULL `*`description`*` text DEFAULT NULL)`

Создаёт определение сервера.

Аргументы:

- *`server`* — имя сервера, которое должно быть уникальным.
- *`connstr`* — строка подключения. Должна содержать все необходимые параметры для подключения со стороны сервера с pgpro_pwr к базе данных целевого сервера.
- *`enabled`* — установите этот параметр, чтобы включить сервер в число серверов, обрабатываемых функцией `take_sample()` без аргументов.
- *`max_sample_age`* — срок хранения выборки (в днях). Переопределяет глобальное значение `pgpro_pwr.max_sample_age` для данного сервера.
- *`description`* — описание сервера, которое будет включаться в отчёты.

Например, определение сервера можно создать так:

```
SELECT profile.create_server('omega','host=192.168.1.100 dbname=postgres
 port=5432');
```

`drop_server(`*`server`*` name)`

Удаляет сервер и все полученные с него выборки.

`set_server_description(`*`server`*` name `*`description`*` text)`

Задаёт описание для сервера.

`set_server_subsampling(`*`server`*` name, `*`subsample_enabled`*` boolean, `*`min_query_duration`*` interval, `*`min_xact_duration`*` interval, `*`min_xact_age`*` integer, `*`min_idle_xact_dur`*` interval hour to second)`

Определяет параметры получения промежуточных выборок для сервера.

Аргументы:

- *`server`* — имя сервера.
- *`subsample_enabled`* — определяет, включено ли получение промежуточных выборок для сервера, то есть должна ли функция `take_subsample()` фактически создавать промежуточные выборки.
- *`min_query_duration`* — предел длительности запроса.
- *`min_xact_duration`* — предел длительности транзакции.
- *`time_range`* — предел возраста транзакций.
- *`min_idle_xact_dur_age`* — предел простоя транзакции.

`enable_server(`*`server`*` name)`

Включает сервер в число серверов, обрабатываемых функцией `take_sample()` без аргументов.

`disable_server(`*`server`*` name)`

Исключает сервер из числа серверов, обрабатываемых функцией `take_sample()` без аргументов.

`rename_server(`*`server`*` name, `*`новое_имя`*` name)`

Переименовывает сервер.

`set_server_max_sample_age(`*`server`*` name, `*`max_sample_age`*` integer)`

Задаёт срок хранения выборки для сервера (в днях). Чтобы сбросить это значение, передайте `NULL` в параметре *`max_sample_age`*.

`set_server_db_exclude(`*`server`*` name, `*`exclude_db`*` name[])`

Исключает указанные базы данных на сервере из числа обрабатываемых. Это полезно, когда pgpro_pwr не может подключиться к некоторым базам в кластере (например, это возможно в кластерах Amazon RDS).

`set_server_connstr(`*`server`*` name, `*`server_connstr`*` text)`

Задаёт строку подключения для сервера.

`set_server_setting(`*`server`*` name, `*`setting`*` text, `*`value`*` jsonb)`

Выполняет тонкую настройку сбора статистики сервера. Настройки `collect*` управляют тем, какая статистика будет собираться, и для них параметр *`value`* принимает логические значения, по умолчанию равные `true`. Доступные настройки:

- `collect_pg_stat_statement` — собирать статистику выполнения операторов с помощью расширений pg_stat_statements и pg_stat_kcache.
- `collect_pg_wait_sampling` — собирать статистку событий ожидания с помощью расширения pg_wait_sampling.
- `collect_objects` — собирать статистику по всем объектам схемы, то есть по таблицам, индексам и функциям, из представлений `pg_stat_*`.
- `collect_relations` — собирать статистику по таблицам и индексам из представлений `pg_stat_*`.
- `collect_functions` — собирать статистику по пользовательским функциям из представления `pg_stat_user_functions`.
- `collect_vacuum_stats` — собирать расширенную статистку очистки баз данных.

`show_server_settings(`*`server`*` name)`

Возвращает параметры сбора статистки для заданного сервера.

`show_servers()`

Выдаёт список настроенных серверов.

G.1.5. Управление выборками

Выборка содержит статистическую информацию о нагрузке базы данных, полученную после предыдущей выборки

G.1.5.1. Функции обработки выборок

Для работы с выборками предназначены следующие функции pgpro_pwr:

`take_sample()`
`take_sample(`*`server`*` name [, `*`skip_sizes`*` boolean])`

Получает выборки.

Без параметров эта форма функции получает выборки со всех включённых серверов, обращаясь к ним последовательно, и выдаёт в результате таблицу со следующими столбцами:

- `server` — имя сервера.
- `result` — результат получения выборки. Может быть строкой `OK`, если выборка получена успешно, либо содержать текст с трассировкой ошибки в случае неудачи.
- `elapsed` — время, потраченное на получение выборки.

С параметром эта функция получает выборку с указанного сервера, даже если этот сервер исключён. Она полезна, когда вам нужно получать выборки с разных серверов с разной частотой. В случае успеха она выдаёт `0`.

Аргументы:

- *`server`* — имя сервера.
- *`skip_sizes`* — если этот параметр опущен или равен NULL, применяется политика анализа размеров; если он равен `false`, анализ размеров производится, а если `true` — пропускается.

`take_sample_subset([`*`sets_cnt`*` integer, `*`current_set`*` integer])`

Получает выборку с каждого сервера в подмножестве. Используйте эту функцию, чтобы получать выборки параллельным образом при большом количестве включённых серверов. Хотя PL/pgSQL не поддерживает параллельное выполнение функции, вы можете запустить эту функцию в нескольких сеансах. Она возвращает тот же тип, что и `take_sample()`. Если оба параметра опущены, функция действует аналогично `take_sample()`, то есть перебирает все включённые серверы по одному.

Аргументы:

- *`sets_cnt`* — количество подмножеств, на которое будет разделено множество всех включённых серверов.
- *`current_set`* — номер подмножества, в котором будут собираться выборки. Принимает значение от 0 до *`sets_cnt`* - 1. Для выбранного подмножества выборки собираются как обычно, с последовательным переходом от сервера к серверу.

Если обнаружится, что после предыдущей выборки произошёл сброс статистики, pgpro_pwr обработает соответствующие абсолютные значения как разницу; однако точность пострадает в любом случае.

`show_samples([`*`server`*` name,] [`*`days`*` integer])`

Выдаёт таблицу с информацией об имеющихся на сервере выборках статистики (если параметр *`server`* опущен, подразумевается `local`) за последние *`days`* дней (если этот параметр опущен, то за всё время). Эта таблица содержит следующие столбцы:

- `sample` — идентификатор выборки.
- `sample_time` — время, когда была получена выборка.
- `dbstats_reset` — `NULL` или время сброса статистики в представлении `pg_stat_database`, если она была сброшена после предыдущей выборки.
- `clustats_reset` — `NULL` или время сброса статистики в представлении `pg_stat_bgwriter`, если она была сброшена после предыдущей выборки.
- `archstats_reset` — `NULL` или время сброса статистики в представлении `pg_stat_archiver`, если она была сброшена после предыдущей выборки.

Функции получения выборок также поддерживают заданное политикой хранения желаемое состояние репозитория, удаляя устаревшие выборки и выборочные линии.

G.1.5.2. Получение выборок

Чтобы получить выборки со всех включённых серверов, вызовите функцию `take_sample()`. Обычно достаточно получать одну-две выборки в час. Для выполнения этой функции по расписанию можно воспользоваться планировщиком cron или подобным. Например, так выглядит расписание cron для получения выборок каждые 30 минут:

```
*/30 * * * *   psql -c 'SELECT profile.take_sample()' &> /dev/null
```

Однако результаты такого вызова не проверяются на ошибки. В производственной среде для целей мониторинга имеет смысл проверять строку результата, которая будет содержать `OK` для всех серверов, с которых выборки были получены успешно, либо текст ошибки, если получить выборку не удалось:

```
SELECT * FROM take_sample();
  server   |                                    result
  |   elapsed
-----------
+---------------------------------------------------------------------------
+-------------
 ok_node   | OK
  | 00:00:00.48
 fail_node | could not establish connection
 +| 00:00:00
           | SQL statement "SELECT dblink_connect('server_connection',server_connstr)"
 +|
           | PL/pgSQL function take_sample(integer) line 69 at PERFORM
 +|
           | PL/pgSQL function take_sample_subset(integer,integer) line 27 at
 assignment+|
           | SQL function "take_sample" statement 1
 +|
           | FATAL:  database "postgresno" does not exist
  |
(2 rows)
```

G.1.5.3. Политика хранения выборок

Политики хранения можно определить на следующих уровнях:

1. *Глобальный*

   Значение параметра `pgpro_pwr.max_sample_age` в файле `postgresql.conf` определяет общее значение параметра хранения, действующее в случае, когда не определены никакие другие параметры.

2. *Сервер*

   Параметр *`max_sample_age`*, указанный при создании сервера или при вызове функции `set_server_max_sample_age(сервер)`, определяет срок хранения на уровне сервера. Значение этого параметра переопределяет значение `pgpro_pwr.max_sample_age` для конкретного сервера.

3. *Выборочная линия*

   Созданная выборочная линия переопределяет все другие заданные сроки хранения для включённых в неё выборок.

G.1.6. Управление анализом размеров отношений

Сбор сведений о размерах всех отношений в базе данных с использованием функций Postgres Pro может занять продолжительное время. Кроме того, эти функции требуют установления блокировки `AccessExclusiveLock` для анализируемых отношений. Однако во многих случаях может быть достаточно собирать информацию о размерах один раз в сутки. Реализовать это можно, определив в pgpro_pwr *политики анализа размеров* для серверов, позволяющие пропускать сбор сведений о размерах в процессе получения выборок. Такая политика устанавливает:

- Окно в течение суток, когда разрешается сбор сведений о размерах отношений.
- Минимальный промежуток между двумя выборками со сведениями о размерах.

Когда определена политика анализа размеров, функции получения выборок собирают сведения о размерах только в заданном окне и только если после предыдущей выборки с размерами прошло не меньше заданного промежутка. Такую политику позволяют определить следующие функции:

`set_server_size_sampling(`*`server`*` name, `*`window_start`*` time with time zone DEFAULT NULL, `*`window_duration`*` interval hour to second DEFAULT NULL, `*`sample_interval`*` interval day to minute DEFAULT NULL, `*`collect_mode`*` text DEFAULT NULL)`

Определяет политику сбора сведений о размере для сервера.

Аргументы:

- *`server`* — имя сервера.
- *`window_start`* — время начала периода сбора.
- *`window_duration`* — длительность периода сбора.
- *`sample_interval`* — минимальный промежуток времени между сборами сведений о размере.
- *`collect_mode`* — при значении `off`, используемом по умолчанию для новых инсталляций, размеры отношений собираются из каталога `pg_class`, при значении `on`, размеры отношений собираются с помощью функции `pg_relation_size()`, при значении `schedule`, pgpro_pwr собирает размеры отношений в заданном окне. Этот параметр переопределяет значение параметра расширения relsize_collect_mode. При обновлении с предыдущих версий для этого параметра устанавливается значение `on` или `schedule`, что не меняет поведения, имевшего место до обновления.

Примечание

Если вы построите отчёт между двумя выборками, хотя бы в одной из которых отсутствуют сведения о размерах отношений, в нём будут разделы с информацией об увеличивающихся в размере объектах, полученной из `pg_class.relpages`. Однако вы можете расширить интервал отчёта до ближайших выборок, содержащих сведения о размерах, передав функциям построения отчётов параметр *`with_growth`*; это позволяет получать более точные данные об увеличении размера.

Размеры отношений необходимо знать, чтобы вычислить объём последовательно сканированных данных в таблицах и непосредственную нагрузку автоочистки при обработке индексов.

Пример:

```
SELECT set_server_size_sampling('local','23:00+03',interval '2 hour',interval '8 hour',
 'schedule');
```

Функция `show_servers_size_sampling` выводит политики анализа размеров для всех серверов:

```
postgres=# SELECT * FROM show_servers_size_sampling();
 server_name | window_start | window_end  | window_duration | sample_interval |
 limited_collection
```

```
-------------+--------------+-------------+----------------+----------------
+--------------------
 local        | 23:00:00+03  | 01:00:00+03 | 02:00:00       | 08:00:00       | t
```

G.1.7. Управление промежуточными выборками

Некоторые доступные в Postgres Pro данные, связанные с производительностью, не являются накопительными. Например, наиболее часто используемые данные о состояниях сеансов доступны через представление pg_stat_activity и могут быть получены только с помощью частых выборок. Однако функция `take_sample()` создаёт большую нагрузку и может занять много времени. Поэтому она не подходит для сбора данных о состоянии сеансов.

В механизме *промежуточных выборок* реализована новая быстрая функция `take_subsample()`. Она может использоваться для сбора относительно быстро меняющихся данных. Каждая промежуточная выборка привязана к следующей обычной выборке и удаляется вместе с ней согласно заданной политике хранения.

Механизм промежуточных выборок можно использовать для захвата наиболее интересных состояний сеансов:

- Длительные запросы
- Длительные транзакции
- Старые транзакции, то есть те, снимок в которых находится позади множества других транзакций
- Транзакции, долгое время находящиеся в состоянии простоя (idle)

G.1.7.1. Функции получения промежуточных выборок

Следующие функции pgpro_pwr относятся к работе с промежуточными выборками:

`take_subsample()`
`take_subsample(`*`server`*` name)`

Без параметров эта функция получает промежуточные выборки со всех *включённых* серверов с включённым механизмом их получения (за подробностями обратитесь к set_server_subsampling). Промежуточные выборки для серверов получаются последовательно, по одному. Функция выдаёт в результате таблицу со следующими столбцами:

- `server` — имя сервера.
- `result` — результат получения промежуточной выборки. Может быть строкой `OK`, если промежуточная выборка получена успешно, либо содержать текст ошибки в случае неудачи.
- `elapsed` — время, потраченное на получение промежуточной выборки.

Этот табличный формат вывода данных позволяет легко управлять созданием промежуточных выборок с помощью SQL-запроса.

С параметром эта функция получает промежуточную выборку с указанного сервера. Она полезна, когда нужно получать промежуточные выборки с разных серверов с разной частотой или получить явную выборку для конкретного сервера.

Аргументы:

- *`server`* — имя сервера.

Примечание

Попытка получить промежуточную выборку во время получения обычной выборки завершится ошибкой.

`take_subsample_subset([sets_cnt integer], [current_set integer])`

Получает промежуточную выборку с каждого сервера в подмножестве с включённым механизмом их получения. Хотя даже последовательное получение промежуточных выборок должно выполняться достаточно быстро, их можно получать параллельно, аналогично обычным выборкам. Эта функция возвращает тот же тип, что и `take_subsample()`. Если оба параметра опущены, функция действует аналогично `take_subsample()`.

Аргументы:

- *`sets_cnt`* — количество подмножеств серверов.
- *`current_set`* — номер подмножества, в котором будут собираться промежуточные выборки. Принимает значение от 0 до *`sets_cnt`* - 1. Для выбранного подмножества промежуточные выборки собираются как обычно, с последовательным переходом от сервера к серверу.

G.1.7.2. Конфигурирование механизма получения промежуточных выборок

На поведение промежуточных выборок влияют следующие параметры:

- `pgpro_pwr.subsample_enabled` — определяет, должна ли функция `take_subsample()` фактически создавать промежуточную выборку.
- `pgpro_pwr.min_query_duration` — предел, по достижении которого запрос считается длительным.
- `pgpro_pwr.min_xact_duration` — предел, по достижении которого транзакция считается длительной.
- `pgpro_pwr.min_xact_age` — предел возраста транзакции.
- `pgpro_pwr.min_idle_xact_dur_age` — предел простоя транзакции.

Это поведение можно задать на уровне сервера с помощью функции set_server_subsampling.

Последнее наблюдаемое состояние сеанса сохраняется в репозитории, когда происходит одно из следующих событий, связанных с пределами:

- Во время выполнения запроса разница между `now()` и `query_start` превышает предел `pgpro_pwr.min_query_duration`.
- Во время выполнения транзакции разница между `now()` и `xact_start` превышает предел `pgpro_pwr.min_xact_duration`.
- Во время выполнения транзакции `age(backend_xmin)` превышает предел `pgpro_pwr.min_xact_age`.
- Во время выполнения транзакции в состоянии `idle in transaction` или `idle in transaction (aborted)` разница между `now()` и `state_change` превышает предел `pgpro_pwr.min_idle_xact_duration`.

За более подробным описанием упомянутых полей обратитесь к Главе 26. Каждая промежуточная выборка может содержать не более `pgpro_pwr.max` записей для каждого типа пределов.

G.1.7.3. Планирование получения промежуточных выборок

Скорость получения промежуточных выборок позволяет получать их довольно часто. Однако обычно требуется не более 2-4 промежуточных выборок в минуту. Очевидно, что частота промежуточных выборок зависит от самого маленького из значений параметров пределов.

Cron допускает только один вызов в минуту, поэтому нужно предпринять дополнительные действия, чтобы получать промежуточные выборки чаще. Например, можно использовать команду psql `\watch`:

```
echo "select take_subsample(); \watch 15" | psql &> /dev/null
```

Вызов psql можно обернуть в systemd следующим образом:

```
Description=pgpro_pwr subsampling unit
[Unit]

[Service]
Type=simple
ExecStart=/bin/sh -c 'echo "select take_subsample(); \\watch 15" | /path/to/psql -qo /
dev/null'
User=postgres
Group=postgres

[Install]
WantedBy=multi-user.target
```

G.1.8. Управление выборочными линиями

Выборочной линией называется именованная последовательность выборок, для которых определяется отдельный срок хранения. Выборочная линия также может задавать интервал для функций построения отчётов. Если срок хранения для выборочной линии не определён, относящиеся к ней выборки не будут удаляться. Выборочные линии позволяют сохранить информацию о нагрузке базы данных за определённый интервал времени.

G.1.8.1. Функции управления выборочными линиями

Для управления выборочными линиями предназначены следующие функции pgpro_pwr:

`create_baseline([`*`server`*` name,] `*`baseline`*` varchar(25), `*`start_id`*` integer, `*`end_id`*` integer [, `*`days`*` integer])`
`create_baseline([`*`server`*` name,] `*`baseline`*` varchar(25), `*`time_range`*` tstzrange [, `*`days`*` integer])`

Создаёт выборочную линию.

Аргументы:

- *`server`* — имя сервера. Если оно опущено, подразумевается `local`.
- *`baseline`* — имя выборочной линии, которое должно быть уникальным на сервере.
- *`start_id`* — идентификатор первой выборки в линии.
- *`end_id`* — идентификатор последней выборки в линии.
- *`time_range`* — период времени для выборочной линии. Выборочная линия будет включать все выборки, относящиеся к минимальному интервалу, который покрывает заданный отрезок *`time_range`*.
- *`days`* — срок хранения выборочной линии, определяемый целым количеством дней с момента `now()`. Чтобы срок хранения не ограничивался, опустите этот параметр или задайте значение NULL.

`drop_baseline([`*`server`*` name,] `*`baseline`*` varchar(25))`

Удаляет выборочную линию. Аргументы этой функции действуют аналогично одноимённым аргументам `create_baseline`. Удаление выборочной линии не влечёт за собой непосредственное удаление относящихся к ней выборок. На них просто перестают распространяться заданные для этой линии параметры хранения.

`keep_baseline([`*`server`*` name,] `*`baseline`*` varchar(25) [, `*`days`*` integer])`

Изменяет срок хранения для выборочной линии. Аргументы этой функции действуют аналогично одноимённым аргументам `create_baseline`. Чтобы изменить срок хранения для всех существующих выборочных линий, опустите параметр *`baseline`* или передайте в нём NULL.

`show_baselines([server name])`

Выводит существующие выборочные линии. Вызовите `show_baselines`, чтобы получить информацию о выборочных линиях, включающую их имена, интервалы и периоды хранения. Если параметр *`server`* опущен, подразумевается сервер `local`.

G.1.9. Экспорт и импорт данных

Собранные выборки можно экспортировать из одного экземпляра расширения pgpro_pwr и загрузить в другой. Это позволяет перемещать накопленную информацию о серверах или передать её специалистам поддержки для анализа.

G.1.9.1. Экспорт данных

Функция export_data экспортирует данные в обычную таблицу. Выгрузить их затем из базы данных можно любым доступным способом. Например, можно воспользоваться метакомандой psql `\copy` и получить данные в файле `csv`:

```
postgres=# \copy (select * from export_data()) to 'export.csv'
```

G.1.9.2. Импорт данных

Так как данные могут импортироваться только из локальной таблицы, сначала загрузите ранее экспортированные данные. Для этого вновь воспользуйтесь метакомандой `\copy`:

```
postgres=# CREATE TABLE import (section_id bigint, row_data json);
CREATE TABLE
postgres=# \copy import from 'export.csv'
COPY 6437
```

Теперь вы можете импортировать данные, передав таблицу `import` функции import_data:

```
postgres=# SELECT * FROM import_data('import');
```

По завершении импорта таблицу `import` можно удалить.

> **Примечание**
>
> Если данные сервера импортируются впервые, локально определённые серверы pgpro_pwr с совпадающими именами вызовут конфликт во время импорта. Чтобы его избежать, можно временно переименовать такие серверы или указать префикс имён серверов для операций импорта. Однако во время последующего импорта данных для ранее импортированных серверов эти данные будут сопоставляться с серверами по системным идентификаторам, поэтому эти серверы можно переименовать как угодно. Также имейте в виду, что pgpro_pwr переводит импортируемые серверы в отключённое состояние, чтобы функция `take_sample()` не обрабатывала их.

G.1.9.3. Функции экспорта и импорта

Для экспорта и импорта данных предназначены следующие функции:

`export_data([server name, [min_sample_id integer,] [max_sample_id integer,]] [, obfuscate_queries boolean])`

Экспортирует собранные данные.

Аргументы:

- *`server`* — имя сервера. Если оно опущено, данные экспортируется со всех настроенных серверов.
- *`min_sample_id`*, *`max_sample_id`* — идентификаторы выборок, ограничивающих диапазон экспортируемых выборок (включающий указанные границы). Если *`min_sample_id`* опу-

щен или равен NULL, экспортируются все выборки до выборки *`max_sample_id`*; если же *`max_sample_id`* опущен или равен NULL, экспортируются все выборки, начиная с выборки *`min_sample_id`*.

- *`obfuscate_queries`* — если равняется `true`, тексты запросов экспортируются в виде хеша `MD5`.

`import_data(`*`data`*` regclass [, `*`server_name_prefix`*` text])`

Импортирует ранее экспортированные данные. Возвращает число строк, фактически загруженных в таблицы pgpro_pwr.

Аргументы:

- *`data`* — имя таблицы, содержащей данные импорта.
- *`server_name_prefix`* задаёт префикс имён серверов для операции импорта. Его можно использовать, чтобы избежать конфликтов имён.

G.1.10. Функции построения отчётов

Отчёты pgpro_pwr формируются в формате HTML функциями построения. В pgpro_pwr имеются следующие типы отчётов:

- *Обычные отчёты* предоставляют статистику по нагрузке для заданного интервала.
- *Разностные отчёты* предоставляют статистику по одинаковым объектам в двух интервалах. Соответствующие значения выводятся рядом, что позволяет легко сравнивать профили нагрузки.

Функции построения отчётов позволяют определить интервалы разными способами, принимая идентификаторы выборок, выборочные линии или диапазоны времени. В последнем случае выбирается минимальный интервал, охватывающий указанный диапазон.

G.1.10.1. Обычные отчёты

Для построения обычных отчётов воспользуйтесь следующими функциями:

```
get_report([server name,] start_id integer, end_id integer [, description text [,
with_growth boolean [, db_exclude name[]]]])
get_report([server name,] time_range tstzrange [, description text [, with_growth boolean
[, db_exclude name[]]]])
get_report([server name,] baseline varchar(25) [, description text [, with_growth boolean
[, db_exclude name[]]]])
```

Строит обычный отчёт согласно заданным аргументам.

Аргументы:

- *`server`* — имя сервера. Если оно опущено, подразумевается `local`.
- *`start_id`* — идентификатор выборки, с которой начинается интервал.
- *`end_id`* — идентификатор выборки, которой заканчивается интервал.
- *`baseline`* — имя выборочной линии.
- *`time_range`* — временной диапазон.
- *`description`* — короткий текст, который будет включён в отчёт в качестве его описания.
- *`with_growth`* — флаг, позволяющий расширить интервал до ближайших выборок, в которых имеются данные об увеличении отношений. Значение по умолчанию: `false`.
- *`db_exclude`* — список исключаемых баз данных. Содержит базы данных, исключаемых из всех таблиц отчёта, содержащих столбец `Database`. Используйте этот параметр, чтобы скрыть в отчёте выбранные базы данных.

```
get_report_latest([server name,])
get_report_latest([server name [, db_exclude name[]]])
```

Строит обычный отчёт для двух последних выборок.

Аргументы:

- *server* — имя сервера. Если оно опущено, подразумевается `local`.
- *db_exclude* — список исключаемых баз данных. Содержит базы данных, исключаемых из всех таблиц отчёта, содержащих столбец `Database`. Используйте этот параметр, чтобы скрыть в отчёте выбранные базы данных.

G.1.10.2. Сравнительные отчёты

Для построения сравнительных отчётов воспользуйтесь следующей функцией:

```
get_diffreport([server name,] start1_id integer, end1_id integer, start2_id integer,
end2_id integer [, description text [, with_growth boolean [, db_exclude name[]]]])
get_diffreport([server name,] time_range1 tstzrange, time_range2 tstzrange [, description
text [, with_growth boolean [, db_exclude name[]]]])
get_diffreport([server name,] baseline1 varchar(25), baseline2 varchar(25) [, description
text [, with_growth boolean [, db_exclude name[]]]])
get_diffreport([server name,] baseline1 varchar(25), time_range2 tstzrange [, description
text [, with_growth boolean[, db_exclude name[]]]])
get_diffreport([server name,] time_range1 tstzrange, baseline2 varchar(25) [, description
text [, with_growth boolean[, db_exclude name[]]]])
get_diffreport([server name,] start1_id integer, end1_id integer, baseline2 varchar(25)
[, description text [, with_growth boolean[, db_exclude name[]]]])
get_diffreport([server name,] baseline1 varchar(25), start2_id integer, end2_id integer
[, description text [, with_growth boolean[, db_exclude name[]]]])
```

Формирует сравнительный отчёт за два интервала. Различные сочетания аргументов позволяют задать интервалы самыми разными способами.

Аргументы:

- *server* — имя сервера. Если оно опущено, подразумевается `local`.
- *start1_id*, *end1_id* — идентификаторы начальной и конечной выборок для первого интервала.
- *start2_id*, *end2_id* — идентификаторы начальной и конечной выборок для второго интервала.
- *baseline1* — имя выборочной линии для первого интервала.
- *baseline2* — имя выборочной линии для второго интервала.
- *time_range1* — временной диапазон, задающий первый интервал.
- *time_range2* — временной диапазон, задающий второй интервал.
- *description* — короткий текст, который будет включён в отчёт в качестве его описания.
- *with_growth* — флаг, позволяющий расширить интервал до ближайших выборок, в которых имеются данные об увеличении отношений. Значение по умолчанию: `false`.
- *db_exclude* — список исключаемых баз данных. Содержит базы данных, исключаемых из всех таблиц отчёта, содержащих столбец `Database`. Используйте этот параметр, чтобы скрыть в отчёте выбранные базы данных.

G.1.10.3. Пример построения отчёта

Построение отчёта для локального сервера (`local`) за интервал, определяемый выборками:

```
psql -Aqtc "SELECT profile.get_report(480,482)" -o report_480_482.html
```

Построение отчёта для другого сервера:

```
psql -Aqtc "SELECT profile.get_report('omega',12,14)" -o report_omega_12_14.html
```

Построение отчёта за временной диапазон:

```
psql -Aqtc "SELECT profile.get_report(tstzrange('2020-05-13 11:51:35+03','2020-05-13
 11:52:18+03'))" -o report_range.html
```

Построение отчёта за временной диапазон, определяемый относительно:

```
psql -Aqtc "SELECT profile.get_report(tstzrange(now() - interval '1 day',now()))" -o
 report_last_day.html
```

G.1.11. Разделы отчёта pgpro_pwr

Каждый отчёт pgpro_pwr включает в себя разделы, описанные ниже. Число первых объектов, выбираемых из отсортированной таблицы при построении отчёта, задаётся параметром `pgpro_pwr.max`.

Почти любой элемент в отчёте можно выделить одним щелчком мыши. Выбранный элемент будет мгновенно выделен во всех разделах отчёта, что облегчит его поиск. Атрибуты, идентифицирующие элемент, появятся в правом нижнем углу страницы. Например, если выбрать имя базы данных в таблице отчёта «Database statistics» (Статистика базы данных), можно заметить небольшую таблицу с атрибутами базы данных в правом нижнем углу страницы.

При прокрутке отчёта вниз его оглавление будет доступно в правой части страницы. Его можно скрыть одним щелчком мыши по закладке «content».

Также доступен фильтр на основе подстроки, который помогает ограничить содержимое отчёта определёнными объектами. В частности, фильтрация на основе подстроки применяется к текстам запросов.

G.1.11.1. Server statistics (Серверная статистика)

Ниже описаны таблицы, относящиеся к этому разделу отчёта pgpro_pwr.

Таблицы «Database statistics» содержат статистику, собранную за интервал времени, в разрезе баз данных. Эта статистика основана на содержимом представления `pg_stat_database`. Столбцы такой таблицы отчёта перечислены в Таблице G.1.

Таблица G.1. Database statistics (Статистика баз данных)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	`datname`
`Commits`	Число зафиксированных транзакций	`xact_commit`
`Rollbacks`	Число отменённых транзакций	`xact_rollback`
`Deadlocks`	Число выявленных взаимоблокировок	`deadlocks`
`Checksum Failures`	Количество ошибок контрольных сумм в страницах данных этой базы. Это поле отображается, только если за отчётный интервал в этой базе данных были обнаружены какие-либо ошибки контрольной суммы.	`checksum_failures`
`Checksums Last`	Время выявления последней ошибки контрольной суммы в страницах данных этой базы. Это поле отображается, толь-	`checksum_last_failure`

Столбец	Описание	Поле/вычисление
	ко если за отчётный интервал в этой базе данных были обнаружены какие-либо ошибки контрольной суммы.	
`Hit%`	Процент попаданий в кеш, то есть отношение числа страниц, прочитанных из буферов, к общему числу страниц	
`Read`	Количество прочитанных дисковых блоков в этой базе данных	`blks_read`
`Hit`	Сколько раз требуемые блоки с диска уже находились в кеше	`blks_hit`
`Ret`	Количество выданных кортежей	`tup_returned`
`Fet`	Количество считанных кортежей	`tup_fetched`
`Ins`	Количество вставленных кортежей	`tup_inserted`
`Upd`	Количество изменённых кортежей	`tup_updated`
`Del`	Количество удалённых кортежей	`tup_deleted`
`Temp Size`	Общий объём данных, записанный во временные файлы при выполнении запросов в этой базе данных	`temp_bytes`
`Temp Files`	Количество временных файлов, созданных запросами в этой базе данных	`temp_files`
`Size`	Размер базы данных в момент получения последней выборки в отчётном интервале	`pg_database_size()`
`Growth`	Прирост объёма базы данных за отчётный интервал	Приращение `pg_database_size()` на конец интервала относительно начала

Таблица отчёта «Cluster I/O statistics» (Статистика ввода-вывода кластера) показывает статистику ввода-вывода по типам объектов, типам обслуживающих процессов и контекстам. Эта таблица основана на представлении `pg_stat_io` Системы накопительной статистики, доступном начиная с Postgres Pro 16. Столбцы этой таблицы перечислены в Таблице G.2. Значения времени в ней выражаются в секундах.

Таблица G.2. Cluster I/O statistics (Статистика ввода-вывода кластера)

Столбец	Описание
`Object`	Целевой объект операции ввода-вывода
`Backend`	Тип обслуживающего процесса, выполнившего операцию ввода-вывода
`Context`	Контекст операции ввода-вывода
`Reads Count`	Количество операций чтения

Столбец	Описание
`Reads Bytes`	Объём прочитанных данных
`Reads Time`	Время, затраченное операциями чтения
`Writes Count`	Количество операций записи
`Writes Bytes`	Объём записанных данных
`Writes Time`	Время, затраченное операциями записи
`Writebacks Count`	Количество блоков, запрошенных процессом для записи ядром в постоянное хранилище
`Writebacks Bytes`	Объём данных, запрошенных для записи в постоянное хранилище
`Writebacks Time`	Время, затраченное на операции отложенной записи, в том числе на постановку в очередь запросов на запись, и, возможно, на запись «грязных» данных
`Extends Count`	Количество операций расширения отношений
`Extends Bytes`	Объём пространства, использованного операциями расширения
`Extends Time`	Время, затраченное операциями расширения
`Hits`	Сколько раз нужный блок был найден в общем буфере
`Evictions`	Сколько раз блок был записан из общего или локального буфера, чтобы его можно было переиспользовать
`Reuses`	Сколько раз существующий буфер в кольцевом буфере с ограниченным размером за пределами общих буферов был повторно использован как часть операции ввода-вывода в контекстах `bulkread`, `bulkwrite` или `vacuum`
`Fsyncs Count`	Количество вызовов fsync. Они отслеживаются только в контексте `normal`
`Fsyncs Time`	Время, затраченное операциями синхронизации с файловой системой

Таблица отчёта «Cluster SLRU statistics» (Статистика SLRU-кеша кластера) показывает статистику доступа к SLRU-кешам (simple least-recently-used, простое вытеснение давно не используемых). Эта таблица основана на представлении `pg_stat_slru` Системы накопительной статистики. Столбцы этой таблицы перечислены в Таблице G.3. Значения времени в ней выражаются в секундах.

Таблица G.3. Cluster SLRU statistics (Статистика SLRU-кеша кластера)

Столбец	Описание	Поле/вычисление
`Name`	Имя SLRU-кеша	`name`
`Zeroed`	Количество блоков, обнулённых при инициализации	`blks_zeroed`
`Hits`	Сколько раз дисковые блоки обнаруживались в SLRU-кеше и чтение с диска не требовалось (здесь учитываются только случаи обнаружения в этом кеше, а не в файловом кеше ОС)	`blks_hit`

Столбец	Описание	Поле/вычисление
`Reads`	Количество дисковых блоков, прочитанных для этого SLRU-кеша	`blks_read`
`%Hit`	Количество попаданий дискового блока для этого SLRU-кеша в процентах от `Reads` + `Hits`	`blks_hit` *100/`blks_read` + `blks_hit`
`Writes`	Количество дисковых блоков, записанных для этого SLRU-кеша	`blks_written`
`Checked`	Количество блоков, проверенных на предмет наличия в этом SLRU-кеше	`blks_exists`
`Flushes`	Количество операций сброса «грязных» данных для этого SLRU-кеша	`flushes`
`Truncates`	Количество операций усечения для этого SLRU-кеша	`truncates`

Таблица «Session statistics by database» выводится в отчёте для баз Postgres Pro, начиная с версии 14. Эта таблица основана на представлении `pg_stat_database` сборщика статистики. Столбцы этой таблицы перечислены в Таблице G.4. Значения времени в ней выражаются в секундах.

Таблица G.4. Session statistics by database (Статистика сеансов по базам данных)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	
`Timing Total`	Длительность сеансов в этой базе за отчётный интервал (обратите внимание, что статистика обновляется только при изменении состояния сеанса, поэтому, если сеансы простаивают в течение длительного времени, время простоя не будет учитываться)	`session_time`
`Timing Active`	Время, затраченное на выполнение операторов SQL в этой базе за отчётный интервал (соответствует состояниям `active` и `fastpath function call` в `pg_stat_activity`)	`active_time`
`Timing Idle(T)`	Время простоя в транзакциях в этой базе за отчётный интервал (соответствует состояниям `idle in transaction` и `idle in transaction (aborted)` в `pg_stat_activity`)	`idle_in_transaction_time`
`Sessions Established`	Общее количество сеансов, относящихся к этой базе, за отчётный интервал	`sessions`
`Sessions Abandoned`	Количество сеансов в этой базе данных, прерванных из-за поте-	`sessions_abandoned`

Столбец	Описание	Поле/вычисление
	ри соединения с клиентом, за отчётный интервал	
`Sessions Fatal`	Количество сеансов в этой базе данных, прерванных из-за критических ошибок, за отчётный интервал	`sessions_fatal`
`Sessions Killed`	Количество сеансов в этой базе данных, прерванных из-за вмешательства оператора, за отчётный интервал	`sessions_killed`

В базах данных Postgres Pro Standard версий, включающих версию pgpro_stats 1.4 и выше, доступна статистика нагрузки процессов очистки. В отчёте выводится таблица «Database vacuum statistics», содержащая общую агрегированную статистику очистки по базам данных из `pgpro_stats_vacuum_tables`. Столбцы этой таблицы перечислены в Таблице G.5. Время в ней указывается в секундах.

Таблица G.5. Database vacuum statistics (Статистика очистки баз данных)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	
`Blocks fetched`	Общее количество блоков БД, полученных операциями очистки	`total_blks_read` + `total_blks_hit`
`%Total`	Общее количество блоков БД (прочитанных и найденных в общих буферах), полученных операциями очистки, в процентах от общего числа блоков, полученных в кластере	`Blocks fetched` * 100 / `Cluster fetched`
`Blocks read`	Общее количество блоков БД, прочитанных операциями очистки	`total_blks_read`
`%Total`	Общее количество блоков БД, прочитанных операциями очистки, в процентах от общего числа блоков, прочитанных в кластере	`Blocks read` * 100 / `Cluster read`
`VM Frozen`	Общее количество блоков, помеченных в карте видимости как полностью замороженные	`pages_frozen`
`VM Visible`	Общее количество блоков, помеченных в карте видимости как полностью видимые	`pages_all_visible`
`Tuples deleted`	Общее количество «мёртвых» кортежей, удалённых операциями очистки из таблиц этой БД	`tuples_deleted`
`Tuples left`	Общее количество «мёртвых» кортежей, оставленных операциями очистки в таблицах этой БД из-за видимости этих кортежей в транзакциях	`dead_tuples`

Столбец	Описание	Поле/вычисление
`%Eff`	Эффективность очистки, оцениваемая по количеству удалённых кортежей. Это процент кортежей, удалённых из таблиц этой базы данных, от всех «мёртвых» кортежей, подлежащих удалению из этих таблиц.	`tuples_deleted` * 100 / (`tuples_deleted` + `dead_tuples`)
`WAL size`	Общий объём WAL (в байтах), сгенерированный операциями очистки, выполненными для таблиц этой БД	`wal_bytes`
`Read I/O time`	Время, затраченное на чтение блоков БД операциями очистки, выполненными для таблиц этой БД	`blk_read_time`
`Write I/O time`	Время, затраченное на запись блоков БД операциями очистки, выполненными для таблиц этой БД	`blk_write_time`
`%Total`	Время, затраченное на чтение/запись в процессе очистки, в процентах от всего времени чтения/записи в кластере	
`Total vacuum time`	Общее время, затраченное на очистку таблиц этой БД	`total_time`
`Delay vacuum time`	Время простоя в точке задержки при выполнении операций очистки таблиц этой БД	`delay_time`
`User CPU time`	Время использования процессора в пользовательском режиме при очистке таблиц этой БД	`user_time`
`System CPU time`	Время использования процессора в режиме ядра при очистке таблиц этой БД	`system_time`
`Interrupts`	Сколько раз операции очистки, выполнявшиеся для таблиц этой БД, были прерваны из-за каких-либо ошибок	`interrupts`

Если в отчётном интервале было доступно расширение pgpro_stats, поддерживающее статистики аннулирования, в отчёте выводится таблица «Invalidation messages by database» с общей агрегированной статистикой событий аннулирования по каждой базе данных. Столбцы этой таблицы перечислены в Таблице G.6.

Таблица G.6. Invalidation messages by database (Число событий аннулирования в базе данных)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	
`Invalidation messages sent`	Общее количество событий аннулирования, отправленных обслуживающими процессами в этой базе данных. Статисти-	Поля столбца `pgpro_stats_totals .inval_msgs`

Столбец	Описание	Поле/вычисление
	ка предоставляется для соответствующих типов сообщений `pgpro_stats_inval_msgs`	
`Cache resets`	Общее число сбросов разделяемого кеша	`pgpro_stats_totals .cache_resets`

Если в отчётном интервале было доступно расширение pgpro_stats, в отчёте выводится таблица «Statement statistics by database» с общей агрегированной статистикой из `pgpro_stats_statements` по каждой базе данных. Столбцы этой таблицы перечислены в Таблице G.7. Значения времени в ней выражаются в секундах.

Таблица G.7. Statement statistics by database (Статистика SQL-операторов в базе данных)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	
`Calls`	Счётчик всех выполненных SQL-операторов в базе	`calls`
`Plan Time`	Время, затраченное на планирование операторов в этой базе	Сумма значений `total_plan_time`
`Exec Time`	Время, затраченное на выполнение всех операторов в этой базе	Сумма значений `total_exec_time`
`Read Time`	Время, затраченное на чтение блоков в этой базе	Сумма значений `blk_read_time`
`Write Time`	Время, затраченное на запись блоков в этой базе	Сумма значений `blk_write_time`
`Trg Time`	Время, затраченное на выполнение триггерных функций в этой базе	
`Shared Fetched`	Общее количество разделяемых блоков, прочитанных в этой базе	Сумма значений `shared_blks_read + shared_blks_hit`
`Local Fetched`	Общее количество локальных блоков, прочитанных в этой базе	Сумма значений `local_blks_read + local_blks_hit`
`Shared Dirtied`	Общее количество разделяемых блоков, загрязнённых при выполнении операторов в этой базе	Сумма значений `shared_blks_dirtied`
`Local Dirtied`	Количество прочитанных локальных блоков, загрязнённых при выполнении операторов в этой базе	Сумма значений `local_blks_dirtied`
`Read Temp`	Общее количество временных блоков, прочитанных в этой базе	Сумма значений `temp_blks_read`
`Write Temp`	Общее количество временных блоков, записанных в этой базе	Сумма значений `temp_blks_written`
`Read Local`	Общее количество локальных блоков, прочитанных в этой базе	Сумма значений `local_blks_read`

Столбец	Описание	Поле/вычисление
`Write Local`	Общее количество локальных блоков, записанных в этой базе	Сумма значений `local_blks_written`
`Statements`	Общее число обработанных SQL-операторов	
`WAL Size`	Общий объём WAL, сгенерированный в этой базе	Сумма значений `wal_bytes`

Таблица отчёта «Statement average min/max timings» содержит агрегированную статистику о минимальных/максимальных замерах времени за отчётный интервал, в разрезе баз данных, из представлений расширения pgpro_stats или pg_stat_statements (предпочтение отдаётся представлению pgpro_stats). Этот отчёт учитывает самое быстрое и самое медленное планирование и выполнение каждого оператора в кластере, то есть позволяет увидеть стабильность выполнения и планирования в базе данных. Столбцы такой таблицы отчёта перечислены в Таблице G.8.

Таблица G.8. Statement average min/max timings (Средние значения минимальных/максимальных замеров времени)

Столбец	Описание
`Database`	Имя базы данных
`Min average planning time (ms)`	Среднее значение `min_plan_time` для всех операторов и всех выборок, включённых в отчёт
`Max average planning time (ms)`	Среднее значение `max_plan_time` для всех операторов и всех выборок, включённых в отчёт
`Delta% of average planning times`	Разность среднего значения `max_plan_time` и среднего значения `min_plan_time` в процентах от среднего значения `min_plan_time`. Чем меньше эта разность, тем стабильнее планирование запросов в базе данных.
`Min average execution time (ms)`	Среднее значение `min_exec_time` для всех операторов и всех выборок, включённых в отчёт
`Max average execution time (ms)`	Среднее значение `max_exec_time` для всех операторов и всех выборок, включённых в отчёт
`Delta% of average execution times`	Разность среднего значения `max_exec_time` и среднего значения `min_exec_time` в процентах от среднего значения `min_exec_time`. Чем меньше эта разность, тем стабильнее выполнение запросов в базе данных.
`Statements`	Общее число операторов, попавших в выборки

Если расширение, собирающее статистику операторов в отчётном интервале, собрало статистику JIT, в отчёте выводится таблица «JIT statistics by database» с общей агрегированной статистикой по использованию JIT в разрезе баз данных. Столбцы этой таблицы перечислены в Таблице G.9. Значения времени в ней выражаются в секундах.

Таблица G.9. JIT statistics by database (Статистика JIT в базах данных)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	

Столбец	**Описание**	**Поле/вычисление**
`Calls`	Счётчик всех выполненных SQL-операторов в базе	`calls`
`Plan Time`	Время, затраченное на планирование операторов в этой базе	Сумма значений `total_plan_time`
`Exec Time`	Время, затраченное на выполнение всех операторов в этой базе	Сумма значений `total_exec_time`
`Generation count`	Общее число функций, скомпилированных в JIT-код при выполнении операторов	Сумма значений `jit_functions`
`Generation time`	Общее время, затраченное на компиляцию JIT-кода при выполнении операторов	Сумма значений `jit_generation_time`
`Inlining count`	Сколько раз встраивались функции	Сумма значений `jit_inlining_count`
`Inlining time`	Общее время, затраченное на встраивание функций при выполнении операторов	Сумма значений `jit_inlining_time`
`Optimization count`	Общее число JIT-оптимизаций для операторов	Сумма значений `jit_optimization_count`
`Optimization time`	Общее время, затраченное на JIT-оптимизацию при выполнении операторов	Сумма значений `jit_optimization_time`
`Emission count`	Сколько раз выдавался код	Сумма значений `jit_emission_count`
`Emission time`	Общее время, затраченное на выдачу кода при выполнении операторов	Сумма значений `jit_emission_time`
`Deform count`	Число функций преобразования кортежей, скомпилированных в JIT-код при выполнении данного оператора	
`Deform time`	Общее время, затраченное операторами на компилирование функций преобразования кортежей в JIT-код	

В таблице отчёта «Cluster statistics» содержатся данные из представления `pg_stat_bgwriter`. Строки данной таблицы перечислены в Таблице G.10. Значения времени в ней выражаются в секундах.

Таблица G.10. Cluster statistics (Статистика кластера)

Строка	**Описание**	**Поле/вычисление**
`Scheduled checkpoints`	Количество запланированных контрольных точек, которые уже были выполнены	`checkpoints_timed`
`Requested checkpoints`	Количество запрошенных контрольных точек, которые уже были выполнены	`checkpoints_req`
`Checkpoint write time (s)`	Общее время, которое было затрачено на этап обработки кон-	`checkpoint_write_time`

Строка	Описание	Поле/вычисление
	трольной точки, в котором файлы записываются на диск	
`Checkpoint sync time (s)`	Общее время, которое было затрачено на этап обработки контрольной точки, в котором файлы синхронизируются с диском	`checkpoint_sync_time`
`Checkpoint buffers written`	Количество буферов, записанных при выполнении контрольных точек	`buffers_checkpoint`
`Background buffers written`	Количество буферов, записанных фоновым процессом записи	`buffers_clean`
`Backend buffers written`	Количество буферов, записанных самим обслуживающим процессом. Не будет показываться в Postgres Pro 17 и выше.	`buffers_backend`
`Backend fsync count`	Сколько раз обслуживающему процессу пришлось выполнить `fsync` самостоятельно (обычно фоновый процесс записи сам обрабатывает эти вызовы, даже когда обслуживающий процесс выполняет запись самостоятельно). Не будет показываться в Postgres Pro 17 и выше.	`buffers_backend_fsync`
`Bgwriter interrupts (too many buffers)`	Сколько раз фоновый процесс записи останавливал сброс грязных страниц на диск из-за того, что записал слишком много буферов	`maxwritten_clean`
`Number of buffers allocated`	Общее количество выделенных буферов	`buffers_alloc`
`WAL generated`	Общий сгенерированный объём WAL	Приращение значения `pg_current_wal_lsn()`
`Start LSN`	Последовательный номер в журнале в начале отчётного интервала	`pg_current_wal_lsn()` в первой выборке отчёта
`End LSN`	Последовательный номер в журнале в конце отчётного интервала	`pg_current_wal_lsn()` в последней выборке отчёта
`WAL generated by vacuum`	Общий объём WAL, сгенерированный при очистке	Основано на значении поля `wal_bytes` представления `pgpro_stats_vacuum_databases` .
`WAL segments archived`	Общее количество заархивированных сегментов WAL	Основано на значении `pg_stat_archiver` `.archived_count`

Строка	Описание	Поле/вычисление
`WAL segments archive failed`	Общее количество ошибок, возникших при архивировании сегментов WAL	Основано на значении `pg_stat_archiver` `.failed_count` .
`Archiver performance`	Средняя производительность процесса архивирования в секунду	Основано на значении поля `active_time` представления `pgpro_stats_archiver` .
`Archive command performance`	Средняя производительность команды `archive_command` в секунду	Основано на значении поля `archive_command_time` представления `pgpro_stats_archiver` .

Таблица «WAL statistics» выводится в отчёте для баз Postgres Pro, начиная с версии 14. Эта таблица основана на представлении `pg_stat_wal` сборщика статистики. Столбцы этой таблицы отчёта перечислены в Таблице G.11. Значения времени в ней выражаются в секундах.

Таблица G.11. WAL statistics (Статистика WAL)

Строка	Описание	Поле/вычисление
`WAL generated`	Общий объём записей WAL, сгенерированных за отчётный интервал	`wal_bytes`
`WAL per second`	Среднее количество записей WAL, генерируемых в секунду, за отчётный интервал	`wal_bytes` / `report_duration`
`WAL records`	Общее число записей WAL, сгенерированных за отчётный интервал	`wal_records`
`WAL FPI`	Общее число образов полных страниц WAL, сгенерированных за отчётный интервал	`wal_fpi`
`WAL buffers full`	Сколько раз данные WAL записывались на диск из-за переполнения буферов WAL за отчётный интервал	`wal_buffers_full`
`WAL writes`	Сколько раз буферы WAL были записаны на диск функцией `XLogWrite` за отчётный интервал	`wal_write`
`WAL writes per second`	Сколько раз в секунду в среднем буферы WAL записывались на диск функцией `XLogWrite` за отчётный интервал	`wal_write` / `report_duration`
`WAL sync`	Сколько раз файлы WAL сбрасывались на диск функцией `issue_xlog_fsync` за отчётный интервал (если fsync включён и wal_sync_method имеет значение `fdatasync`, `fsync` или `fsync_writethrough` , в противном случае — ноль). Более подробную информацию о внутренней функции WAL `issue_`	`wal_sync`

Строка	Описание	Поле/вычисление
	`xlog_fsync` можно найти в Разделе 27.5.	
`WAL syncs per second`	Сколько раз в секунду в среднем файлы WAL сбрасывались на диск функцией `issue_xlog_fsync` за отчётный интервал	`wal_sync` / `report_duration`
`WAL write time`	Общее время, затраченное на запись буферов WAL на диск функцией `XLogWrite`, за отчётный интервал (если включён `track_wal_io_timing`, в противном случае — ноль; за дополнительными сведениями обратитесь к разделу Разделе 18.9). Учитывается и время синхронизации, когда `wal_sync_method` имеет значение `open_datasync` или `open_sync`.	`wal_write_time`
`WAL write duty`	Процент `WAL write time` от продолжительности отчёта	`wal_write_time` * 100 / `report_duration`
`WAL sync time`	Общее время, затраченное на сброс файлов WAL на диск функцией `issue_xlog_fsync`, за отчётный интервал (если `track_wal_io_timing` включён, значение `fsync` — on и `wal_sync_method` имеет значение `fdatasync`, `fsync` или `fsync_writethrough`, в противном случае — ноль).	`wal_sync_time`
`WAL sync duty`	Процент `WAL sync time` от продолжительности отчёта	`wal_sync_time` * 100 / `report_duration`

В таблице отчёта «Tablespace statistics» содержится информация о размере и приросте табличных пространств. Столбцы этой таблицы перечислены в Таблице G.12.

Таблица G.12. Tablespace statistics (Статистика табличных пространств)

Столбец	Описание	Поле/вычисление
`Tablespace`	Имя табличного пространства	`pg_tablespace.spcname`
`Path`	Путь табличного пространства	`pg_tablespace_location()`
`Size`	Размер табличного пространства в момент получения последней выборки в отчётном интервале	`pg_tablespace_size()`
`Growth`	Прирост объёма табличного пространства за отчётный интервал	Приращение `pg_tablespace_size()` на конец интервала относительно начала

Если в отчётном интервале было доступно расширение pgpro_stats, в отчёте выводится таблица «Wait statistics by database» с информацией об общем времени ожидания в разрезе типов ожидания и баз данных. Столбцы этой таблицы перечислены в Таблице G.13.

Таблица G.13. Wait statistics by database (Статистика ожидания в базах данных)

Столбец	Описание
`Database`	Имя базы данных
`Wait event type`	Тип события, которого ожидали серверные процессы. Звёздочка вместо типа соответствует совокупности всех типов ожидания в базе данных.
`Waited (s)`	Время, проведённое в ожидании событий типа `Wait event type`, в секундах
`%Total`	Отношение времени, проведённого в ожидании событий типа `Wait event type`, к общему времени ожидания таких событий в кластере

Если в отчётном интервале было доступно расширение pgpro_stats, в отчёте выводится таблица «Top wait events» с информацией о наиболее длительных событиях ожидания в кластере. Столбцы этой таблицы перечислены в Таблице G.14.

Таблица G.14. Top wait events (Преобладающие события ожидания)

Столбец	Описание
`Database`	Имя базы данных
`Wait event type`	Тип события, которого ждали серверные процессы
`Wait event`	Название события ожидания, которого ждали серверные процессы
`Waited`	Общее время, проведённое в ожидании событий типа `Wait event` в базе данных, в секундах
`%Total`	Отношение времени, проведённого в ожидании события `Wait event` в базе данных к общему времени ожидания этого события в кластере

G.1.11.2. Load distribution (Распределение нагрузки)

Этот раздел отчёта pgpro_pwr основан на представлении `pgpro_stats_totals` расширения pgpro_stats, если оно было доступно в течение отчётного интервала. Каждая таблица в данном разделе предоставляет данные за отчётный интервал о распределении нагрузки для определённого типа объектов, для которых собирается агрегированная статистика, например, баз данных, приложений, узлов или пользователей. Каждая таблица содержит по одной строке для каждого из ресурсов (таких, как общее время или общее число записанных разделяемых блоков), где распределение нагрузки показано на графике в виде линейчатой диаграммы с накоплением для объектов с наибольшей нагрузкой по этому ресурсу. Если область диаграммы, соответствующая объекту, слишком узка для включения заголовков, наведите указатель на эту область, чтобы получить подсказку с заголовком, значением и процентом. Таблицы «Load distribution among heavily loaded databases», «Load distribution among heavily loaded applications», «Load distribution among heavily loaded hosts» и «Load distribution among heavily loaded users» показывают распределение нагрузки для соответствующих объектов. Строки этих таблиц описаны в Таблице G.15.

Таблица G.15. Load distribution (Распределение нагрузки)

Строка	Описание	Вычисление
`Total time (sec.)`	Общее время, затраченное на планирование и выполнение операторов	`total_plan_time` + `total_exec_time`
`Executed count`	Число выполненных запросов	`queries_executed`

Строка	**Описание**	**Вычисление**
`I/O time (sec.)`	Общее время, затраченное операторами на чтение или запись блоков (если включён track_io_timing, или ноль в противном случае)	`blk_read_time` + `blk_write_time`
`Blocks fetched`	Общее число разделяемых блоков, прочитанных с диска и из кеша, для данного оператора	`shared_blks_hit` + `shared_blks_read`
`Shared blocks read`	Общее количество разделяемых блоков, прочитанных операторами	`shared_blks_read`
`Shared blocks dirtied`	Общее число разделяемых блоков, «загрязнённых» операторами	`shared_blks_dirtied`
`Shared blocks written`	Общее число разделяемых блоков, записанных операторами	`shared_blks_written`
`WAL generated`	Общий объём WAL, сгенерированный при выполнении операторов	`wal_bytes`
`Temp and Local blocks written`	Общее число временных и локальных блоков, записанных операторами	`temp_blks_written` + `local_blks_written`
`Temp and Local blocks read`	Общее количество временных и локальных блоков, прочитанных операторами	`temp_blks_read` + `local_blks_read`
`Invalidation messages sent`	Общее число событий аннулирования, отправленных обслуживающими процессами в этой базе данных	(`pgpro_stats_totals`.`inval_msgs`).all
`Cache resets`	Общее число сбросов разделяемого кеша	`pgpro_stats_totals`.`cache_resets`

G.1.11.3. Состояния сеансов, попадающие в промежуточные выборки

В этом разделе отчёта pgpro_pwr содержится информация о состояниях сеансов, попавших в промежуточные выборки за отчётный интервал.

Ниже описаны таблицы, относящиеся к этому разделу отчёта.

Подраздел отчёта «Chart with session state» показывает состояния сеансов, попавшие в промежуточные выборки. Это график линии времени, иллюстрирующий попавшие в выборку состояния сеансов в обслуживающих процессах и транзакциях. Каждое состояние содержит всплывающее окно с атрибутами состояния сеанса. Нажмите на состояние, чтобы увидеть его в таблице состояний сеансов.

В таблице отчёта «Session state statistics by database» (Статистики состояний сеансов по базам данных) содержатся агрегированные данные о состояниях сеансов. Считаются только состояния сеансов, попавшие в промежуточные выборки. Столбцы этой таблицы перечислены в Таблице G.16.

Таблица G.16. Session state statistics by database (Статистики состояний сеансов по базам данных)

Столбец	**Описание**
`Database`	Имя базы данных

Столбец	Описание
`Summary Active`	Общее время, проведённое в состояниях `active`, которые попали в промежуточные выборки
`Summary Idle in xact`	Общее время состояний `idle in transaction`, попавших в промежуточные выборки
`Summary Idle in xact (A)`	Общее время состояний `idle in transaction (aborted)` , попавших в промежуточные выборки
`Maximal Active`	Продолжительность самого длительного состояния `active`, попавшего в промежуточные выборки
`Maximal Idle in xact`	Продолжительность самого длительного состояния `idle in transaction`, попавшего в промежуточные выборки
`Maximal Idle in xact (A)`	Продолжительность самого длительного состояния `idle in transaction (aborted)` , попавшего в промежуточные выборки
`Maximal xact age`	Максимальный возраст транзакций, попавший в промежуточные выборки

В таблице отчёта «Top 'idle in transaction' session states by duration» показано `pgpro_pwr.max` самых длительных состояний `idle in transaction` для каждого сеанса. Эта информация основана на данных последнего наблюдения каждого из них в представлении pg_stat_activity. Столбцы такой таблицы отчёта перечислены в Таблице G.17.

Таблица G.17. Top 'idle in transaction' session states by duration (Самые длительные состояния 'idle in transaction' сеансов)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	`datname`
`User`	Имя пользователя	`usename`
`App`	Имя приложения	`application_name`
`Pid`	Идентификатор процесса	`pid`
`Xact start`	Временная метка начала транзакции	`xact_start`
`State change`	Временная метка изменения состояния	`state_change`
`State duration`	Продолжительность состояния	Разность `clock_timestamp()` и `state_change`

В таблице отчёта «Top 'active' session states by duration» показаны `pgpro_pwr.max` самых длительных состояний `active` для каждого сеанса. Эта информация основана на данных последнего наблюдения каждого из них в представлении pg_stat_activity. Столбцы такой таблицы отчёта перечислены в Таблице G.18.

Таблица G.18. Top 'active' session states by duration (Самые длительные состояния 'active' сеансов)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	`datname`
`User`	Имя пользователя	`usename`
`App`	Имя приложения	`application_name`

Столбец	Описание	Поле/вычисление
`Pid`	Идентификатор процесса	`pid`
`Xact start`	Временная метка начала транзакции	`xact_start`
`State change`	Временная метка изменения состояния	`state_change`
`State duration`	Продолжительность состояния	Разность `clock_timestamp()` и `state_change`

В таблице отчёта «Top states by transaction age» показаны состояния сеансов с самым большим возрастом транзакций. Эта информация основана на данных последнего наблюдения каждого из них в представлении pg_stat_activity. Столбцы такой таблицы отчёта перечислены в Таблице G.19.

Таблица G.19. Top states by transaction age (Состояния с самым большим возрастом транзакций)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	`datname`
`User`	Имя пользователя	`usename`
`App`	Имя приложения	`application_name`
`Pid`	Идентификатор процесса	`pid`
`Xact start`	Временная метка начала транзакции	`xact_start`
`Xact duration`	Продолжительность транзакции	Разность `clock_timestamp()` и `xact_start`
`Age`	Возраст транзакции	`age( backend_xmin )`
`State`	Состояние сеанса при максимальном обнаруженном возрасте	
`State change`	Временная метка изменения состояния	`state_change`
`State duration`	Продолжительность состояния	Разность `clock_timestamp()` и `state_change`

В таблице отчёта «Top states by transaction duration» показаны самые длительные состояния сеансов. Эта информация основана на данных последнего наблюдения каждого из них в представлении pg_stat_activity. Столбцы такой таблицы отчёта перечислены в Таблице G.19.

Таблица G.20. Top states by transaction duration (Состояния с наибольшей продолжительностью транзакций)

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных	`datname`
`User`	Имя пользователя	`usename`
`App`	Имя приложения	`application_name`
`Pid`	Идентификатор процесса	`pid`
`Xact start`	Временная метка начала транзакции	`xact_start`
`Xact duration`	Продолжительность транзакции	Разность `clock_timestamp()` и `xact_start`
`Age`	Возраст транзакции	`age( backend_xmin )`

Столбец	Описание	Поле/вычисление
`State`	Состояние сеанса при максимальном обнаруженном возрасте	
`State change`	Временная метка изменения состояния	`state_change`
`State duration`	Продолжительность состояния	Разность `clock_timestamp()` и `state_change`

G.1.11.4. Статистика SQL-запросов

В этом разделе отчёта pgpro_pwr представлена информация о наиболее выдающихся по разным критериям операторах за отчётный интервал. В основном эти данные получаются из представлений расширения pgpro_stats и pg_stat_statements (предпочтение отдаётся представлению pgpro_stats). Каждый оператор можно выделить во всех разделах, связанных с SQL, одним щелчком мыши. При этом прямо под строкой статистики запроса также доступен для предварительного просмотра текст запроса, который можно скрыть повторным щелчком мыши по запросу.

Ниже описаны таблицы, относящиеся к этому разделу отчёта.

Таблица отчёта «Top SQL by elapsed time» показывает запросы с наибольшей длительностью, рассчитанной как сумма полей `total_plan_time` и `total_exec_time` в представлении `pgpro_stats_statements` или `pg_stat_statements`. Столбцы этой таблицы перечислены в Таблице G.21. Значения времени в ней выражаются в секундах.

Таблица G.21. Top SQL by elapsed time (SQL-запросы с наибольшей длительностью)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`%Total`	Отношение времени, затраченного на выполнение плана, к общему времени всех запросов в кластере	
`Elapsed Time (s)`	Общее время, затраченное на планирование и выполнение плана запроса	`total_plan_time` + `total_exec_time`
`Plan Time (s)`	Общее время, затраченное на планирование запроса	`total_plan_time`

Столбец	Описание	Поле/вычисление
`Exec Time (s)`	Общее время, затраченное на выполнение плана запроса	`total_exec_time`
`JIT Time (s)`	Общее время, затраченное на выполнение этого плана оператора с применением JIT, в секундах	`jit_generation_time` + `jit_inlining_time` + `jit_optimization_time` + `jit_emission_time`
`Read I/O time (s)`	Общее время, потраченное при выполнении запроса на чтение блоков	`blk_read_time`
`Write I/O time (s)`	Общее время, потраченное при выполнении запроса на запись блоков	`blk_write_time`
`Usr CPU time (s)`	Процессорное время, потраченное в пользовательском режиме, в секундах	`rusage.user_time`
`Sys CPU time (s)`	Процессорное время, потраченное в режиме ядра, в секундах	`rusage.system_time`
`Plans`	Сколько раз строился данный план запроса	`plans`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by planning time» показывает запросы с наибольшей длительностью планирования, определяемой по значению поля `total_plan_time` представления `pgpro_stats_statements` или `pg_stat_statements`. Столбцы этой таблицы перечислены в Таблице G.22.

Таблица G.22. Top SQL by planning time (SQL-запросы с наибольшей длительностью планирования)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`

Столбец	Описание	Поле/вычисление
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Plan elapsed(s)`	Общее время, затраченное на построение плана данного запроса, в секундах	`total_plan_time`
`%Elapsed`	Отношение `total_plan_time` к сумме `total_plan_time` и `total_exec_time` для данного плана	
`Mean plan time`	Среднее время планирования данного запроса, в миллисекундах	`mean_plan_time`
`Min plan time`	Минимальное время планирования данного запроса, в миллисекундах	`min_plan_time`
`Max plan time`	Максимальное время планирования данного запроса, в миллисекундах	`max_plan_time`
`StdErr plan time`	Стандартное отклонение времени, затраченного на планирование запроса, в миллисекундах	`stddev_plan_time`
`Plans`	Сколько раз строился данный план запроса	`plans`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by execution time» показывает запросы с наибольшей длительностью выполнения, определяемой по значению поля `total_time` представления `pgpro_stats_statements` или `pg_stat_statements`. Столбцы этой таблицы перечислены в Таблице G.23.

Таблица G.23. Top SQL by execution time (SQL-запросы с наибольшей длительностью выполнения)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Exec (s)`	Общее время, потраченное на выполнение плана запроса, в секундах	`total_exec_time`
`%Elapsed`	Процент времени `total_exec_time` , затраченного на выполнение плана, от времени выполнения данного оператора	
`%Total`	Отношение времени `total_exec_time` , затраченного на выполнение плана, к общему времени выполнения всех запросов в кластере	
`JIT Time (s)`	Общее время, затраченное на выполнение этого плана оператора с применением JIT, в секундах	`jit_generation_time` + `jit_inlining_time` + `jit_optimization_time` + `jit_emission_time`
`Read I/O time (s)`	Общее время, затраченное на чтение страниц при выполнении плана, в секундах	`blk_read_time`
`Write I/O time (s)`	Общее время, затраченное на запись страниц при выполнении плана, в секундах	`blk_write_time`
`Usr CPU time (s)`	Процессорное время, потраченное в пользовательском режиме, в секундах	`rusage.user_time`
`Sys CPU time (s)`	Процессорное время, потраченное в режиме ядра, в секундах	`rusage.system_time`
`Rows`	Число строк, полученных или обработанных при выполнении плана	`rows`
`Mean execution time`	Среднее время, потраченное на выполнение плана, в миллисекундах	`mean_exec_time`
`Min execution time`	Минимальное время, потраченное на выполнение плана, в миллисекундах	`min_exec_time`
`Max execution time`	Максимальное время, потраченное на выполнение плана, в миллисекундах	`max_exec_time`
`StdErr execution time`	Стандартное отклонение времени, затраченного на выполнение плана, в миллисекундах	`stddev_exec_time`
`Executions`	Сколько раз выполнялся план	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в	

Столбец	Описание	Поле/вычисление
	процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by mean execution time» показывает `pgpro_pwr.max` запросов с наибольшей средней длительностью выполнения, определяемой по значению поля `mean_time` или `mean_exec_time` представления `pgpro_stats_statements` или `pg_stat_statements`. Столбцы этой таблицы перечислены в Таблице G.24.

Таблица G.24. Top SQL by mean execution time (SQL-запросы с наибольшей средней длительностью выполнения)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Mean execution time (ms)`	Среднее время, потраченное на выполнение оператора, в миллисекундах	`mean_exec_time`
`Min execution time (ms)`	Минимальное время, потраченное на выполнение оператора, в миллисекундах	`min_exec_time`
`Max execution time (ms)`	Максимальное время, потраченное на выполнение оператора, в миллисекундах	`max_exec_time`
`StdErr execution time`	Стандартное отклонение времени, затраченного на выполнение оператора, в миллисекундах	`stddev_exec_time (ms)`
`Exec (s)`	Время, потраченное на выполнение этого оператора, в секундах	`total_exec_time`
`%Elapsed`	Время выполнения этого оператора в процентах от общего времени, затраченного на оператор	
`%Total`	Время выполнения этого оператора в процентах от общего времени, затраченного на все операторы в кластере	

Столбец	Описание	Поле/вычисление
`JIT time (s)`	Общее время, затраченное на выполнение этого оператора с применением JIT, в секундах	`jit_generation_time` + `jit_inlining_time` + `jit_optimization_time` + `jit_emission_time`
`Read I/O time (s)`	Время, затраченное на чтение блоков, в секундах	`blk_read_time`
`Write I/O time (s)`	Время, затраченное на запись блоков, в секундах	`blk_write_time`
`Rows`	Число строк, полученных или обработанных оператором	`rows`
`Executions`	Счётчик выполнений этого оператора	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by executions» показывает запросы, которые выполнялись чаще других, что определяется по значению поля `calls` представления `pgpro_stats_statements` или `pg_stat_statements`. Столбцы этой таблицы перечислены в Таблице G.25.

Таблица G.25. Top SQL by executions (Наиболее частые SQL-запросы)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Total`	Отношение числа выполнений этого запроса (`calls`) к сумме значений `calls` по всем запросам, выполненным в кластере	
`Rows`	Число строк, полученных или обработанных при выполнении плана	`rows`

Столбец	**Описание**	**Поле/вычисление**
`Mean (ms)`	Среднее время, потраченное на выполнение плана, в миллисекундах	`mean_exec_time`
`Min (ms)`	Минимальное время, потраченное на выполнение плана, в миллисекундах	`min_exec_time`
`Max (ms)`	Максимальное время, потраченное на выполнение плана, в миллисекундах	`max_exec_time`
`StdErr (ms)`	Стандартное отклонение времени, затраченного на выполнение плана, в миллисекундах	`stddev_time`
`Elapsed(s)`	Общее время, потраченное на выполнение плана запроса, в секундах	`total_exec_time`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by I/O wait time» показывает запросы с наибольшей длительностью операций чтения/записи, определяемой как сумма полей `blk_read_time` и `blk_write_time` представления `pgpro_stats_statements` или `pg_stat_statements`. Столбцы этой таблицы перечислены в Таблице G.26. Значения времени в ней выражаются в секундах.

Таблица G.26. Top SQL by I/O wait time (SQL-запросы с наибольшим временем I/O)

Столбец	**Описание**	**Поле/вычисление**
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`IO(s)`	Общее время, затраченное на чтение/запись при выполнении плана, то есть время ввода-вывода	`blk_read_time` + `blk_write_time`
`R(s)`	Общее время, затраченное на чтение при выполнении плана	`blk_read_time`

Столбец	Описание	Поле/вычисление
`W(s)`	Общее время, затраченное на запись при выполнении плана	`blk_write_time`
`%Total`	Отношение времени ввода-вывода при выполнении этого плана к общему времени ввода-вывода для всех запросов в кластере	
`Shr Reads`	Общее число разделяемых блоков, прочитанных при выполнении плана	`shared_blks_read`
`Loc Reads`	Общее число локальных блоков, прочитанных при выполнении плана	`local_blks_read`
`Tmp Reads`	Общее число временных блоков, прочитанных при выполнении плана	`temp_blks_read`
`Shr Writes`	Общее число разделяемых блоков, записанных при выполнении плана	`shared_blks_written`
`Loc Writes`	Общее число локальных блоков, записанных при выполнении плана	`local_blks_written`
`Tmp Writes`	Общее число временных блоков, записанных при выполнении плана	`temp_blks_written`
`Elapsed(s)`	Общее время, затраченное на выполнение плана запроса	`total_plan_time + total_exec_time`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by shared blocks fetched» показывает запросы с наибольшим количеством полученных (с диска или из кеша) блоков, что помогает выявить запросы, наиболее активно читающие данные. Столбцы этой таблицы перечислены в Таблице G.27.

Таблица G.27. Top SQL by shared blocks fetched (SQL-запросы, получившие максимум разделяемых блоков)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	

Столбец	**Описание**	**Поле/вычисление**
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Blks fetched`	Число блоков, полученных при выполнении этого плана запроса	`shared_blks_hit` + `shared_blks_read`
`%Total`	Отношение числа блоков, полученных при выполнении этого плана, к общему числу блоков, полученных при выполнении всех запросов в кластере	
`Hits(%)`	Отношение числа блоков, полученных из буферов, к общему числу полученных блоков	
`Elapsed(s)`	Общее время, затраченное на выполнение плана запроса, в секундах	`total_plan_time` + `total_exec_time`
`Rows`	Число строк, полученных или обработанных при выполнении плана	`rows`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by shared blocks read» показывает запросы с наибольшим количеством прочитанных разделяемых блоков, что помогает выявить запросы, наиболее активно читающие данные. Столбцы этой таблицы перечислены в Таблице G.28.

Таблица G.28. Top SQL by shared blocks read (SQL-запросы, прочитавшие максимум разделяемых блоков)

Столбец	**Описание**	**Поле/вычисление**
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`

Столбец	**Описание**	**Поле/вычисление**
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Reads`	Количество прочитанных разделяемых блоков при выполнении этого плана	`shared_blks_read`
`%Total`	Отношение количества разделяемых блоков, прочитанных для этого плана, к количеству разделяемых блоков, прочитанных всеми выполненными в кластере запросами	
`Hits(%)`	Отношение числа блоков, полученных из буферов, к общему числу блоков, полученных при выполнении этого плана	
`Elapsed(s)`	Общее время, затраченное на выполнение плана запроса, в секундах	`total_plan_time` + `total_exec_time`
`Rows`	Число строк, полученных или обработанных при выполнении плана	`rows`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by shared blocks dirtied» показывает запросы с наибольшим количеством загрязнённых разделяемых буферов, что помогает выявить запросы, наиболее активно меняющие данные. Столбцы этой таблицы перечислены в Таблице G.29.

Таблица G.29. Top SQL by shared blocks dirtied (SQL-запросы, «загрязнившие» максимум разделяемых блоков)

Столбец	**Описание**	**Поле/вычисление**
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`

Столбец	Описание	Поле/вычисление
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Dirtied`	Количество разделяемых блоков, загрязнённых при выполнении этого плана	`shared_blks_dirtied`
`%Total`	Отношение количества загрязнённых разделяемых буферов для этого плана к количеству разделяемых блоков, загрязнённых всеми запросами, выполненными в кластере	
`Hits(%)`	Отношение числа блоков, полученных из буферов, к общему числу блоков, полученных при выполнении этого плана	
`WAL`	Общий объём WAL (в байтах), сгенерированный при выполнении плана	`wal_bytes`
`%Total`	Отношение объёма WAL, сгенерированного при выполнении плана, ко всему объёму WAL, сгенерированному в кластере	
`Elapsed(s)`	Общее время, затраченное на выполнение плана запроса, в секундах	`total_plan_time` + `total_exec_time`
`Rows`	Число строк, полученных или обработанных при выполнении плана	`rows`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by shared blocks written» показывает запросы, записавшие наибольшее количество блоков. Столбцы этой таблицы перечислены в Таблице G.30.

Таблица G.30. Top SQL by shared blocks written (SQL-запросы, записавшие максимум разделяемых блоков)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызывае-	

Столбец	Описание	Поле/вычисление
	мые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Written`	Количество блоков, записанных при выполнении плана	`shared_blks_written`
`%Total`	Отношение числа блоков, записанных при выполнении этого плана, к общему числу записанных блоков в кластере	Отношение `shared_blks_written` к `pg_stat_bgwriter .buffers_checkpoint` + `pg_stat_bgwriter .buffers_clean` + `pg_stat_bgwriter .buffers_backend`
`%BackendW`	Отношение числа блоков, записанных при выполнении этого плана, к общему числу блоков, записанных обслуживающими процессами	Отношение `shared_blks_written` к `pg_stat_bgwriter .buffers_backend`
`Hits(%)`	Отношение числа блоков, полученных из буферов, к общему числу блоков, полученных при выполнении этого плана	
`Elapsed(s)`	Общее время, затраченное на выполнение плана запроса, в секундах	`total_plan_time` + `total_exec_time`
`Rows`	Число строк, полученных или обработанных при выполнении плана	`rows`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by WAL size» показывает запросы, которые породили наибольший объём записей WAL. Столбцы этой таблицы перечислены в Таблице G.31.

Таблица G.31. Top SQL by WAL size (SQL-запросы, породившие наибольший объём WAL)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приво-	

Столбец	Описание	Поле/вычисление
	дится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`WAL`	Общий объём WAL (в байтах), сгенерированный при выполнении плана	`wal_bytes`
`%Total`	Отношение объёма WAL, сгенерированного при выполнении плана, ко всему объёму WAL, сгенерированному в кластере	
`Dirtied`	Количество разделяемых блоков, загрязнённых при выполнении этого плана	`shared_blks_dirtied`
`WAL FPI`	Общее число образов полных страниц в WAL, сгенерированных при выполнении плана	`wal_fpi`
`WAL records`	Общее число записей WAL, сгенерированных при выполнении плана	`wal_records`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by temp usage» показывает запросы с наибольшим объёмом ввода-вывода временных блоков, которое считается как сумма полей `temp_blks_read`, `temp_blks_written`, `local_blks_read` и `local_blks_written`. Столбцы этой таблицы перечислены в Таблице G.32.

Таблица G.32. Top SQL by temp usage (SQL-запросы с максимальным использованием временных блоков)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	

Столбец	**Описание**	**Поле/вычисление**
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Local fetched`	Число полученных локальных блоков	`local_blks_hit` + `local_blks_read`
`Hits(%)`	Отношение числа локальных блоков, полученных из буферов, к общему числу полученных локальных блоков	
`Write Local (blk)`	Количество блоков, записанных при выполнении этого плана и относящихся к временным таблицам	`local_blks_written`
`Write Local %Total`	Отношение значения `local_blks_written` этого плана к сумме `local_blks_written` по всем запросам, выполненным в кластере	
`Read Local (blk)`	Количество блоков, прочитанных при выполнении этого плана и относящихся к временным таблицам	`local_blks_read`
`Read Local %Total`	Отношение значения `local_blks_read` этого плана к сумме `local_blks_written` по всем запросам, выполненным в кластере	
`Write Temp (blk)`	Количество временных блоков, записанных при выполнении плана	`temp_blks_written`
`Write Temp %Total`	Отношение значения `temp_blks_written` этого плана к сумме `temp_blks_written` по всем запросам, выполненным в кластере	
`Read Temp (blk)`	Количество временных блоков, прочитанных при выполнении плана	`temp_blks_read`
`Read Temp %Total`	Отношение значения `temp_blks_read` этого плана к сумме `temp_blks_read` по всем запросам, выполненным в кластере	
`Elapsed(s)`	Общее время, затраченное на выполнение плана запроса, в секундах	`total_plan_time` + `total_exec_time`

Столбец	Описание	Поле/вычисление
`Rows`	Число строк, полученных или обработанных при выполнении плана	`rows`
`Executions`	Сколько раз выполнялся план запроса	`calls`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

Таблица отчёта «Top SQL by invalidation messages sent» показывает операторы, для которых было отправлено наибольшее число событий аннулирования. Столбцы этой таблицы перечислены в Таблице G.33.

Таблица G.33. Top SQL by invalidation messages sent (Операторы с наибольшим числом отправленных событий аннулирования)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	`queryid`
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Invalidation messages sent`	Общее количество событий аннулирования, которые были отправлены обслуживающими процессами, выполняющими этот оператор. Статистика предоставляется для соответствующих типов сообщений `pgpro_stats_inval_msgs`	Поля столбца `pgpro_stats_statements .inval_msgs`

G.1.11.4.1. rusage statistics (Статистика использования ресурсов)

Этот раздел добавляется в отчёт, только если в отчётном интервале было доступно расширение pgpro_stats или pg_stat_kcache.

Таблица отчёта «Top SQL by system and user time» показывает запросы с наибольшей суммой значений полей `user_time` и `system_time` в представлении `pg_stat_kcache` или `pgpro_stats_totals`. Столбцы этой таблицы перечислены в Таблице G.34.

Таблица G.34. Top SQL by system and user time (SQL-запросы с наибольшей системной и пользовательской нагрузкой CPU)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Plan User (s)`	Время процессора в пользовательском режиме, затраченное на планирование, в секундах	`plan_user_time`
`Exec User (s)`	Время процессора в пользовательском режиме, затраченное на выполнение, в секундах	`exec_user_time`
`User %Total`	Отношение значения `plan_user_time` + `exec_user_time` к общему времени использования процессора в пользовательском режиме всеми запросами	
`Plan System (s)`	Время процессора в режиме ядра, затраченное на планирование, в секундах	`plan_system_time`
`Exec System (s)`	Время процессора в режиме ядра, затраченное на выполнение, в секундах	`exec_system_time`
`System %Total`	Отношение значения `plan_system_time` + `exec_system_time` к общему времени использования процессора в режиме ядра всеми запросами	

Таблица отчёта «Top SQL by reads/writes done by filesystem layer» показывает запросы с наибольшей суммой значений `reads` и `writes` представления `pg_stat_kcache`. Столбцы этой таблицы перечислены в Таблице G.35.

Таблица G.35. Top SQL by reads/writes done by filesystem layer (SQL-запросы, выполнившие максимум операций чтения/записи в файловой системе)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Plan Read Bytes`	Количество байт, прочитанное при планировании	`plan_reads`
`Exec Read Bytes`	Количество байт, прочитанное при выполнении	`exec_reads`
`Read Bytes %Total`	Отношение значения `plan_reads` + `exec_reads` к общему количеству байт, прочитанному на уровне файловой системы всеми запросами	
`Plan Writes`	Количество байт, записанное при планировании	`plan_writes`
`Exec Writes`	Количество байт, записанное при выполнении	`exec_writes`
`Write %Total`	Отношение значения `plan_writes` + `exec_writes` к общему количеству байт, прочитанному на уровне файловой системы всеми запросами	

G.1.11.5. SQL query wait statistics (Статистика ожидания по SQL-запросам)

Если в отчётном интервале было доступно расширение pgpro_stats, в этом разделе отчёта будет содержаться таблица, разделённая на секции, в каждой из которой будут показываться запросы с наибольшем временем ожидания в целом или с наибольшим временем ожидания определённого типа события. Разделы этой таблицы, относящиеся к определённым событиям ожидания, располагаются в порядке уменьшения общего времени ожидания событий данного типа. Столбцы этой таблицы перечислены в Таблице G.36. Значения времени в ней выражаются в секундах.

Таблица G.36. SQL query wait statistics (Статистика ожидания по SQL-запросам)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от иденти-	

Столбец	Описание	Поле/вычисление
	фикатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Хеш данного плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`
`Waited`	Общее время, затраченное на ожидание всех типов событий при выполнении этого плана	
`%Total`	Отношение суммарного времени ожидания при выполнении этого плана к общему времени ожидания для всех запросов в кластере	
`Details`	Детализация событий ожидания по типам	

Если расширение, собирающее статистику операторов в отчётном интервале, собрало статистику JIT, в отчёте выводится таблица «Top SQL by JIT elapsed time», где показаны основные операторы по сумме значений полей `jit_*_time` представления `pgpro_stats_statements` или `pg_stat_statements`. Столбцы этой таблицы перечислены в Таблице G.37. Значения времени в ней выражаются в секундах.

Таблица G.37. Top SQL by JIT elapsed time (SQL-запросы с наибольшей длительностью JIT)

Столбец	Описание	Поле/вычисление
`Query ID`	Шестнадцатеричное представление `queryid`. Хеш от идентификатора запроса, идентификатора базы данных и идентификатора пользователя приводится в квадратных скобках. Для вложенных операторов (таких как операторы, вызываемые внутри операторов верхнего уровня) здесь будет отображаться метка `(N)` .	
`Plan ID`	Внутренний хеш-код, вычисленный для дерева плана запроса	`planid`
`Database`	Имя базы данных, в которой выполнялся запрос	Выводится из `dbid`
`User`	Имя пользователя, выполняющего запрос	Выводится из `userid`

Столбец	**Описание**	**Поле/вычисление**
`JIT Time (s)`	Общее время, затраченное на выполнение этого плана оператора с применением JIT	`jit_generation_time` + `jit_inlining_time` + `jit_optimization_time` + `jit_emission_time`
`Generation count`	Общее число функций, скомпилированных в JIT-код при выполнении данного оператора	Сумма значений `jit_functions`
`Generation time`	Общее время, затраченное на компиляцию JIT-кода при выполнении оператора	Сумма значений `jit_generation_time`
`Inlining count`	Сколько раз встраивались функции	Сумма значений `jit_inlining_count`
`Inlining time`	Общее время, затраченное на встраивание функций при выполнении оператора	Сумма значений `jit_inlining_time`
`Optimization count`	Число JIT-оптимизаций для данного оператора	Сумма значений `jit_optimization_count`
`Optimization time`	Общее время, затраченное на JIT-оптимизацию при выполнении данного оператора	Сумма значений `jit_optimization_time`
`Emission count`	Сколько раз выдавался код	Сумма значений `jit_emission_count`
`Emission time`	Общее время, затраченное на выдачу кода при выполнении оператора	Сумма значений `jit_emission_time`
`Deform count`	Число функций преобразования кортежей, скомпилированных в JIT-код при выполнении оператора	
`Deform time`	Общее время, затраченное оператором на компилирование преобразования кортежей в JIT-код	
`Plan Time (s)`	Общее время, затраченное на планирование запроса	`total_plan_time`
`Exec Time (s)`	Общее время, затраченное на выполнение плана запроса	`total_exec_time`
`Read I/O time (s)`	Общее время, потраченное при выполнении запроса на чтение блоков	`blk_read_time`
`Write I/O time (s)`	Общее время, потраченное при выполнении запроса на запись блоков	`blk_write_time`
`%Cvr`	Охват: продолжительность сбора статистики по операторам в процентах от продолжительности отчётного интервала	

G.1.11.6. Complete list of SQL texts (Полный текст SQL-запросов)

Раздел отчёта «Complete list of SQL texts» содержит таблицу с полным текстом и планом запроса для всех SQL-операторов, упомянутых в отчёте. Перейти к тексту соответствующего запроса/плана, можно из любой таблицы со статистикой по ссылке `Query ID`/`Plan ID`. Столбцы этой таблицы перечислены в Таблице G.38.

Таблица G.38. Complete list of SQL texts (Полный текст SQL-запросов)

Столбец	Описание
`ID`	Шестнадцатеричное представление идентификатора запроса или плана
`Query/Plan Text`	Текст или план запроса

G.1.11.7. Schema object statistics (Статистика по объектам схемы)

Таблицы в этом разделе отчёта показывают наиболее активно используемые объекты базы согласно информации Сборщика статистики в Postgres Pro. Таблицы отчётов, содержащие данные по таблицам и индексам, позволяют просматривать параметры хранения. Щелчок мышью по соответствующей строке показывает параметры хранения непосредственно под строкой.

Таблица отчёта «Top tables by estimated sequentially scanned volume» показывает таблицы с наибольшим приблизительным объёмом, просканированным последовательным образом. Исходя из этого, можно понять, что для каких-то таблиц не хватает индексов. В отсутствие данных о размере, собираемых функцией `pg_relation_size()`, оценка размера берётся из поля `pg_class.relpages`. Для обозначения меньшей точности оценки она отображается в квадратных скобках. Данная информация основана на представлении `pg_stat_all_tables`. Столбцы этой таблицы перечислены в Таблице G.39.

Таблица G.39. Top tables by estimated sequentially scanned volume (Таблицы с наибольшим объёмом последовательно сканированных блоков)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`~SeqBytes`	Приблизительный объём, прочитанный при последовательном сканировании	Сумма (`pg_relation_size()` * `seq_scan`)
`SeqScan`	Число операций последовательного сканирования, выполненных в таблице	`seq_scan`
`IxScan`	Число операций сканирования по индексу, выполненных в этой таблице	`idx_scan`
`IxFet`	Количество «живых» строк, отобранных при сканированиях по индексу	`idx_tup_fetch`
`Ins`	Количество вставленных строк	`n_tup_ins`
`Upd`	Количество изменённых строк	`n_tup_upd`
`Del`	Количество удалённых строк	`n_tup_del`

Столбец	**Описание**	**Поле/вычисление**
`Upd(HOT)`	Количество строк, изменённых по схеме HOT	`n_tup_hot_upd`

В таблице отчёта «Top tables by blocks fetched» полученными блоками считаются блоки, как прочитанные с диска (read), так и найденные в общих буферах (hit). В ней показываются таблицы с максимальным суммарным количеством блоков, полученных из кучи, из индексов, из таблицы TOAST (при наличии) и индекса таблицы TOAST (при наличии). Это помогает понять, какие таблицы активнее других используют блоки данных. Эта информация основана на представлении `pg_statio_all_tables`. Столбцы этой таблицы перечислены в Таблице G.40.

Таблица G.40. Top tables by blocks fetched (Таблицы, для которых получено максимум блоков)

Столбец	**Описание**	**Поле/вычисление**
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Heap Blks`	Число блоков, полученных из кучи таблицы	`heap_blks_read + heap_blks_hit`
`Heap Blks %Total`	Отношение числа блоков, полученных из кучи таблицы, к общему числу блоков, полученных запросами в кластере	
`Ix Blks`	Число блоков, полученных из индексов таблицы	`idx_blks_read + idx_blks_hit`
`Ix Blks %Total`	Отношение числа блоков, полученных из индексов таблицы, к общему числу блоков, полученных запросами в кластере	
`TOAST Blks`	Число блоков, полученных из связанной таблицы TOAST	`toast_blks_read + toast_blks_hit`
`TOAST Blks %Total`	Отношение числа блоков, полученных из связанной таблицы TOAST, к общему числу блоков, полученных запросами в кластере	
`TOAST-Ix Blks`	Число блоков, полученных из индекса связанной таблицы TOAST	`tidx_blks_read + tidx_blks_hit`
`TOAST-Ix Blks %Total`	Отношение числа блоков, полученных из индекса связанной таблицы TOAST, к общему числу блоков, полученных запросами в кластере	

В таблице отчёта «Top tables by blocks read» показаны таблицы с максимальным суммарным количество блоков прочитанных из кучи, из индексов, из таблицы TOAST (при наличии) и индекса таблицы TOAST (при наличии). Это помогает понять, какие таблицы активнее других читают блоки данных с диска. Эта информация основана на представлении `pg_statio_all_tables`. Столбцы этой таблицы перечислены в Таблице G.41.

Таблица G.41. Top tables by blocks read (Таблицы, для которых прочитано максимум блоков)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Heap Blks`	Число блоков, прочитанных из кучи таблицы	`heap_blks_read`
`Heap Blks %Total`	Отношение числа блоков, прочитанных из кучи таблицы, к общему числу блоков, прочитанных запросами в кластере	
`Ix Blks`	Число блоков, прочитанных из индексов таблицы	`idx_blks_read`
`Ix Blks %Total`	Отношение числа блоков, прочитанных из индексов таблицы, к общему числу блоков, прочитанных запросами в кластере	
`TOAST Blks`	Число блоков, прочитанных из таблицы TOAST, относящейся к данной	`toast_blks_read`
`TOAST Blks %Total`	Отношение числа блоков, прочитанных из связанной таблицы TOAST, к общему числу блоков, прочитанных запросами в кластере	
`TOAST-Ix Blks`	Число блоков, прочитанных из индекса таблицы TOAST, относящейся к данной	`tidx_blks_read`
`TOAST-Ix Blks %Total`	Отношение числа блоков, прочитанных из индекса связанной таблицы TOAST, к общему числу блоков, прочитанных запросами в кластере	
`Hit(%)`	Отношение числа блоков таблицы, индекса, TOAST-таблицы и индекса TOAST-таблицы, полученных из буферов для данной таблицы, к общему числу блоков, полученных для этой таблицы либо из файловой системы, либо из буферов	

В таблице отчёта «Top DML tables» показаны таблицы с наибольшим числом строк, затронутых операциями DML, то есть с наибольшей суммой значений `n_tup_ins`, `n_tup_upd` и `n_tup_del` (включая таблицы TOAST). Эта информация основана на представлении `pg_stat_all_tables`. Столбцы этой таблицы перечислены в Таблице G.42.

Таблица G.42. Top DML tables (Таблицы с наибольшим объёмом DML-изменений)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Ins`	Количество вставленных строк	`n_tup_ins`
`Upd`	Количество изменённых строк, включая изменения по схеме HOT	`n_tup_upd`
`Del`	Количество удалённых строк	`n_tup_del`
`Upd(HOT)`	Количество строк, изменённых по схеме HOT	`n_tup_hot_upd`
`SeqScan`	Число операций последовательного сканирования, выполненных в таблице	`seq_scan`
`SeqFet`	Количество «живых» строк, прочитанных при последовательных чтениях	`seq_tup_read`
`IxScan`	Количество сканирований по индексу, запущенных по этой таблице	`idx_scan`
`IxFet`	Количество «живых» строк, отобранных при сканированиях по индексу	`idx_tup_fetch`

В таблице отчёта «Top tables by updated/deleted tuples» показаны таблицы с наибольшим количеством кортежей, затронутых операциями UPDATE/DELETE, то есть с наибольшей суммой значений `n_tup_upd` и `n_tup_del` (включая таблицы TOAST). Эта информация основана на представлении `pg_stat_all_tables`. Столбцы этой таблицы перечислены в Таблице G.43.

Таблица G.43. Top tables by updated/deleted tuples (Таблицы с наибольшим количеством изменённых/удалённых кортежей)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Upd`	Количество изменённых строк, включая изменения по схеме HOT	`n_tup_upd`

Столбец	Описание	Поле/вычисление
`Upd(HOT)`	Количество строк, изменённых по схеме HOT	`n_tup_hot_upd`
`Del`	Количество удалённых строк	`n_tup_del`
`Vacuum`	Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`
`AutoVacuum`	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`
`Analyze`	Сколько раз сбор статистики для этой таблицы был выполнен вручную	`analyze_count`
`AutoAnalyze`	Сколько раз сбор статистики для этой таблицы был выполнен фоновым процессом автоочистки	`autoanalyze_count`

Таблица отчёта «Top tables by removed all-visible marks» показывает таблицы с наибольшим количеством меток полной видимости, удалённых обслуживающими процессами из карты видимости. Этот раздел отчёта отображается только при наличии соответствующей статистики. В Таблице G.44 перечислены столбцы этой таблицы отчёта.

Таблица G.44. Top tables by removed all-visible marks (Таблицы с наибольшим количеством удалённых меток полной видимости)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`All-Visible marks cleared`	Общее число меток полной видимости, удалённых из карты видимости отношения	`rev_all_visible_pages`
`All-Visible marks set`	Общее число меток полной видимости, установленных в карте видимости отношения	`pages_all_visible`
`All-Visible marks %Set`	Отношение числа установленных меток полной видимости к общему числу установленных и удалённых меток полной видимости	`pages_all_visible` * 100% / (`rev_all_visible_pages` + `pages_all_visible`)
`Vacuum`	Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`
`AutoVacuum`	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`

Таблица отчёта «Top tables by removed all-frozen marks» показывает таблицы с наибольшим количеством меток полной заморозки, удалённых обслуживающими процессами из карты видимости. Этот раздел отчёта отображается только при наличии соответствующей статистики. В Таблице G.45 перечислены столбцы этой таблицы отчёта.

Таблица G.45. Top tables by removed all-visible marks (Таблицы с наибольшим количеством удалённых меток полной заморозки)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`All-Frozen marks cleared`	Общее число меток полной заморозки, удалённых из карты видимости отношения	`rev_all_frozen_pages`
`All-Frozen marks set`	Общее число меток полной заморозки, установленных в карте видимости отношения	`pages_frozen`
`All-Frozen marks %Set`	Отношение числа установленных меток полной заморозки к общему числу установленных и удалённых меток полной заморозки	`pages_frozen` * 100% / (`rev_all_frozen_pages` + `pages_frozen`)
`Vacuum`	Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`
`AutoVacuum`	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`

Таблица отчёта «Top tables by new-page updated tuples» (Таблицы с наибольшим количеством изменённых кортежей, попавших на новую страницу) показывает таблицы с наибольшим количеством изменённых строк, новая версия которых переходит на *новую* страницу кучи, оставляя исходную версию с полем `t_ctid`, которое указывает на другую страницу кучи. Учитываются только изменения не по схеме HOT. Столбцы этой таблицы перечислены в Таблице G.46.

Таблица G.46. Top tables by new-page updated tuples (Таблицы с наибольшим количеством изменённых кортежей, попавших на новую страницу)

Столбец	Описание
`DB`	Имя базы данных, к которой относится таблица
`Tablespace`	Имя табличного пространства, в котором находится таблица
`Schema`	Имя схемы, содержащей таблицу
`Table`	Имя таблицы
`NP Upd`	Количество изменённых строк, попавших на новую страницу кучи
`%Upd`	Количество изменённых строк, попавших на новую страницу, в процентах от количества всех изменённых строк

Столбец	Описание
`Upd`	Количество изменённых строк, включая изменения по схеме HOT
`Upd(HOT)`	Количество строк, изменённых по схеме HOT (т. е. без отдельного изменения индекса)

Таблица отчёта «Top growing tables» показывает таблицы, которые увеличились в объёме больше других. Эта информация основана на представлении `pg_stat_all_tables`. В отсутствие данных о размере, собираемых функцией `pg_relation_size()`, оценка размера берётся из поля `pg_class.relpages`. Для обозначения меньшей точности оценки она отображается в квадратных скобках. Столбцы этой таблицы перечислены в Таблице G.47.

Таблица G.47. Top growing tables (Наиболее быстро растущие таблицы)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Size`	Размер таблицы в момент получения последней выборки в отчётном интервале	`pg_table_size()` - `pg_relation_size(toast)`
`Growth`	Увеличение размера таблицы	
`Ins`	Количество вставленных строк	`n_tup_ins`
`Upd`	Количество изменённых строк, включая изменения по схеме HOT	`n_tup_upd`
`Del`	Количество удалённых строк	`n_tup_del`
`Upd(HOT)`	Количество строк, изменённых по схеме HOT	`n_tup_hot_upd`

В таблице отчёта «Top indexes by blocks fetched» полученными блоками считаются блоки как прочитанные с диска (read), так и найденные в общих буферах (hit). Эта информация основана на представлении `pg_statio_all_indexes`. Столбцы этой таблицы перечислены в Таблице G.48.

Таблица G.48. Top indexes by blocks fetched (Индексы, из которых получено максимум блоков)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится индекс	
`Tablespace`	Имя табличного пространства, в котором находится индекс	
`Schema`	Имя схемы, содержащей нижележащую таблицу	
`Table`	Имя таблицы, для которой создан индекс	
`Index`	Имя индекса	
`Scans`	Количество произведённых сканирований по этому индексу	`idx_scan`

Столбец	Описание	Поле/вычисление
`Blks`	Число блоков, полученных из индекса	`idx_blks_read` + `idx_blks_hit`
`%Total`	Отношение числа блоков, полученных из индекса, к общему числу блоков, полученных запросами в кластере	

Таблица отчёта «Top indexes by blocks read» также основана на представлениях `pg_statio_all_indexes` и `pg_stat_all_indexes`. Столбцы этой таблицы перечислены в Таблице G.49.

Таблица G.49. Top indexes by blocks fetched (Индексы, из которых прочитано максимум блоков)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится индекс	
`Tablespace`	Имя табличного пространства, в котором находится индекс	
`Schema`	Имя схемы, содержащей нижележащую таблицу	
`Table`	Имя таблицы, для которой создан индекс	
`Index`	Имя индекса	
`Scans`	Количество произведённых сканирований по этому индексу	`idx_scan`
`Blk Reads`	Количество дисковых блоков, прочитанных из этого индекса	`idx_blks_read`
`%Total`	Отношение числа дисковых блоков, прочитанных из этого индекса, к общему числу блоков, прочитанных с диска запросами в кластере	
`Hits(%)`	Отношение числа блоков индекса, полученных из буферов, к общему числу блоков, полученных для этого индекса	

Таблица отчёта «Top growing indexes» показывает индексы, которые увеличились в объёме больше других. Эта информация основана на представлениях `pg_stat_all_tables` и `pg_stat_all_indexes`. В отсутствие данных о размере, собираемых функцией `pg_relation_size()`, оценка размера берётся из поля `pg_class.relpages`. Для обозначения меньшей точности оценки она отображается в квадратных скобках. Столбцы этой таблицы перечислены в Таблице G.50.

Таблица G.50. Top growing tables (Наиболее быстро растущие индексы)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится индекс	
`Tablespace`	Имя табличного пространства, в котором находится индекс	

Столбец	Описание	Поле/вычисление
`Schema`	Имя схемы, содержащей нижележащую таблицу	
`Table`	Имя таблицы, для которой создан индекс	
`Index`	Имя индекса	
`Index Size`	Размер индекса в момент получения последней выборки в отчётном интервале	`pg_relation_size()`
`Index Growth`	Прирост объёма индекса за отчётный интервал	
`Table Ins`	Количество строк, вставленных в нижележащую таблицу	`n_tup_ins`
`Table Upd`	Количество строк, изменённых в нижележащей таблице	`n_tup_upd` - `n_tup_hot_upd`
`Table Del`	Количество строк, удалённых из нижележащей таблицы	`n_tup_del`

Таблица отчёта «Unused indexes» показывает индексы, в нижележащих таблицах которых за отчётный интервал произведён наибольший объём изменений (требующих поддержания индекса), но при этом сами эти индексы не использовались. Индексы ограничений при этом не учитываются. Эта информация основана на представлении `pg_stat_all_tables`. Столбцы этой таблицы перечислены в Таблице G.51.

Таблица G.51. Unused indexes (Неиспользуемые индексы)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится индекс	
`Tablespace`	Имя табличного пространства, в котором находится индекс	
`Schema`	Имя схемы, содержащей нижележащую таблицу	
`Table`	Имя таблицы, для которой создан индекс	
`Index`	Имя индекса	
`Index Size`	Размер индекса в момент получения последней выборки в отчётном интервале	`pg_relation_size()`
`Index Growth`	Прирост объёма индекса за отчётный интервал	
`Table Ins`	Количество строк, вставленных в нижележащую таблицу	`n_tup_ins`
`Table Upd`	Количество строк, изменённых в нижележащей таблице	`n_tup_upd` - `n_tup_hot_upd`
`Table Del`	Количество строк, удалённых из нижележащей таблицы	`n_tup_del`

G.1.11.8. User function statistics (Статистика пользовательских функций)

Таблицы в этом разделе отчёта показывают функции, выделяющиеся по разным показателям, получаемым из представления `pg_stat_user_functions`. Значения времени в этих таблицах выражаются в секундах.

Таблица отчёта «Top functions by total time» показывает функции с наибольшей суммарной длительностью, таблица «Top functions by executions» — функции, выполняемые чаще других, а таблица «Top trigger functions by total time» — триггерные функции с наибольшей суммарной длительностью. Столбцы этих таблиц перечислены в Таблице G.52.

Таблица G.52. User function statistics (Статистика пользовательских функций)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, содержащей функцию	
`Schema`	Имя схемы, содержащей функцию	
`Function`	Имя функции	
`Executions`	Сколько раз вызывалась функция	`calls`
`Total Time (s)`	Общее время, затраченное на выполнение этой функции и всех других функций, вызванных ею	`total_time`
`Self Time (s)`	Общее время, затраченное на выполнение собственно функции, без учёта других функций, которые были ею вызваны	`self_time`
`Mean Time (s)`	Среднее время выполнения функции	`total_time /calls`
`Mean self Time (s)`	Среднее время выполнения собственно функции	`self_time /calls`

G.1.11.9. Vacuum-related statistics (Статистика, связанная с очисткой)

Таблица отчёта «Top tables by vacuum time spent» выводится, если pgpro_stats может предоставлять расширенную статистику очистки. Данная таблица показывает таблицы, на очистку которых было потрачено больше всего времени. Эта информация основана на представлении `pgpro_stats_vacuum_tables`. Столбцы этой таблицы перечислены в Таблице G.53.

Таблица G.53. Top tables by vacuum time spent (Таблицы, на очистку которых затрачено больше всего времени)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Total vacuum time`	Общее время, затраченное на очистку этой таблицы	`total_time`

Столбец	Описание	Поле/вычисление
`Delay vacuum time`	Время простоя в точке задержки при выполнении операций очистки для этой таблицы	`delay_time`
`Read I/O time`	Время, затраченное операциями очистки для этой таблицы на чтение блоков БД	`blk_read_time`
`Write I/O time`	Время, затраченное операциями очистки для этой таблицы на запись блоков БД	`blk_write_time`
`User CPU time`	Время использования процессора в пользовательском режиме при очистке таблиц этой БД	`user_time`
`System CPU time`	Время использования процессора в режиме ядра при очистке таблиц этой БД	`system_time`
`Vacuum count`	Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`
`Autovacuum count`	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`
`Total fetched`	Общее количество блоков БД, полученных операциями очистки, выполненными для этой таблицы	`total_blks_read` + `total_blks_hit`
`Heap fetched`	Общее количество блоков, полученных из этой таблицы операциями очистки, выполненными для неё	`rel_blks_read` + `rel_blks_hit`
`Scanned`	Количество страниц, к которым обращались операции очистки, выполненные для этой таблицы	`pages_scanned`

Таблица отчёта «Top indexes by vacuum time spent» выводится, если pgpro_stats может предоставлять расширенную статистику очистки. Данная таблица показывает индексы, на очистку которых было потрачено больше всего времени. Эта информация основана на представлении `pgpro_stats_vacuum_indexes`. Столбцы этой таблицы перечислены в Таблице G.54.

Таблица G.54. Top indexes by vacuum time spent (Индексы, на очистку которых затрачено больше всего времени)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Index`	Имя индекса	

Столбец	Описание	Поле/вычисление
`Total vacuum time`	Общее время, затраченное на очистку этого индекса	`total_time`
`Delay vacuum time`	Время простоя в точке задержки при выполнении операций очистки для этого индекса	`delay_time`
`Read I/O time`	Время, затраченное операциями очистки для этого индекса на чтение блоков БД	`blk_read_time`
`Write I/O time`	Время, затраченное операциями очистки для этого индекса на запись блоков БД	`blk_write_time`
`User CPU time`	Время использования процессора в пользовательском режиме при очистке этого индекса	`user_time`
`System CPU time`	Время использования процессора в режиме ядра при очистке этого индекса	`system_time`
`Vacuum count`	Сколько раз очистка нижележащей таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`
`Autovacuum count`	Сколько раз очистка нижележащей таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`
`Total fetched`	Общее количество блоков БД, полученных операциями очистки, выполненными для нижележащей таблицы	`total_blks_read + total_blks_hit`
`Index fetched`	Общее количество блоков, прочитанных из этого индекса операциями очистки, выполненными для него	`rel_blks_read + rel_blks_hit`

Таблица отчёта «Top tables by blocks vacuum fetched» выводится, если pgpro_stats может предоставлять расширенную статистику очистки. Данная таблица показывает таблицы, при очистке которых было получено наибольшее количество блоков. Эта информация основана на представлении `pgpro_stats_vacuum_tables`. Столбцы этой таблицы перечислены в Таблице G.55.

Таблица G.55. Top tables by blocks vacuum fetched (Таблицы с наибольшим количеством блоков, полученных при очистке)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`DB fetched`	Общее количество блоков БД, полученных операциями очист-	`total_blks_read + total_blks_hit`

Столбец	Описание	Поле/вычисление
	ки, выполненными для этой таблицы	
`%Total`	Общее количество блоков БД, которые были получены операциями очистки, выполненными для этой таблицы, в процентах от общего числа блоков, полученных в кластере	
`DB read`	Общее количество блоков БД, прочитанных операциями очистки для этой таблицы	`total_blks_read`
`%Total`	Общее количество блоков БД, которые были прочитаны операциями очистки, выполненными для этой таблицы, в процентах от общего числа блоков, прочитанных в кластере	
`Heap fetched`	Общее количество блоков, полученных из этой таблицы операциями очистки, выполненными для неё	`rel_blks_read` + `rel_blks_hit`
`%Rel`	Общее количество блоков таблицы, которые были прочитаны операциями очистки, выполненными для этой таблицы, в процентах от общего числа блоков, полученных из этой таблицы	
`Heap read`	Общее количество блоков, которые были прочитаны из этой таблицы операциями очистки, выполненными для неё	`rel_blks_read`
`%Rel`	Общее число блоков таблицы, прочитанных операциями очистки, выполненными в этой таблице, в процентах от общего числа блоков, прочитанных из этой таблицы	
`Scanned`	Количество страниц, к которым обращались операции очистки, выполненные для этой таблицы	`pages_scanned`

Таблица отчёта «Top indexes by blocks vacuum fetched» выводится, если pgpro_stats может предоставлять расширенную статистику очистки. Данная таблица показывает индексы с наибольшим количеством блоков, полученных при очистке нижележащих таблиц. Эта информация основана на представлении `pgpro_stats_vacuum_tables`. Столбцы этой таблицы перечислены в Таблице G.56.

Таблица G.56. Top indexes by blocks vacuum fetched (Индексы с наибольшим количеством блоков, полученных при очистке)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	

Столбец	Описание	Поле/вычисление
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Index`	Имя индекса	
`DB fetched`	Общее количество блоков БД, которые были получены операциями очистки, выполненными для этого индекса	`total_blks_read` + `total_blks_hit`
`%Total`	Общее количество блоков БД, которые были получены операциями очистки, выполненными для этого индекса, в процентах от общего числа блоков, полученных в кластере	
`DB read`	Общее количество блоков БД, прочитанных операциями очистки для этого индекса	`total_blks_read`
`%Total`	Общее количество блоков БД, которые были прочитаны операциями очистки, выполненными для этого индекса, в процентах от общего числа блоков, прочитанных в кластере	
`Idx fetched`	Общее количество блоков, которые были прочитаны из этого индекса операциями очистки, выполненными для него	`rel_blks_read` + `rel_blks_hit`
`%Idx`	Общее число блоков индекса, полученных операциями очистки, выполненными для этого индекса, в процентах от общего числа блоков, полученных из этого индекса	
`Idx read`	Общее количество блоков, которые были прочитаны из этого индекса операциями очистки, выполненными для него	`rel_blks_read`
`%Idx`	Общее число блоков индекса, прочитанных операциями очистки, выполненными для этого индекса, в процентах от общего числа блоков, прочитанных из этого индекса	

Таблица отчёта «Top tables by blocks vacuum read» выводится, если pgpro_stats может предоставлять расширенную статистику очистки. Данная таблица показывает таблицы с наибольшим количеством блоков, прочитанных при очистке этих таблиц. Эта информация основана на представлении `pgpro_stats_vacuum_tables`. Столбцы этой таблицы перечислены в Таблице G.57.

Таблица G.57. Top tables by blocks vacuum read (Таблицы с наибольшим количеством блоков, прочитанных при очистке)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`DB read`	Общее количество блоков БД, прочитанных операциями очистки для этой таблицы	`total_blks_read`
`%Total`	Общее количество блоков БД, которые были прочитаны операциями очистки, выполненными для этой таблицы, в процентах от общего числа блоков, прочитанных в кластере	
`%Hit`	Общее количество блоков БД, найденных операциями очистки для этой таблицы в общем буфере, в процентах от общего числа блоков, полученных операциями очистки для этой таблицы	
`Heap read`	Общее количество блоков БД, прочитанных из этой таблицы операциями очистки	`rel_blks_read`
`%Rel`	Общее число блоков таблицы, прочитанных операциями очистки, выполненными в этой таблице, в процентах от общего числа блоков, прочитанных из этой таблицы	
`%Hit`	Общее число блоков таблицы, найденных операциями очистки для этой таблицы в общем буфере, в процентах от общего числа блоков таблицы, полученных операциями очистки для этой таблицы	
`Scanned`	Количество страниц, к которым обращались операции очистки, выполненные для этой таблицы	`pages_scanned`

Таблица отчёта «Top indexes by blocks vacuum read» выводится, если pgpro_stats может предоставлять расширенную статистику очистки. Данная таблица показывает индексы с наибольшим количеством блоков, полученных при очистке нижележащих таблиц. Эта информация основана на представлении `pgpro_stats_vacuum_indexes`. Столбцы этой таблицы перечислены в Таблице G.58.

Таблица G.58. Top indexes by blocks vacuum read (Индексы с наибольшим количеством блоков, прочитанных при очистке)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Index`	Имя индекса	
`DB read`	Общее количество блоков БД, прочитанных операциями очистки для этого индекса	`total_blks_read`
`%Total`	Общее количество блоков БД, которые были прочитаны операциями очистки, выполненными для этого индекса, в процентах от общего числа блоков, прочитанных в кластере	
`%Hit`	Общее количество блоков БД, найденных операциями очистки для этого индекса в общем буфере, в процентах от общего числа блоков БД, полученных операциями очистки для этого индекса	
`Idx read`	Общее количество блоков, которые были прочитаны из этого индекса операциями очистки, выполненными для него	`rel_blks_read`
`%Idx`	Общее число блоков индекса, прочитанных операциями очистки, выполненными для этого индекса, в процентах от общего числа блоков, прочитанных из этого индекса	
`%Hit`	Общее число блоков индекса, найденных операциями очистки для этого индекса в общем буфере, в процентах от общего числа блоков индекса, полученных операциями очистки для этого индекса	

Таблица отчёта «Top tables by dead tuples vacuum left» выводится, если pgpro_stats может предоставлять расширенную статистику очистки. Данная таблица показывает таблицы с наибольшим количеством «мёртвых» кортежей, оставленных при очистке из-за их видимости в транзакциях. Эта информация основана на представлении `pgpro_stats_vacuum_tables`. Столбцы этой таблицы перечислены в Таблице G.59.

Таблица G.59. Top tables by dead tuples vacuum left (Таблицы с наибольшим количеством оставленных «мёртвых» кортежей)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Dead tuples left`	Общее количество «мёртвых» кортежей, оставленных операциями очистки в этой таблице из-за видимости этих кортежей в транзакциях	`dead_tuples`
`Dead tuples deleted`	Общее количество «мёртвых» кортежей, удалённых операциями очистки из этой таблицы	`tuples_deleted`
`%Eff`	Эффективность очистки, оцениваемая по количеству удалённых кортежей. Это процент «мёртвых» кортежей, удалённых из этой таблицы, от всех «мёртвых» кортежей, подлежащих удалению из этой таблицы.	`tuples_deleted` * 100 / (`tuples_deleted` + `dead_tuples`)
`Tuples del`	Количество удалённых строк	`pg_stat_all_tables .n_tup_del`
`Tuples upd`	Количество изменённых строк (включая изменения по схеме HOT)	`pg_stat_all_tables .n_tup_upd`
`Vacuum`	Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`
`Autovacuum`	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`

Таблица отчёта «Top tables by WAL size generated by vacuum» выводится, если pgpro_stats может предоставлять расширенную статистику очистки. Данная таблица показывает таблицы с наибольшим объёмом WAL, сгенерированным операциями очистки, выполненными для этих таблиц. Эта информация основана на представлении `pgpro_stats_vacuum_tables`. Столбцы этой таблицы перечислены в Таблице G.60.

Таблица G.60. Top tables by WAL size generated by vacuum (Таблицы с наибольшим объёмом WAL, сгенерированным операциями очистки)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	

Столбец	**Описание**	**Поле/вычисление**
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`WAL size`	Общий объём WAL (в байтах), сгенерированный операциями очистки для этой таблицы	`wal_bytes`
`%Total`	Общий объём WAL (в байтах), сгенерированный операциями очистки для этой таблицы, в процентах от общего объёма WAL, сгенерированного в кластере	
`WAL FPI`	Общее число образов полных страниц в WAL, сгенерированных операциями очистки для этой таблицы	`wal_fpi`
`Scanned blocks`	Количество страниц, к которым обращались операции очистки, выполненные для этой таблицы	`pages_scanned`
`Dirtied blocks`	Количество блоков БД, загрязнённых операциями очистки для этой таблицы	`total_blks_dirtied`
`Removed blocks`	Количество страниц, удалённых операциями очистки для этой таблицы	`pages_removed`
`Vacuum`	Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`
`Autovacuum`	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`

В таблице отчёта «Top tables by vacuum operations» показаны таблицы, с которыми произведено наибольшее количество операций очистки, что определяется суммой значений `vacuum_count` + `autovacuum_count`. Эта информация основана на представлении `pg_stat_all_tables`. Столбцы этой таблицы перечислены в Таблице G.61.

Таблица G.61. Top tables by vacuum operations (Таблицы, наиболее часто подвергающиеся очистке)

Столбец	**Описание**	**Поле/вычисление**
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Vacuum count`	Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`

Столбец	**Описание**	**Поле/вычисление**
`Autovacuum count`	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`
`Ins`	Количество вставленных строк	`n_tup_ins`
`Upd`	Количество изменённых строк (включая изменения по схеме HOT)	`n_tup_upd`
`Del`	Количество удалённых строк	`n_tup_del`
`Upd(HOT)`	Количество строк, изменённых по схеме HOT	`n_tup_hot_upd`

В таблице отчёта «Top tables by analyze operations» показаны таблицы, для которых было произведено наибольшее количество операций анализа, что определяется суммой значений `analyze_count` + `autoanalyze_count`. Эта информация основана на представлении `pg_stat_all_tables`. Столбцы этой таблицы перечислены в Таблице G.62.

Таблица G.62. Top tables by analyze operations (Таблицы, наиболее часто анализируемые)

Столбец	**Описание**	**Поле/вычисление**
`DB`	Имя базы данных, к которой относится таблица	
`Tablespace`	Имя табличного пространства, в котором находится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Analyze count`	Сколько раз сбор статистики для этой таблицы был выполнен вручную	`analyze_count`
`Autoanalyze count`	Сколько раз сбор статистики для этой таблицы был выполнен фоновым процессом автоочистки	`autoanalyze_count`
`Ins`	Количество вставленных строк	`n_tup_ins`
`Upd`	Количество изменённых строк, включая изменения по схеме HOT	`n_tup_upd`
`Del`	Количество удалённых строк	`n_tup_del`
`Upd(HOT)`	Количество строк, изменённых по схеме HOT	`n_tup_hot_upd`

Таблица отчёта «Top indexes by estimated vacuum load» показывает индексы, с которыми предположительно связана наибольшая нагрузка при очистке. Эта нагрузка вычисляется как произведение числа операций очистки, выполненных с нижележащей таблицей, и размера индекса. Эта информация основана на представлении `pg_stat_all_indexes`. В отсутствие данных о размере, собираемых функцией `pg_relation_size()`, оценка размера берётся из поля `pg_class.relpages`. Для обозначения меньшей точности оценки она отображается в квадратных скобках. Столбцы этой таблицы перечислены в Таблице G.63.

Таблица G.63. Top indexes by estimated vacuum load (Индексы, провоцирующие наибольшую нагрузку при очистке)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится индекс	
`Tablespace`	Имя табличного пространства, в котором находится индекс	
`Schema`	Имя схемы, содержащей нижележащую таблицу	
`Table`	Имя таблицы, для которой создан индекс	
`Index`	Имя индекса	
`~Vacuum bytes`	Оценка нагрузки при очистке	(`vacuum_count` + `autovacuum_count`) * `index_size`
`Vacuum count`	Сколько раз очистка этой таблицы была выполнена вручную (`VACUUM FULL` не учитывается)	`vacuum_count`
`Autovacuum count`	Сколько раз очистка этой таблицы была выполнена фоновым процессом автоочистки	`autovacuum_count`
`IX size`	Средний объём индекса за отчётный интервал	
`Relsize`	Средний объём отношения за отчётный интервал	

Таблица отчёта «Top tables by dead tuples ratio» показывает таблицы с наибольшим процентом «мёртвых» кортежей, имеющие размер больше 5 МБ. Данная статистика актуальна на момент последней выборки в отчётном интервале. Эта информация основана на представлении `pg_stat_all_tables`. В отсутствие данных о размере, собираемых функцией `pg_relation_size()`, оценка размера берётся из поля `pg_class.relpages`. Для обозначения меньшей точности оценки она отображается в квадратных скобках. Столбцы этой таблицы перечислены в Таблице G.64.

Таблица G.64. Top tables by dead tuples ratio (Таблицы с наибольшим процентом «мёртвых» кортежей)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Live`	Оценочное количество «живых» строк	`n_live_tup`
`Dead`	Оценочное количество «мёртвых» строк	`n_dead_tup`
`%Dead`	Процент «мёртвых» строк среди всех строк таблицы	
`Last AV`	Время последней очистки таблицы фоновым процессом автоочистки	`last_autovacuum`

Столбец	Описание	Поле/вычисление
`Size`	Размер таблицы	`pg_table_size() - pg_relation_size(toast)`

Таблица отчёта «Top tables by modified tuples ratio» показывает таблицы с наибольшим процентом изменённых кортежей, имеющие размер больше 5 МБ. Данная статистика актуальна на момент последней выборки в отчётном интервале. Эта информация основана на представлении `pg_stat_all_tables`. В отсутствие данных о размере, собираемых функцией `pg_relation_size()`, оценка размера берётся из поля `pg_class.relpages`. Для обозначения меньшей точности оценок они отображаются в квадратных скобках. Столбцы этой таблицы перечислены в Таблице G.65.

Таблица G.65. Top tables by modified tuples ratio (Таблицы с наибольшим процентом изменённых кортежей)

Столбец	Описание	Поле/вычисление
`DB`	Имя базы данных, к которой относится таблица	
`Schema`	Имя схемы, содержащей таблицу	
`Table`	Имя таблицы	
`Live`	Оценочное количество «живых» строк	`n_live_tup`
`Dead`	Оценочное количество «мёртвых» строк	`n_dead_tup`
`Mod`	Оценочное число строк, изменённых в этой таблице с момента последнего сбора статистики	`n_mod_since_analyze`
`%Mod`	Процент изменённых строк среди всех строк таблицы	
`Last AA`	Время последнего выполнения сбора статистики для этой таблицы фоновым процессом автоочистки	`last_autoanalyze`
`Size`	Размер таблицы	`pg_table_size() - pg_relation_size(toast)`

G.1.11.10. Cluster settings during the report interval (Параметры кластера в отчётном интервале)

Этот раздел отчёта содержит таблицу, в которой показываются значения GUC-параметров Postgres Pro, данные функций `version()`, `pg_postmaster_start_time()`, `pg_conf_load_time()` и поле `system_identifier` функции `pg_control_system()`, полученные в отчётном интервале. Данные в этой таблице собраны в две группы — `Defined settings` (Заданные параметры) и `Default settings` (Параметры по умолчанию). Столбцы этой таблицы перечислены в Таблице G.66.

Таблица G.66. Cluster settings during the report interval (Параметры кластера в отчётном интервале)

Столбец	Описание
`Setting`	Имя параметра
`reset_val`	Поле `reset_val` представления `pg_settings` . Значения, изменявшиеся в отчётном интервале, выделяются полужирным шрифтом.

Столбец	Описание
`Unit`	Единица измерения параметра
`Source`	Файл конфигурации, в котором определён параметр, и номер строки в нём через двоеточие
`Notes`	Время выборки, в которой было получено данное значение в первый раз

G.1.11.11. Extension versions during the report interval (Версии расширения в отчётном интервале)

Этот раздел отчёта содержит таблицу со списком установленных версий расширения, найденных в базах данных за отчётный интервал. Столбцы `First seen` и `Last seen` не отображаются, если версии расширения не изменялись за отчётный интервал. Столбцы этой таблицы перечислены в Таблице G.67.

Таблица G.67. Extension versions during the report interval (Версии расширения в отчётном интервале)

Столбец	Описание
`Name`	Название расширения
`DB`	Имя базы данных
`First seen`	Время выборки, в которой эта версия расширения появилась в первый раз
`Last seen`	Время выборки, в которой эта версия расширения появилась в последний раз
`Version`	Название версии расширения

G.1.12. Диагностические средства pgpro_pwr

В pgpro_pwr имеются средства для самодиагностики.

G.1.12.1. Сбор детальной статистики о времени выполнения процедур получения выборок

Расширение pgpro_pwr собирает подробную информацию о длительности действий, связанных с получением выборок, когда включён параметр *`pgpro_pwr.track_sample_timings`*. Данную информацию вы можете просмотреть в представлении `v_sample_timings`. Столбцы этого представления перечислены в Таблице G.68.

Таблица G.68. Представление `v_sample_timings`

Столбец	Описание
`server_name`	Имя сервера
`sample_id`	Идентификатор выборки
`sample_time`	Время получения выборки
`sampling_event`	Этап получения выборки. Описания всех этапов приведены в Таблице G.69.
`time_spent`	Длительность данного события

Таблица G.69. Описание событий `sampling_event`

Событие	Описание
`total`	Получение выборки (все этапы)
`connect`	Установление подключения к серверу (с использованием dblink)

Событие	Описание
`get server environment`	Получение от сервера GUC-параметров, списка доступных расширений и т. п.
`collect database stats`	Получение из представления `pg_stat_database` статистики по базам данных
`calculate database stats`	Вычисление изменения статистики по базам данных относительно предыдущей выборки
`collect tablespace stats`	Получение из представления `pg_tablespace` статистики по табличным пространствам
`collect statement stats`	Сбор статистики по SQL-операторам с использованием расширений pgpro_stats и pg_stat_kcache
`query pg_stat_bgwriter`	Сбор статистики уровня кластера с использованием представления `pg_stat_bgwriter`
`query pg_stat_archiver`	Сбор статистики уровня кластера с использованием представления `pg_stat_archiver`
`collect object stats`	Сбор статистики по объектам базы (включает события из Таблицы G.70)
`maintain repository`	Выполнение процедур обслуживания
`calculate tablespace stats`	Вычисление изменения статистики по табличным пространствам
`calculate object stats`	Вычисление изменения статистики по объектам базы (включает события из Таблицы G.71)
`calculate cluster stats`	Вычисление изменения статистики на уровне кластера
`calculate archiver stats`	Вычисление изменения статистики архиватора
`delete obsolete samples`	Удаление устаревших выборочных линий и выборок

Таблица G.70. Events of Collecting Statistics on Database Objects (События, связанные со сбором статистики по объектам БД)

Событие	Описание
`db:`*`имя_бд`*` collect tables stats`	Сбор статистики по таблицам базы *`имя_бд`*
`db:`*`имя_бд`*` collect indexes stats`	Сбор статистики по индексам базы *`имя_бд`*
`db:`*`имя_бд`*` collect functions stats`	Сбор статистики по функциям базы *`имя_бд`*

Таблица G.71. Events of Calculating Differences of Statistics on Database Objects (События, связанные с вычислением изменения статистики по объектам БД)

Событие	Описание
`calculate tables stats`	Вычисление изменения статистики по таблицам всех баз данных
`calculate indexes stats`	Вычисление изменения статистики по индексам всех баз данных
`calculate functions stats`	Вычисление изменения статистики по функциям всех баз данных

G.1.13. Важные замечания

Используя расширение pgpro_pwr, имейте в виду следующее:

- Postgres Pro собирает статистику о выполнении запроса *после* того, как он будет выполнен. Если выполнение одного запроса по времени охватывает несколько выборок, связанная с ним статистика попадёт только в ту выборку, ко времени которой он завершится. Кроме того, получить статистику по запросам, выполняемым в текущий момент, нет возможности. Также учтите, что процедуры обслуживания, например очистки и контрольных точек, обновляют статистику только при завершении.
- В случае сброса статистики Postgres Pro информация в следующей выборке может оказаться неточной.
- Установленные для отношений исключительные блокировки препятствуют вычислению размера этих отношений. Если функция `take_sample()` не сможет дождаться снятия такой блокировки в течение короткого периода времени (за 3 секунды), её выполнение прервётся и выборка не будет получена.

G.2. pgpro_stats — сбор статистики планирования и выполнения всех обрабатываемых сервером SQL-операторов

Расширение `pgpro_stats` предоставляет возможности для сбора статистики планирования и выполнения всех обрабатываемых сервером SQL-операторов. Оно основано на модуле pg_stat_statements и реализует следующую дополнительную функциональность:

- Сохранение помимо текста запросов соответствующих планов запросов.
- Возможность изменения частоты сбора статистики для оптимизации связанных с ним издержек.
- Вычисление статистики по событиям ожидания для выполняемых запросов.
- Подсчёт статистики использования ресурсов при планировании и выполнении операторов.
- Подсчёт статистики аннулирования кеша.
- Вычисление дополнительной статистики архиватора.
- Предоставление возможности просмотра статистики по очистке баз данных, таблиц и индексов, собранных ядром системы.
- Трассировка сеансов приложений.
- Создание представлений, эмулирующих другие расширения.

Примечание

Во время остановки сервера расширение pgpro_stats выгружает собранную статистику в два файла на диске в формате, который зависит от количества и порядка столбцов в представлениях `pgpro_stats_statements` и `pgpro_stats_totals`. Если этот формат изменился в новой версии pgpro_stats, то ранее сохранённые статистики не будут прочитаны при первом перезапуске сервера после обновления расширения, и в журнал будет записана ошибка:

```
LOG:  pgpro_stats: could not load statements data file of obsolete format
 "pg_stat/pgpro_stats.stat"
```

или

```
LOG:  pgpro_stats: could not load statements data file of obsolete format
 "pg_stat/pgpro_stats_totals.stat"
```

Содержимое файлов будет проигнорировано, а соответствующие статистики — обнулены. Чтобы узнать, изменился ли формат, ознакомьтесь с Замечаниями к выпускам для устанавливаемой версии pgpro_stats. Чтобы сохранить ранее собранные статистики, выгрузите их в CSV-файлы перед обновлением.

Описание типов и представлений, отображающих статистику аннулирования кешей, а также общая информация об этой статистике приводится в разделе Подраздел G.2.9.

G.2.1. Ограничения

- При некоторых обстоятельствах `pgpro_stats` может не сопоставить параметры в тексте запроса с теми же параметрами в соответствующем плане запроса.
- Некоторые запросы SPI в статистике не учитываются.
- Тексты и планы некоторых запросов SPI не нормализованы.
- Расширение `pgpro_stats` несовместимо с pg_stat_statements, а также другими расширениями, встраивающими свои обработчики в анализатор, планировщик или исполнитель и влияющими на разбор, планирование или выполнение запросов. Более того, если и `pgpro_stats`, и pg_stat_statements включены в список `shared_preload_libraries`, сервер баз данных не стартует. Также обратите внимание, что для получения окончательной версии запросов и планов библиотека `pgpro_stats` должна указываться последней в списке `shared_preload_libraries`, но есть расширения, которые работают, только если их библиотека указана последней.
- `pgpro_stats` может не работать корректно со сторонними расширениями, добавляющими в планы узлы `CustomScan` и `ForeignScan`.

G.2.2. Установка и подготовка

Расширение `pgpro_stats` поставляется вместе с Postgres Pro Standard в виде отдельного пакета `pgpro-stats-std-17` (подробные инструкции по установке представлены в Главе 16). Установив `pgpro_stats`, выполните следующие действия, чтобы подготовить `pgpro_stats` к работе:

1. Добавьте `pgpro_stats` в параметр shared_preload_libraries в файле `postgresql.conf`:

   ```
   shared_preload_libraries = 'pgpro_stats'
   ```
2. Перезапустите Postgres Pro Standard, чтобы изменения вступили в силу.

   Когда сервер перезапустится, `pgpro_stats` начнёт собирать статистику по всем базам данных в кластере. При необходимости вы можете изменять область сбора или отключить его, воспользовавшись параметрами конфигурации `pgpro_stats`.
3. Обращаться к собираемой статистике вы сможете после того, как создадите расширение `pgpro_stats`:

   ```
   CREATE EXTENSION pgpro_stats;
   ```

Кроме того, необходимо включить вычисление идентификатора запроса, чтобы расширение pgpro_stats было активно. Это происходит автоматически, если для compute_query_id задано значение `auto` или `on`, или загружен любой сторонний модуль, вычисляющий идентификаторы запросов.

G.2.3. Использование

- Сбор статистики по операторам запросов и планам
- Сбор дополнительных метрик

G.2.3.1. Сбор статистики по операторам запросов и планам

После того как расширение `pgpro_stats` будет установлено, оно начнёт собирать статистику по выполняемым операторам. Собираемые данные во многом пересекаются с теми, что собирает pg_stat_statements, но также включают информацию о планах запросов и событиях ожидания для каждого типа запроса. Полученная статистика сохраняется в кольцевом буфере в памяти, а прочитать её можно через представление pgpro_stats_statements.

По умолчанию `pgpro_stats` собирает статистику по всем операторам, с учётом значений параметров pgpro_stats.track и pgpro_stats.track_utility. Однако в целях оптимизации производительности вы можете уменьшить частоту выборки, воспользовавшись параметром pgpro_stats.query_sample_rate, и тогда `pgpro_stats` для вычисления статистики будет случайным образом выбирать заданную долю запросов из общего количества.

Для сбора статистики по событиям ожидания `pgpro_stats` производит выборки с определённой периодичностью. Эта периодичность задаётся параметром pgpro_stats.profile_period и по умолчанию равна 10 мс. Если выборка показывает, что процесс находится в состоянии ожидания, к общей длительности ожидания события добавляется значение pgpro_stats.profile_period. Таким образом, оценка длительности ожидания остаётся корректной даже при изменениях значения pgpro_stats.profile_period. Если вас не интересует статистика по событиям ожидания, сбор этой информации можно отключить, установив для параметра pgpro_stats.enable_profile значение `false`.

Показатели `pgpro_stats_statements.plans` и `pgpro_stats_statements.calls` не обязательно должны совпадать, так как статистика планирования и выполнения обновляется в конце соответствующей фазы и только при успешном завершении этой фазы. Например, если для оператора успешно выполнилось планирование, но во время выполнения произошла ошибка, изменится только статистика планирования. Если же планирование пропускается по причине использования кешированного плана, увеличивается только счётчик выполнения.

Например, давайте создадим таблицу со случайными данными и построим в ней индекс:

```
CREATE TABLE test AS (SELECT i, random() x FROM generate_series(1,1000000) i);
CREATE INDEX test_x_idx ON test (x);
```

Выполните несколько раз следующий запрос с различными значениями *`:x_min`* и *`:x_max`*:

```
select * from test where x >= :x_min and x <= :x_max;
```

После этого собранная статистика должна появиться в представлении `pgpro_stats_statements`:

```
SELECT queryid, query, planid, plan, wait_stats FROM pgpro_stats_statements WHERE query
 LIKE 'select * from test where%';
-[ RECORD
 1 ]---------------------------------------------------------------------------------------
queryid    | 1109491335754870054
query      | select * from test where x >= $1 and x <= $2
planid     | 8287793242828473388
plan       | Gather
           |   Output: i, x
           |   Workers Planned: 2
           |   ->  Parallel Seq Scan on public.test
           |         Output: i, x
           |         Filter: ((test.x >= $3) AND (test.x <= $4))
           |
wait_stats | {"IO": {"DataFileRead": 10}, "IPC": {"BgWorkerShutdown": 10}, "Total":
 {"IO": 10, "IPC": 10, "Total": 20}}
-[ RECORD
 2 ]---------------------------------------------------------------------------------------
queryid    | 1109491335754870054
query      | select * from test where x >= $1 and x <= $2
planid     | -9045072158333552619
plan       | Bitmap Heap Scan on public.test
           |   Output: i, x
           |   Recheck Cond: ((test.x >= $3) AND (test.x <= $4))
           |   ->  Bitmap Index Scan on test_x_idx
           |         Index Cond: ((test.x >= $5) AND (test.x <= $6))
           |
wait_stats | {"IO": {"DataFileRead": 40}, "Total": {"IO": 40, "Total": 40}}
-[ RECORD
 3 ]---------------------------------------------------------------------------------------
queryid    | 1109491335754870054
query      | select * from test where x >= $1 and x <= $2
```

```
planid     | -1062789671372193287
plan       | Seq Scan on public.test
           |   Output: i, x
           |   Filter: ((test.x >= $3) AND (test.x <= $4))
           |
wait_stats | NULL
-[ RECORD
 4 ]---------------------------------------------------------------------------------------------------------------
queryid    | 1109491335754870054
query      | select * from test where x >= $1 and x <= $2
planid     | -1748292253893834280
plan       | Index Scan using test_x_idx on public.test
           |   Output: i, x
           |   Index Cond: ((test.x >= $3) AND (test.x <= $4))
           |
wait_stats | NULL
```

G.2.3.2. Сбор дополнительных метрик

Используя `pgpro_stats`, вы можете определить дополнительные интересующие вас метрики. Собираемые данные будут накапливаться в кольцевом буфере для последующей передачи их в систему мониторинга. Этот подход лучше, чем непосредственный опрос базы данных системой мониторинга, так как в случае кратковременного прерывания соединения собранные данные не будут потеряны, а сохранятся в буфере и могут быть считаны позднее.

Чтобы настроить сбор дополнительной метрики, выполните следующее:

1. Для каждой метрики необходимо определить все параметры конфигурации, перечисленные в Подразделе G.2.7.2. В именах параметров должен присутствовать уникальный идентификатор (номер) метрики.

   Например, чтобы раз в 60 секунд собирать информацию о раздувании индексов, вы можете определить первую дополнительную метрику, установив соответствующие параметры следующим образом:

   ```
   pgpro_stats.metric_1_name = index_bloat
   pgpro_stats.metric_1_query = 'select iname, ibloat, ipages from bloat'
   pgpro_stats.metric_1_db = 'postgres'
   pgpro_stats.metric_1_user = postgres
   pgpro_stats.metric_1_period = '60s'
   ```

2. Перезапустите сервер.

   `pgpro_stats` начинает собирать статистику по выполняемым операторам и сохраняет её в кольцевом буфере; собранная статистика появляется в представлении `pgpro_stats_metrics`:

   ```
   SELECT * FROM pgpro_stats_metrics;
   ```

   Когда метрика уже добавлена, её параметры можно изменять без перезапуска сервера, просто перезагружая файл конфигурации `postgresql.conf`.

3. При необходимости настройте экспорт данных в систему мониторинга, которую вы используете.

G.2.4. Представления

G.2.4.1. Представление `pgpro_stats_statements`

Статистика, собираемая модулем, выдаётся через представление с именем `pgpro_stats_statements`. Это представление содержит отдельные строки для каждой комбинации идентификатора базы данных, идентификатора пользователя и идентификатора запроса (но в количестве, не превышающем максимальное число различных операторов, которые может отслеживать модуль). Столбцы представления показаны в Таблице G.72.

Таблица G.72. Столбцы `pgpro_stats_statements`

Name	Тип	Ссылки	Описание
`userid`	`oid`	`pg_authid .oid`	OID пользователя, выполнявшего оператор
`dbid`	`oid`	`pg_database .oid`	OID базы данных, в которой выполнялся оператор
`toplevel`	`bool`		True, если данный запрос выполнялся на верхнем уровне (всегда true, если для параметра pgpro_stats.track задано значение `top`)
`queryid`	`bigint`		Внутренний хеш-код, вычисленный для дерева разбора оператора
`planid`	`bigint`		Внутренний хеш-код, вычисленный для дерева плана оператора
`query`	`text`		Текст, представляющий оператор
`plan`	`text`		План запроса, представленный в формате, определённом параметром конфигурации pgpro_stats.plan_format
`plans`	`int8`		Число операций планирования этого оператора (если включён параметр `pgpro_stats.track_planning` , иначе 0)
`total_plan_time`	`float8`		Общее время, затраченное на планирование этого оператора в миллисекундах (если включён параметр `pgpro_stats.track_planning` , иначе 0)
`min_plan_time`	`float8`		Минимальное время, затраченное на планирование этого оператора в миллисекундах (если включён параметр `pgpro_stats.track_planning` , иначе 0)
`max_plan_time`	`float8`		Максимальное время, затраченное на планирование этого оператора в миллисекундах (если включён параметр

Name	Тип	Ссылки	Описание
			`pgpro_stats.track_planning` , иначе 0)
`mean_plan_time`	`float8`		Среднее время, затраченное на планирование этого оператора в миллисекундах (если включён параметр `pgpro_stats.track_planning` , иначе 0)
`stddev_plan_time`	`float8`		Стандартное отклонение времени, затраченного на планирование этого оператора в миллисекундах (если включён параметр `pgpro_stats.track_planning` , иначе 0)
`plan_rusage`	pgpro_stats_rusage		Статистика использования ресурсов при планировании оператора.
`calls`	`int8`		Счётчик выполнений данного оператора
`total_exec_time`	`float8`		Общее время, затраченное на выполнение оператора, в миллисекундах
`min_exec_time`	`float8`		Минимальное время, потраченное на выполнение оператора, в миллисекундах
`max_exec_time`	`float8`		Максимальное время, потраченное на выполнение оператора, в миллисекундах
`mean_exec_time`	`float8`		Среднее время, потраченное на выполнение оператора, в миллисекундах
`stddev_exec_time`	`float8`		Стандартное отклонение времени, затраченного на выполнение оператора, в миллисекундах
`exec_rusage`	pgpro_stats_rusage		Статистика использования ресурсов при выполнении оператора.
`rows`	`int8`		Общее число строк, полученных или затронутых оператором
`shared_blks_hit`	`int8`		Общее число попаданий разделяемых бло-

Name	Тип	Ссылки	Описание
			ков в кеш для данного оператора
`shared_blks_read`	`int8`		Общее число разделяемых блоков, прочитанных данным оператором
`shared_blks_dirtied`	`int8`		Общее число разделяемых блоков, «загрязнённых» данным оператором
`shared_blks_written`	`int8`		Общее число разделяемых блоков, записанных данным оператором
`local_blks_hit`	`int8`		Общее число попаданий локальных блоков в кеш для данного оператора
`local_blks_read`	`int8`		Общее число локальных блоков, прочитанных данным оператором
`local_blks_dirtied`	`int8`		Общее число локальных блоков, «загрязнённых» данным оператором
`local_blks_written`	`int8`		Общее число локальных блоков, записанных данным оператором
`temp_blks_read`	`int8`		Общее число временных блоков, прочитанных данным оператором
`temp_blks_written`	`int8`		Общее число временных блоков, записанных данным оператором
`shared_blk_read_time`	`float8`		Общее время, затраченное оператором на чтение разделяемых блоков, в миллисекундах (если включён track_io_timing, иначе ноль)
`shared_blk_write_time`	`float8`		Общее время, затраченное оператором на запись разделяемых блоков, в миллисекундах (если включён track_io_timing, иначе ноль)

Name	Тип	Ссылки	Описание
`local_blk_read_time`	`float8`		Общее время, затраченное оператором на чтение локальных блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае). В версиях Postgres Pro ниже 17 всегда содержит ноль.
`local_blk_write_time`	`float8`		Общее время, затраченное оператором на запись локальных блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае). В версиях Postgres Pro ниже 17 всегда содержит ноль.
`temp_blk_read_time`	`float8`		Общее время, затраченное оператором на чтение временных блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае). В версиях Postgres Pro ниже 15 всегда содержит ноль.
`temp_blk_write_time`	`float8`		Общее время, затраченное оператором на запись временных блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае). В версиях Postgres Pro ниже 15 всегда содержит ноль.
`wal_records`	`int8`		Общее число записей WAL, сгенерированных при выполнении оператора
`wal_fpi`	`int8`		Общее число образов полных страниц в WAL, сгенерированных при выполнении оператора
`wal_bytes`	`numeric`		Общий объём WAL (в байтах), сгенерированный при выполнении оператора

Name	Тип	Ссылки	Описание
`jit_functions`	`int8`		Общее число функций, скомпилированных в JIT-код при выполнении оператора. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_generation_time`	`float8`		Общее время, затраченное на компиляцию JIT-кода при выполнении оператора, в миллисекундах. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_inlining_count`	`int8`		Количество встраиваний функций, используемых в операторе. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_inlining_time`	`float8`		Общее время, затраченное на встраивание функций при выполнении оператора, в миллисекундах. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_optimization_count`	`int8`		Количество JIT-оптимизаций для оператора. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_optimization_time`	`float8`		Общее время, затраченное на JIT-оптимизацию при выполнении оператора, в миллисекундах. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_emission_count`	`int8`		Сколько раз выдавался код при выполнении оператора. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_emission_time`	`float8`		Общее время, затраченное на выдачу кода при выполнении оператора, в миллисекундах. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_deform_count`	`int8`		Общее число функций преобразования кортежей, скомпилирован-

Name	**Тип**	**Ссылки**	**Описание**
			ных в JIT-код при выполнении оператора. В версиях Postgres Pro ниже 17 всегда содержит ноль.
`jit_deform_time`	`float8`		Общее время, затраченное оператором на компилирование функций преобразования кортежей в JIT-код, в миллисекундах. В версиях Postgres Pro ниже 17 всегда содержит ноль.
`wait_stats`	`jsonb`		Объект типа `jsonb`, содержащий статистику по событиям ожидания для каждого выполнения запроса по соответствующему плану. Все показатели представлены в миллисекундах и кратны параметру конфигурации pgpro_stats.profile_period.
`inval_msgs`	pgpro_stats_inval_msgs		Число сообщений аннулирования кеша по типам, выданных при выполнении оператора (если это поддерживает сервер, иначе — 0).
`stats_since`	`timestamp with time zone`		Время начала сбора статистики по этому оператору
`minmax_stats_since`	`timestamp with time zone`		Время начала сбора статистики минимальных/максимальных значений по этому оператору (столбцы `min_plan_time` , `max_plan_time` , `min_exec_time` и `max_exec_time`)

Аналогично расширению pg_stat_statements, DML запросы (то есть `SELECT`, `INSERT`, `UPDATE`, `DELETE` и `MERGE`) в pgpro_stats объединяются в одну запись, когда они имеют идентичные структуры запросов согласно внутреннему вычисленному хешу. Обычно два запроса будут считаться равными при таком сравнении, если они семантически равнозначны, не считая значений констант, фигурирующих в запросе. Когда значение константы игнорируется в целях сравнения запроса с другими запросами, эта константа заменяется в выводе pgpro_stats обозначением параметра, например, `$k`, где `k` - натуральное число. Если в запросе уже есть параметры, то начальное значение `k` равно номеру, следующим за номером последнего параметра `$n` в исходном тексте запроса. Если параметров нет, то начальное значение `k` равно единице. Стоит отметить, что в некоторых случаях на эту нумерацию могут влиять скрытые символы параметров. Например, PL/pgSQL применяет такие символы для добавления в запросы значений локальных переменных функций, так что оператор PL/pgSQL вида `SELECT i + 1 INTO j` будет представлен в тексте как `SELECT i + $2`.

Подобный механизм объединения текстов pgpro_stats использует и для текстов планов. При этом делается попытка сопоставить номера констант в тексте плана соответствующим номерам констант в тексте запроса. Если такая попытка для какой-либо константы в тексте плана оканчивается неудачей, то ей присваивается следующий номер, больший максимального номера заменённой константы в тексте запроса. Например, для запроса:

```
SELECT 1::int, 'abc'::VARCHAR(3), 2::int;
```

в его тексте и в тексте соответствующего плана pgpro_stats подставит номера констант следующим образом:

```
postgres=# SELECT query, plan FROM pgpro_stats_statements;
                     query                      |                                plan
------------------------------------------------
+--------------------------------------------------
SELECT $1::int, $2::VARCHAR(3), $3::int         | Result
            +
                                                |   Output: $1, $4, $3
            +
```

В данном тексте плана удалось сопоставить константы с номерами 1 и 3 из текста запроса, а константу с номером 2 сопоставить не удалось, и она была заменена номером, следующим за максимальным номером в тексте запроса — номером 4.

При подстановке номеров в тесте плана делается исключение для номера версии XML документа. Если в исходном запросе он был задан константой, например '1.0', то в тексте плана это значение сохранится как есть и не будет заменено на подстановочный символ вида `$k`. Если в исходном запросе номер версии XML документа был задан выражением, то подстановка символов вместо констант будет сделана по обычным правилам.

G.2.4.2. Представление `pgpro_stats_totals`

Агрегированная статистика, собранная модулем, выдаётся через представление `pgpro_stats_totals`. Это представление содержит отдельные строки для каждого отдельного объекта БД (но в количестве, не превышающем максимальное число различных объектов, которое может отслеживать модуль). Столбцы представления показаны в Таблице G.73.

Таблица G.73. Столбцы `pgpro_stats_totals`

Name	Тип	Описание
`object_type`	`text`	Тип объекта, для которого агрегируется статистика: «cluster» (кластер БД), «database» (база данных), «user» (пользователь), «client_addr» (адрес клиента), «application» (приложение), «backend» (обслуживающий процесс), «session» (сеанс)
`object_id`	`bigint`	Идентификатор объекта: oid для баз и пользователей, pid для обслуживающих процессов, sid для сеансов, NULL для иных объектов
`object_name`	`text`	Текстовое имя объекта либо NULL

Name	**Тип**	**Описание**
`queries_planned`	`int8`	Число спланированных запросов
`total_plan_time`	`float8`	Общее время, затраченное на планирование операторов, в миллисекундах
`total_plan_rusage`	pgpro_stats_rusage	Агрегированная статистика использования ресурсов при планировании операторов
`queries_executed`	`int8`	Число выполненных запросов
`total_exec_time`	`float8`	Общее время, затраченное на выполнение операторов, в миллисекундах
`total_exec_rusage`	pgpro_stats_rusage	Агрегированная статистика использования ресурсов при выполнении операторов
`rows`	`int8`	Общее число строк, полученных или затронутых операторами
`shared_blks_hit`	`int8`	Общее число попаданий разделяемых блоков в кеш для данных операторов
`shared_blks_read`	`int8`	Общее число разделяемых блоков, прочитанных операторами
`shared_blks_dirtied`	`int8`	Общее число разделяемых блоков, «загрязнённых» операторами
`shared_blks_written`	`int8`	Общее число разделяемых блоков, записанных операторами
`local_blks_hit`	`int8`	Общее число попаданий локальных блоков в кеш для данных операторов
`local_blks_read`	`int8`	Общее число локальных блоков, прочитанных операторами
`local_blks_dirtied`	`int8`	Общее число локальных блоков, «загрязнённых» операторами
`local_blks_written`	`int8`	Общее число локальных блоков, записанных операторами
`temp_blks_read`	`int8`	Общее число локальных блоков, прочитанных операторами
`temp_blks_written`	`int8`	Общее число временных блоков, записанных операторами
`shared_blk_read_time`	`float8`	Общее время, затраченное операторами на чтение разделяемых блоков, в миллисекундах (если включён track_io_timing, иначе ноль)
`shared_blk_write_time`	`float8`	Общее время, затраченное операторами на запись разделяе-

Name	Тип	Описание
		мых блоков, в миллисекундах (если включён track_io_timing, иначе ноль)
`local_blk_read_time`	`float8`	Общее время, затраченное операторами на чтение локальных блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае). В версиях Postgres Pro ниже 17 всегда содержит ноль.
`local_blk_write_time`	`float8`	Общее время, затраченное операторами на запись локальных блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае). В версиях Postgres Pro ниже 17 всегда содержит ноль.
`temp_blk_read_time`	`float8`	Общее время, затраченное операторами на чтение временных блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае). В версиях Postgres Pro ниже 15 всегда содержит ноль.
`temp_blk_write_time`	`float8`	Общее время, затраченное операторами на запись временных блоков, в миллисекундах (если включён track_io_timing, или ноль в противном случае). В версиях Postgres Pro ниже 15 всегда содержит ноль.
`wal_records`	`int8`	Общее число записей WAL, сгенерированных при выполнении операторов
`wal_fpi`	`int8`	Общее число образов полных страниц в WAL, сгенерированных при выполнении операторов
`wal_bytes`	`numeric`	Общий объём WAL (в байтах), сгенерированный при выполнении операторов
`jit_functions`	`int8`	Общее число функций, скомпилированных в JIT-код при выполнении операторов. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_generation_time`	`float8`	Общее время, затраченное на компиляцию JIT-кода при выполнении операторов, в миллисекундах. В версиях Postgres Pro ниже 15 всегда содержит ноль.

Name	**Тип**	**Описание**
`jit_inlining_count`	`int8`	Количество встраиваний функций, используемых в операторах. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_inlining_time`	`float8`	Общее время, затраченное на встраивание функций при выполнении операторов, в миллисекундах. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_optimization_count`	`int8`	Количество JIT-оптимизаций для операторов. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_optimization_time`	`float8`	Общее время, затраченное на JIT-оптимизацию при выполнении операторов, в миллисекундах. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_emission_count`	`int8`	Сколько раз выдавался код при выполнении операторов. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_emission_time`	`float8`	Общее время, затраченное на выдачу кода при выполнении операторов, в миллисекундах. В версиях Postgres Pro ниже 15 всегда содержит ноль.
`jit_deform_count`	`int8`	Общее число функций преобразования кортежей, скомпилированных в JIT-код при выполнении операторов. В версиях Postgres Pro ниже 17 всегда содержит ноль.
`jit_deform_time`	`float8`	Общее время, затраченное операторами на компилирование функций преобразования кортежей в JIT-код, в миллисекундах. В версиях Postgres Pro ниже 17 всегда содержит ноль.
`wait_stats`	`jsonb`	Объект типа `jsonb`, содержащий статистику по событиям ожидания для каждого выполнения запроса. Все показатели представлены в миллисекундах и кратны параметру конфигурации pgpro_stats.profile_period.
`inval_msgs`	pgpro_stats_inval_msgs	Число сообщений аннулирования кеша по типам, выданных при выполнении операторов (

Name	Тип	Описание
		если это поддерживает сервер, иначе — 0).
`cache_resets`	`int4`	Число сбросов разделяемого кеша (только для кластера БД, баз данных и обслуживающих процессов). Увеличивается для обслуживающего процесса, когда он получает сообщение, вызывающее полный сброс кеша.
`stats_since`	`timestamp with time zone`	Время, когда начался сбор статистики по операторам

G.2.4.3. Представление `pgpro_stats_info`

Статистика самого модуля `pgpro_stats` собирается и выдаётся через представление с именем `pgpro_stat_info`. Это представление содержит только одну строку. Столбцы представления показаны в Таблице G.74.

Таблица G.74. Столбцы `pgpro_stats_info`

Name	Тип	Описание
`dealloc`	`bigint`	Сколько всего раз в `pgpro_stats_statements` отбрасывались записи о редко выполняемых операторах, поскольку на обработку поступало больше различных операторов, чем задано в параметре pgpro_stats.max
`stats_reset`	`timestamp with time zone`	Время последнего сброса всех статистических данных в представлении `pgpro_stats_statements`

G.2.4.4. Представление `pgpro_stats_metrics`

Метрики, собранные модулем `pgpro_stats`, выводятся в представлении `pgpro_stats_metrics`. В таблице ниже описаны столбцы этого представления.

Таблица G.75. Столбцы `pgpro_stats_metrics`

Name	Тип	Описание
`metric_number`	`int4`	Уникальный идентификатор собираемой метрики, назначаемый пользователем. Этот идентификатор входит в имена параметров, определяющих метрику.
`metric_name`	`text`	Имя метрики, определённое параметром `pgpro_stats.metric_`*`N`*`_`*`name`*
`db_name`	`text`	Имя базы данных, для которой была получена метрика
`ts`	`timestamptz`	Время, когда было вычислено значение метрики

Name	Тип	Описание
`value`	`jsonb`	Результат запроса, производившего вычисление метрики. Он сериализуется в типе `jsonb` как массив объектов, полученных с помощью функции `to_jsonb(` *`строка_результата`* `)`. В случае ошибки возвращается один объект, содержащий поля `code` (код ошибки), `message` (сообщение), `detail` (подробности) и `hint` (подсказка).

G.2.4.5. Представление `pgpro_stats_archiver`

Представление `pgpro_stats_archiver` будет содержать одну строку, показывающую данные о процессе архивации кластера.

Таблица G.76. Столбцы `pgpro_stats_archiver`

Столбец	Тип	Описание
`archived_count`	`bigint`	Число файлов WAL, которые были успешно заархивированы
`last_archived_wal`	`text`	Имя последнего успешно заархивированного файла WAL
`last_archived_time`	`timestamp with time zone`	Время последней успешной архивации
`failed_count`	`bigint`	Число неудачных попыток архивации файлов WAL
`last_failed_wal`	`text`	Имя файла WAL, который обрабатывался в ходе последней неудавшейся архивации
`last_failed_time`	`timestamp with time zone`	Время последней неудавшейся архивации
`active_time`	`int8`	Общее время, в течение которого процесс архивации был активен
`archive_command_time`	`int8`	Общее время выполнения команды архивации
`stats_reset`	`timestamp with time zone`	Последнее время сброса этих статистических данных

G.2.4.6. Представление `pgpro_stats_vacuum_database`

> **Важно**
>
> Начиная с Postgres Pro 16, это представление не содержит данных, поскольку отображаемые в нём статистики доступны через представление каталога `pg_stats_vacuum_database` (за подробностями обратитесь к разделу Системные представления).

Представление `pgpro_stats_vacuum_database` будет содержать по одной строке для каждой базы данных в текущем кластере, показывая статистику очистки этой конкретной БД. Эта статистика собирается ядром системы, как описано в Разделе 26.2. В таблице ниже описаны столбцы представления.

Таблица G.77. Столбцы `pgpro_stats_vacuum_database`

Столбец	Тип	Описание
`dbid`	`oid`	OID базы данных
`total_blks_read`	`int8`	Количество блоков базы данных, прочитанных операциями очистки для этой БД
`total_blks_hit`	`int8`	Количество попаданий в буферный кеш блоков базы данных в процессе операций очистки, выполненных для этой БД
`total_blks_dirtied`	`int8`	Количество блоков базы данных, загрязнённых операциями очистки для этой БД
`total_blks_written`	`int8`	Количество блоков базы данных, записанных операциями очистки для этой БД
`wal_records`	`int8`	Общее число записей WAL, сгенерированных операциями очистки для этой базы данных
`wal_fpi`	`int8`	Общее число образов полных страниц в WAL, сгенерированных операциями очистки для этой базы данных
`wal_bytes`	`numeric`	Общий объём WAL (в байтах), сгенерированный операциями очистки для этой базы данных
`blk_read_time`	`float8`	Общее время, затраченное на чтение блоков базы данных операциями очистки, выполненными для этой БД, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`blk_write_time`	`float8`	Общее время, затраченное на запись блоков базы данных операциями очистки, выполненными для этой БД, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`delay_time`	`float8`	Время простоя в точке задержки при выполнении операций очистки для этой базы данных, в миллисекундах (за подробностями обратитесь к Подразделу 18.4.4).
`system_time`	`float8`	Время использования процессора в режиме ядра при очистке этой базы данных, в миллисекундах
`user_time`	`float8`	Время использования процессора в пользовательском режи-

Столбец	Тип	Описание
		ме при очистке этой базы данных, в миллисекундах
`total_time`	`float8`	Общее время, затраченное на очистку этой базы данных, в миллисекундах
`interrupts`	`int4`	Сколько раз операции очистки для этой базы данных были прерваны из-за каких-либо ошибок

G.2.4.7. Представление `pgpro_stats_vacuum_tables`

> **Важно**
>
> Начиная с Postgres Pro 16, это представление не содержит данных, поскольку отображаемые в нём статистики доступны через представление каталога `pg_stats_vacuum_tables` (за подробностями обратитесь к разделу Системные представления).

Представление `pgpro_stats_vacuum_tables` будет содержать по одной строке для каждой таблицы в текущей базе данных (включая таблицы TOAST), показывая статистику очистки этой конкретной таблицы. Эта статистика собирается ядром системы, как описано в Разделе 26.2. В таблице ниже описаны столбцы представления.

Таблица G.78. Столбцы `pgpro_stats_vacuum_tables`

Столбец	Тип	Описание
`relid`	`oid`	OID таблицы
`schema`	`name`	Имя схемы, в которой расположена эта таблица
`relname`	`name`	Имя данной таблицы
`total_blks_read`	`int8`	Количество блоков БД, прочитанных операциями очистки для этой таблицы
`total_blks_hit`	`int8`	Количество попаданий в буферный кеш блоков базы данных в процессе операций очистки, выполненных для этой таблицы
`total_blks_dirtied`	`int8`	Количество блоков базы данных, загрязнённых операциями очистки для этой таблицы
`total_blks_written`	`int8`	Количество блоков базы данных, записанных операциями очистки для этой таблицы
`rel_blks_read`	`int8`	Количество блоков, прочитанных из этой таблицы операциями очистки
`rel_blks_hit`	`int8`	Сколько раз блоки этой таблицы уже обнаруживались в буферном кеше операциями очистки, так что чтение с диска не потребовалось (здесь учитываются только случаи обнаружения блоков в кеше

Столбец	Тип	Описание
		Postgres Pro, а не в кеше операционной системы)
`pages_scanned`	`int8`	Количество страниц, к которым обращались операции очистки, выполненные для этой таблицы
`pages_removed`	`int8`	Количество страниц, удалённых из физического хранилища операциями очистки для этой таблицы
`pages_frozen`	`int8`	Сколько раз страницы этой таблицы были помечены операциями очистки в карте видимости как полностью замороженные
`pages_all_visible`	`int8`	Сколько раз страницы этой таблицы были помечены операциями очистки в карте видимости как полностью видимые
`tuples_deleted`	`int8`	Количество «мёртвых» кортежей, удалённых операциями очистки из этой таблицы
`tuples_frozen`	`int8`	Количество кортежей этой таблицы, помеченных операциями очистки как замороженные
`dead_tuples`	`int8`	Количество «мёртвых» кортежей, оставленных операциями очистки в этой таблице из-за видимости этих кортежей в транзакциях
`index_vacuum_count`	`int8`	Сколько раз была выполнена очистка индексов этой таблицы
`rev_all_frozen_pages`	`int8`	Сколько раз для страниц данной таблицы снималась метка полной заморозки в карте видимости
`rev_all_visible_pages`	`int8`	Сколько раз для страниц данной таблицы снималась метка полной видимости в карте видимости
`wal_records`	`int8`	Общее число записей WAL, сгенерированных операциями очистки для этой таблицы
`wal_fpi`	`int8`	Общее число образов полных страниц в WAL, сгенерированных операциями очистки для этой таблицы
`wal_bytes`	`numeric`	Общий объём WAL (в байтах), сгенерированный операциями очистки для этой таблицы

Столбец	**Тип**	**Описание**
`blk_read_time`	`float8`	Общее время, затраченное на чтение блоков БД операциями очистки, выполненными для этой таблицы, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`blk_write_time`	`float8`	Общее время, затраченное на запись блоков БД операциями очистки, выполненными для этой таблицы, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`delay_time`	`float8`	Время простоя в точке задержки при выполнении операций очистки для этой таблицы, в миллисекундах (за подробностями обратитесь к Подразделу 18.4.4).
`system_time`	`float8`	Время использования процессора в режиме ядра при очистке этой таблицы, в миллисекундах
`user_time`	`float8`	Время использования процессора в пользовательском режиме при очистке этой таблицы, в миллисекундах
`total_time`	`float8`	Общее время, затраченное на очистку этой таблицы, в миллисекундах
`interrupts`	`integer`	Сколько раз операции очистки для этой таблицы были прерваны из-за каких-либо ошибок

Столбцы `total_*`, `wal_*` и `blk_*` содержат данные об очистке индексов этой таблицы, а столбцы `system_time` и `user_time` содержат только данные об очистке кучи.

G.2.4.8. Представление `pgpro_stats_vacuum_indexes`

Важно

Начиная с Postgres Pro 16, это представление не содержит данных, поскольку отображаемые в нём статистики доступны через представление каталога `pg_stats_vacuum_indexes` (за подробностями обратитесь к разделу Системные представления).

Представление `pgpro_stats_vacuum_indexes` будет содержать одну строку для каждого индекса в текущей базе данных (включая индексы таблицы TOAST), показывая статистику очистки этого конкретного индекса. Эта статистика собирается ядром системы, как описано в Разделе 26.2. В таблице ниже описаны столбцы представления.

Таблица G.79. Столбцы `pgpro_stats_vacuum_indexes`

Столбец	Тип	Описание
`relid`	`oid`	OID индекса
`schema`	`name`	Имя схемы, в которой расположен этот индекс
`relname`	`name`	Имя этого индекса
`total_blks_read`	`int8`	Количество блоков БД, прочитанных операциями очистки для этого индекса
`total_blks_hit`	`int8`	Количество попаданий в буферный кеш блоков базы данных в процессе операций очистки, выполненных для этого индекса
`total_blks_dirtied`	`int8`	Количество блоков базы данных, загрязнённых операциями очистки для этого индекса
`total_blks_written`	`int8`	Количество блоков базы данных, записанных операциями очистки для этого индекса
`rel_blks_read`	`int8`	Количество блоков, прочитанных для этого индекса операциями очистки
`rel_blks_hit`	`int8`	Сколько раз блоки этого индекса уже обнаруживались в буферном кеше операциями очистки, так что чтение с диска не потребовалось (здесь учитываются только случаи обнаружения блоков в кеше Postgres Pro, а не в кеше операционной системы ОС)
`pages_deleted`	`int8`	Количество страниц, которые были удалены операциями очистки, выполненными для этого индекса
`tuples_deleted`	`int8`	Количество «мёртвых» кортежей, удалённых операциями очистки для этого индекса
`wal_records`	`int8`	Общее число записей WAL, сгенерированных операциями очистки для этого индекса
`wal_fpi`	`int8`	Общее число образов полных страниц в WAL, сгенерированных операциями очистки для этого индекса
`wal_bytes`	`numeric`	Общий объём WAL (в байтах), сгенерированный операциями очистки для этого индекса
`blk_read_time`	`float8`	Общее время, затраченное на чтение блоков БД операциями

Столбец	Тип	Описание
		очистки, выполненными для этого индекса, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`blk_write_time`	`float8`	Общее время, затраченное на запись блоков БД операциями очистки, выполненными для этой таблицы, в миллисекундах (если включён параметр track_io_timing, или ноль в противном случае)
`delay_time`	`float8`	Время простоя в точке задержки при выполнении операций очистки для этого индекса, в миллисекундах (за подробностями обратитесь к Подразделу 18.4.4).
`system_time`	`float8`	Время использования процессора в режиме ядра при очистке этого индекса, в миллисекундах
`user_time`	`float8`	Время использования процессора в пользовательском режиме при очистке этого индекса, в миллисекундах
`total_time`	`float8`	Общее время, затраченное на очистку этого индекса, в миллисекундах
`interrupts`	`integer`	Сколько раз операции очистки для этого индекса были прерваны из-за каких-либо ошибок

G.2.5. Типы данных

G.2.5.1. Тип `pgpro_stats_rusage`

Тип `pgpro_stats_rusage` представляет собой запись, содержащую статистику использования ресурсов во время планирования и выполнения запросов. Поля этого типа показаны в Таблице G.80.

Таблица G.80. Поля `pgpro_stats_rusage`

Name	Тип	Описание
`reads`	`bigint`	Количество байт, прочитанное на уровне файловой системы
`writes`	`bigint`	Количество байт, записанное на уровне файловой системы
`user_time`	`double precision`	Время использования процессора в пользовательском режиме
`system_time`	`double precision`	Время использования процессора в режиме ядра

Name	**Тип**	**Описание**
`minflts`	`bigint`	Количество восстановленных страниц (мягких ошибок обращения к страницам)
`majflts`	`bigint`	Количество ошибок обращения к страницам (жёстких)
`nswaps`	`bigint`	Количество обращений к пространству подкачки
`msgsnds`	`bigint`	Количество переданных сообщений IPC
`msgrcvs`	`bigint`	Количество полученных сообщений IPC
`nsignals`	`bigint`	Количество полученных сигналов
`nvcsws`	`bigint`	Количество «добровольных» переключений контекста
`nivcsws`	`bigint`	Количество принудительных переключений контекста

G.2.6. Функции

`pgpro_stats_statements_reset(userid Oid, dbid Oid, queryid bigint, planid bigint, minmax_only boolean) returns timestamp with time zone`

Функция `pgpro_stats_statements_reset` очищает всю статистику, собранную к этому времени модулем `pgpro_stats` для заданного пользователя (`userid`), базы данных (`dbid`), запроса (`queryid`) и плана (`planid`). В случае отсутствия одного из параметров для него подразумевается нулевое значение (неприменимое ограничение) и очищается статистика, соответствующая другим параметрам. Если никакой параметр не задан или все параметры имеют нулевое значение (неприменимое), очищается вся статистика. Вместе с очисткой всей статистики в `pgpro_stats_statements` также сбрасывается статистика в `pgpro_stats_info`. Когда для параметра `minmax_only` установлено значение `true`, сбрасываются значения только минимального и максимального времени планирования и выполнения оператора (т. е. значения столбцов `min_plan_time`, `max_plan_time`, `min_exec_time` и `max_exec_time`). Значение `minmax_only` по умолчанию — `false`. Время последнего сброса минимальных/максимальных значений можно увидеть в столбце `minmax_stats_since` представления `pgpro_stats_statements`. Функция возвращает время сброса статистики. Это время записывается в поле `stats_reset` представления `pgpro_stats_info` или `minmax_stats_since` представления `pgpro_stats_statements`, если соответствующий сброс значений был действительно выполнен. По умолчанию эту функцию могут выполнять только суперпользователи. Другим пользователям можно дать доступ к ней, используя `GRANT`.

> **Примечание**
>
> Поскольку статистика в представлениях `pgpro_stats_vacuum_database`, `pgpro_stats_vacuum_tables` и `pgpro_stats_vacuum_indexes` собирается ядром системы, для её сброса вызовите функцию `pg_stat_reset( )` (за подробностями обратитесь к Подразделу 26.2.26).

`pgpro_stats_statements(showtext boolean) returns setof record`

Представление `pgpro_stats_statements` определяется как результат вызова одноимённой функции. Функцию `pgpro_stats_statements` можно также вызывать непосредственно, а с аргументом `showtext := false` можно получить результат без текста запроса (то есть в выходном аргументе, соответствующем столбцу представления `query`, будет выдаваться null). Эта возможность предусмотрена для того, чтобы внешние средства мониторинга могли не загружать каждый раз текст запросов, который может быть довольно объёмным. Средство мониторинга может просто один раз запомнить текст запросов для каждой записи в представлении (как это и делает сам модуль `pgpro_stats`) и затем запрашивать текст для новых запросов только при необходимости. Так как сервер сохраняет текст запросов в файле, при таком подходе постоянное чтение содержимого `pgpro_stats_statements` будет создавать меньшую нагрузку на ввод-вывод.

`pgpro_stats_info() returns record`

Представление `pgpro_stats_info` построено на базе одноимённой функции, которую пользователи также могут вызывать непосредственно.

`pgpro_stats_totals_reset(type text, id bigint) returns timestamp with timezone`

Функция `pgpro_stats_totals_reset` очищает всю статистику, собранную к этому времени модулем `pgpro_stats` для заданного типа (`type`) и идентификатора (`id`) объекта. В случае отсутствия параметров или когда параметр `type` равен `0`, очищается вся статистика. Если в параметре `type` передаётся корректный тип объекта, то в случае указания `id` очищается статистика только для объекта с заданным идентификатором, а в отсутствие такого указания — для всех объектов заданного типа. Если же параметр `type` не задаёт существующий тип объекта, никакая статистика не очищается. Эта функция возвращает время выполнения очистки. По умолчанию эту функцию могут выполнять только суперпользователи. Другим пользователям можно дать доступ к ней, используя `GRANT`.

`pgpro_stats_totals() returns setof record`

Представление `pgpro_stats_totals` построено на базе одноимённой функции, которую пользователи также могут вызывать непосредственно.

`pgpro_stats_metrics() returns setof record`

Определяет представление `pgpro_stats_metrics`, описанное подробно в Таблице G.75.

`pgpro_stats_get_archiver() returns setof record`

Определяет представление `pgpro_stats_archiver`, описанное подробно в Таблице G.76.

`pgpro_stats_wal_sender_crc_errors() returns bigint`

В Postgres Pro Enterprise возвращает количество ошибок, обнаруженных процессом-передатчиком WAL, когда параметр `wal_sender_check_crc` включён (`on`). В Postgres Pro Standard возвращает ноль. Обратите внимание, что при перезапуске кластера этот счётчик сбрасывается до нуля.

`pgpro_stats_vacuum_database(dboid oid) returns setof record`

Определяет строку представления `pgpro_stats_vacuum_database`, которое подробно описано в Таблице G.77, для базы данных, указанной в *`dboid`*.

`pgpro_stats_vacuum_tables(dboid oid, relid oid) returns setof record`

Определяет строку представления `pgpro_stats_vacuum_tables`, которое подробно описано в Таблице G.78, для базы данных, указанной в *`dboid`*, и таблицы, указанной в *`reloid`*. Если *`reloid`* = 0, возвращается статистика для каждой таблицы в указанной базе данных.

`pgpro_stats_vacuum_indexes(dboid oid, relid oid) returns setof record`

Определяет строку представления `pgpro_stats_vacuum_indexes`, которое подробно описано в Таблице G.79, для базы данных, указанной в *`dboid`*, и таблицы, указанной в *`reloid`*. Если *`reloid`* = 0, возвращается статистика для каждого индекса в указанной базе данных.

G.2.6.1. Функции для создания представлений, эмулирующих другие расширения

В расширении pgpro_stats предусмотрена возможность создания представлений, аналогичных тем, которые создаются расширениями pg_stat_statements и *pg_stat_kcache*. Можно создать представления `pg_stat_statements`, `pg_stat_statements_info`, `pg_stat_kcache` и `pg_stat_kcache_detail`. Каждое представление создаётся в версии Postgres Pro, только если оно доступно в расширении pg_stat_statements/pg_stat_kcache для той же версии Postgres Pro/ PostgreSQL. Например, представление `pg_stat_statements_info` создаётся только в версиях Postgres Pro 14 и выше. Эти представления могут быть созданы следующими функциями:

`pgpro_stats_create_pg_stat_statements_compatible_views() returns void`

Создаёт представления `pg_stat_statements` и `pg_stat_statements_info`.

`pgpro_stats_create_pg_stat_kcache_compatible_views() returns void`

Создаёт представления `pg_stat_kcache` и `pg_stat_kcache_detail`.

По умолчанию эти функции могут выполнять только суперпользователи. Другим пользователям можно дать доступ к ним, используя `GRANT`.

Для создания представлений `pg_stat_statements*` необходимо удалить расширение pg_stat_statements, если оно было установлено ранее, и вызвать функцию:

```
select pgpro_stats_create_pg_stat_statements_compatible_views();
```

Для создания представлений `pg_stat_kcache*` необходимо удалить расширение pg_stat_kcache, если оно было установлено ранее, и вызвать функцию:

```
select pgpro_stats_create_pg_stat_kcache_compatible_views();
```

После этого с созданными представлениями можно работать так же, как если бы были установлены соответствующие расширения.

Удалить ранее созданные представления можно обычным способом:

```
drop view pg_stat_statements;
drop view pg_stat_statements_info;
drop view pg_stat_kcache;
drop view pg_stat_kcache_detail;
```

G.2.7. Параметры конфигурации

G.2.7.1. Основные параметры

`pgpro_stats.max` (`integer`)

pgpro_stats собирает статистику только о наиболее часто встречающихся запросах. Чем меньше частота появления некоторого запроса во время работы сервера, тем меньше вероятность его попадания в статистику. Редко возникающие запросы будут почти сразу вытесняться из статистики более частыми. Параметр `pgpro_stats.max` задаёт максимальное число различных уникальных пар (нормализованный текст запроса, нормализованный текст плана), отслеживаемых pgpro_stats (то есть максимальное число строк в представлении `pgpro_stats_statements`). Чем выше значение этого параметра, тем о большем числе запросов информация сохранится, но ценой снижения производительности сервера как во время ожидании на блокировках при работе с таблицей статистики в разделяемой памяти, так и при периодической сборке мусора в файле с текстами запросов и планов. Значение по умолчанию — 5000. Этот параметр можно задать только при запуске сервера.

`pgpro_stats.max_totals` (`integer`)

Параметр `pgpro_stats.max_totals` задаёт максимальное число объектов, отслеживаемых модулем (то есть максимальное число строк в представлении `pgpro_stats_totals`). Когда количество наблюдаемых объектов превышает заданное, информация о наименее активно используемых объектах удаляется. Значение по умолчанию — 1000. Этот параметр можно задать только при запуске сервера.

`pgpro_stats.track` (enum)

Параметр `pgpro_stats.track` определяет, какие операторы будут отслеживаться модулем. Со значением `top` отслеживаются операторы верхнего уровня (те, что непосредственно выполняются клиентами), со значением `all` также отслеживаются вложенные операторы (например, операторы, вызываемые внутри функций) с уровнем вложенности не более 100, значение `none` полностью отключает сбор статистики по операторам. Значение по умолчанию — `top`. Изменять этот параметр могут только суперпользователи.

`pgpro_stats.track_utility` (boolean)

Параметр `pgpro_stats.track_utility` определяет, будет ли этот модуль отслеживать служебные команды. Служебными командами считаются команды, отличные от `SELECT`, `INSERT`, `UPDATE` и `DELETE`. Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.track_planning` (boolean)

Параметр `pgpro_stats.track_planning` определяет, будет ли этот модуль отслеживать операции планирования и их длительность. Включение этого параметра может привести к заметному снижению производительности, особенно когда во множестве одновременных сеансов выполняются операторы с одинаковой структурой запросов, в результате чего эти сеансы пытаются одновременно изменить одни и те же записи в `pgpro_stats_statements`. Значение по умолчанию — `off` (выкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.track_totals` (boolean)

Параметр `pgpro_stats.track_totals` определяет, будет ли агрегироваться статистика по объектам (на уровне кластера, баз, пользователей и т. д.). Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.track_cluster` (boolean)

Параметр `pgpro_stats.track_cluster` определяет, будет ли агрегироваться статистика по кластеру в целом. Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.track_databases` (boolean)

Параметр `pgpro_stats.track_databases` определяет, будет ли агрегироваться статистика по базам данных. Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.track_users` (boolean)

Параметр `pgpro_stats.track_users` определяет, будет ли агрегироваться статистика по пользователям. Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.track_applications` (boolean)

Параметр `pgpro_stats.track_applications` определяет, будет ли агрегироваться статистика по приложениям (по именам, задаваемым в application_name). Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.track_client_addr` (boolean)

Параметр `pgpro_stats.track_client_addr` определяет, будет ли агрегироваться статистика по IP-адресам клиентов. Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.track_backends` (`boolean`)

Параметр `pgpro_stats.track_backends` определяет, будет ли агрегироваться статистика по обслуживающим процессам. Значение по умолчанию — `on` (вкл.). Изменить этот параметр могут только суперпользователи.

`pgpro_stats.save` (`boolean`)

Параметр `pgpro_stats.save` определяет, должна ли статистика операторов сохраняться после перезагрузки сервера. Если он отключён (имеет значение `off`), статистика не сохраняется при остановке сервера и не перезагружается при запуске. Значение по умолчанию — `on` (вкл.). Этот параметр можно задать только в `postgresql.conf` или в командной строке сервера.

`pgpro_stats.plan_format` (`text`)

Параметр `pgpro_stats.plan_format` выбирает формат плана запроса в выводе `EXPLAIN`. Возможные значения: `text`, `xml`, `json` и `yaml`. Значение по умолчанию: `text`. При изменении этого параметра требуется перезапустить сервер.

`pgpro_stats.enable_profile` (`boolean`)

Параметр `pgpro_stats.enable_profile` включает профилирование событий ожидания по отдельным операторам. Значение по умолчанию: `true` (вкл.). При изменении этого параметра требуется перезапустить сервер.

`pgpro_stats.query_sample_rate` (`float`)

Так как pgpro_stats собирает статистику только о наиболее часто встречающихся запросах, а редко встречающиеся запросы в неё не попадают, то есть возможность снизить влияние сбора статистики на производительность сервера. Это выполнимо, потому что через некоторое время после начала сбора практически вся статистика, собранная модулем, будет состоять из запросов, частота появления которых не очень мала, и в дальнейшем множество типов собранных запросов будет меняться слабо, если установлено достаточное значение параметра pgpro_stats.max, а средняя частота появления основной массы частых запросов не претерпевает значительных изменений с периодами, которые превышают время наблюдения. К такому "устоявшемуся" множеству собранных запросов также можно прийти, увеличив время сбора, например, в два раза, но при этом забирая в статистику не каждый запрос, а каждый второй, или же рассматривая каждый запрос, но забирая его в статистику случайно, с некоторой заданной вероятностью. В данном случае — 0.5. Это позволяет заметно снизить нагрузку на сервер за счёт увеличения времени сбора статистики. Реализовать данный подход можно при помощи параметра `pgpro_stats.query_sample_rate`. Например, при значении `pgpro_stats.query_sample_rate`, равном 0.2 (1/5), за 5 часов работы сервера при выполнении указанных выше условий будет собрана такая же статистика, как и при `pgpro_stats.query_sample_rate` = 1 за 1 час работы.

Можно экспериментально подобрать значение параметра `pgpro_stats.query_sample_rate`. Выберите время наблюдения в зависимости от конкретных условий, в которых работает сервер. Например, один час. Посмотрите собранную статистику через час, а затем, не сбрасывая её, ещё через час. Если разница будет в основном не в типе собранных запросов, а в увеличении их количества (`calls`), то можно безопасно уменьшить значение `pgpro_stats.query_sample_rate` в два раза. Если же будет наблюдаться заметная разница в типах собранных запросов, то нужно увеличить время наблюдения и/или параметр `pgpro_stats.max`.

Значение по умолчанию: `1.0`. При изменении этого параметра требуется перезапустить сервер.

`pgpro_stats.profile_period` (`integer`)

Параметр `pgpro_stats.profile_period` задаёт интервал (в миллисекундах) для сбора информации о событиях ожидания. Значение по умолчанию: `10`. Изменить этот параметр могут только суперпользователи.

`pgpro_stats.metrics_buffer_size` (`integer`)

Параметр `pgpro_stats.metrics_buffer_size` задаёт размер кольцевого буфера для сбора статистических метрик. Значение по умолчанию: `16kB`. При изменении этого параметра требуется перезапустить сервер.

`pgpro_stats.metrics_workers` (`integer`)

Параметр `pgpro_stats.metrics_workers` задаёт количество рабочих процессов, которые будут собирать статистические метрики. Если этот параметр больше или равен 2, один из рабочих процессов будет главным, распределяющим запросы между всеми остальными. Если количество рабочих процессов ограничивается одним, он будет перезапускаться для подключения к разным базам данных. Когда этот параметр равен `0`, сбор метрик отключается. Значение по умолчанию: `2`. При изменении этого параметра требуется перезапустить сервер.

`pgpro_stats.stats_temp_directory` (`string`)

Параметр `pgpro_stats.stats_temp_directory` задаёт каталог с внешним файлом для хранения текстов запросов. Это может быть путь от каталога данных или абсолютный путь. При изменении этого параметра требуется перезапустить сервер.

G.2.7.2. Параметры, определяющие метрики

Следующие параметры предназначены для определения дополнительных метрик, которые будет собирать `pgpro_stats`. Номер *`N`* в имени параметра служит уникальным идентификатором метрики, к которой будет применяться задаваемое значение; он должен быть положительным и уникальным среди всех метрик.

Если вы добавляете параметры для определения новой метрики, необходимо перезапустить сервер, чтобы это определение начало действовать. Когда метрика уже добавлена, её параметры можно изменять без перезапуска сервера, просто перезагружая файл конфигурации `postgresql.conf`.

`pgpro_stats.metric_N_name` (`text`)

Имя метрики с номером *`N`*. Это имя будет выводиться в столбце `metric_name` представления `pgpro_stats_metrics`.

`pgpro_stats.metric_N_query` (`text`)

Текст запроса, определяющий значение собираемой метрики.

`pgpro_stats.metric_N_period` (`integer`)

Интервал, с которым будет собираться метрика *`N`*, в миллисекундах. Значение по умолчанию: 60000 ms

`pgpro_stats.metric_N_db` (`text`)

Список баз данных, в которых будет собираться метрика *`N`*. Имена баз данных должны разделяться запятыми. Чтобы выбрать все базы данных в кластере, за исключением баз-шаблонов, можно указать значение `*`. Если вы также хотите анализировать запросы в базах-шаблонах, их нужно перечислить явно.

`pgpro_stats.metric_N_user` (`text`)

Имя пользователя, от имени которого будет собираться метрика *`N`*. Этот пользователь должен иметь доступ к базе данных, в которой будет выполняться запрос для сбора метрики.

G.2.8. Трассировка сеансов приложений

В расширении pgpro_stats реализована трассировка сеансов приложений. Она основана на фильтрах, которые запускают протоколирование выполнения запросов, соответствующих условиям фильтрации. Запросы и вывод `EXPLAIN` для них записываются в так называемые *файлы трассировки*, указанные пользователем, или в файл системного журнала (если файл трассировки не указан). Фильтры хранятся в таблице, находящейся в общей памяти. Строки этой таблицы соответствуют фильтрам, а столбцы содержат условия фильтрации и параметры, с которыми вызывается `EXPLAIN` (за подробностями обратитесь к EXPLAIN). Чтобы начать трассировку запросов, следует заполнить эту таблицу фильтрами.

Как только администратор базы данных добавит фильтр в каком-либо сеансе, все последующие выполнения запросов, соответствующих условиям фильтра, будут трассироваться всеми сеансами экземпляра СУБД без необходимости перезапуска сервера. Другими словами, фильтры можно добавлять, удалять или изменять «на лету», и трассировка с этими фильтрами сразу же начнётся для существующих и будущих сеансов.

Примечание

Операции добавления, удаления или изменения фильтров не являются транзакционными. Изменения таблицы фильтров не отменяются вместе с транзакцией, в которой эти изменения были сделаны.

Столбцы таблицы, в которой хранятся фильтры, также называются *атрибутами фильтров*. Они показаны в Таблице G.81.

Таблица G.81. Атрибуты фильтров

Name	Тип	Описание
`filter_id`	`integer`	Уникальный идентификатор фильтра, который автоматически присваивается фильтру при его создании, нумерация начинается с 1.
`active`	`boolean`	При значении `false` фильтр не активен, то есть запросы, соответствующие этому фильтру, не регистрируются. По умолчанию: `true`.
`alias`	`name`	Имя фильтра, которое можно указать для удобства. Показывается только в выводе функции `pgpro_stats_trace_show()`. По умолчанию: пустая строка.
`tracefile`	`name`	Имя файла трассировки, в котором регистрируются запросы, соответствующие фильтру, и вывод `EXPLAIN` для них. Если это имя не указано, запись ведётся в системный журнал с уровнем, указанным в pgpro_stats.trace_log_level. Файлы трассировки создаются в каталоге `PGDATA/pg_stat` и имеют расширение `trace`.
`pid`	`integer`	Идентификатор обслуживающего процесса, который выполняет этот оператор
`database_name`	`name`	Имя базы данных, в которой выполняется оператор

Name	**Тип**	**Описание**
`client_addr`	`name`	IP-адрес клиента, подключённого к этому обслуживающему процессу
`application_name`	`name`	Имя приложения, вызвавшего выполнение оператора
`username`	`name`	Имя пользователя, выполняющего оператор
`queryid`	`bigint`	Внутренний хеш-код, вычисленный для дерева разбора оператора
`planid`	`bigint`	Внутренний хеш-код, вычисленный для дерева плана оператора
`duration`	`float8`	Время, затраченное на планирование и выполнение оператора, в миллисекундах
`plan_time`	`float8`	Время, затраченное на планирование оператора, в миллисекундах
`exec_time`	`float8`	Время, затраченное на выполнение оператора, в миллисекундах
`user_time`	`float8`	Время использования процессора в пользовательском режиме, затраченное на планирование и выполнение оператора
`system_time`	`float8`	Время использования процессора в режиме ядра, затраченное на планирование и выполнение оператора
`rows`	`int8`	Общее число строк, полученных или затронутых оператором
`shared_blks_hit`	`int8`	Общее число попаданий разделяемых блоков в кеш для данного оператора
`shared_blks_read`	`int8`	Общее число разделяемых блоков, прочитанных данным оператором
`shared_blks_fetched`	`int8`	Общее число разделяемых блоков, прочитанных данным оператором из буферов
`shared_blks_dirtied`	`int8`	Общее число разделяемых блоков, «загрязнённых» данным оператором
`shared_blks_written`	`int8`	Общее число разделяемых блоков, записанных данным оператором

Name	Тип	Описание
`local_blks_hit`	`int8`	Общее число попаданий локальных блоков в кеш для данного оператора
`local_blks_read`	`int8`	Общее число локальных блоков, прочитанных данным оператором
`local_blks_fetched`	`int8`	Общее число локальных блоков, прочитанных данным оператором из буферов
`local_blks_dirtied`	`int8`	Общее число локальных блоков, «загрязнённых» данным оператором
`local_blks_written`	`int8`	Общее число локальных блоков, записанных данным оператором
`temp_blks_read`	`int8`	Общее число временных блоков, прочитанных данным оператором
`temp_blks_written`	`int8`	Общее число временных блоков, записанных данным оператором
`wal_bytes`	`numeric`	Общий объём WAL (в байтах), сгенерированный при выполнении оператора
`total_wait_time`	`float8`	Общее время, затраченное на ожидание при выполнении этого оператора
`total_inval_msgs`	`bigint`	Общее число сообщений аннулирования кеша, выданных при выполнении оператора (если это поддерживает сервер)
`explain_analyze`	`boolean`	При значении true вывод `EXPLAIN` будет протоколироваться с параметром `ANALYZE`. Значение по умолчанию определяется параметром конфигурации pgpro_stats.explain_analyze_default.
`explain_verbose`	`boolean`	При значении true вывод `EXPLAIN` будет протоколироваться с параметром `VERBOSE`. Значение по умолчанию определяется параметром конфигурации pgpro_stats.explain_verbose_default.
`explain_costs`	`boolean`	При значении true вывод `EXPLAIN` будет протоколироваться с параметром `COSTS`. Значение по умолчанию определяется параметром конфи-

Name	**Тип**	**Описание**
		гурации pgpro_stats.explain_costs_default.
`explain_settings`	`boolean`	При значении true вывод `EXPLAIN` будет протоколироваться с параметром `SETTINGS`. Значение по умолчанию определяется параметром конфигурации pgpro_stats.explain_settings_default.
`explain_buffers`	`boolean`	При значении true вывод `EXPLAIN` будет протоколироваться с параметром `BUFFERS`. Значение по умолчанию определяется параметром конфигурации pgpro_stats.explain_buffers_default.
`explain_wal`	`boolean`	При значении true вывод `EXPLAIN` будет протоколироваться с параметром `WAL`. Значение по умолчанию определяется параметром конфигурации pgpro_stats.explain_wal_default.
`explain_timing`	`boolean`	При значении true вывод `EXPLAIN` будет протоколироваться с параметром `TIMING`. Значение по умолчанию определяется параметром конфигурации pgpro_stats.explain_timing_default.
`explain_format`	`text`	Значение параметра `FORMAT` протоколируемой команды `EXPLAIN`, которое может быть TEXT, XML, JSON или YAML. Значение по умолчанию определяется параметром конфигурации pgpro_stats.plan_format.
`time_info`	`boolean`	При значении true добавляет дополнительную информацию в вывод трассировки сеанса в формате, установленном атрибутом `explain_format` : `pid`, `duration` в миллисекундах, `start time` — время начала выполнения запроса и `stop time` — время окончания выполнения запроса. Вывод трассировки с дополнительной информацией показан в Примере G.2.

Атрибуты фильтров можно разделить на следующие группы:

- Управляющие атрибуты, от `filter_id` до `tracefile` в Таблица G.81.
- Идентифицирующие атрибуты, от `pid` до `planid` в Таблица G.81.

- Пороговые атрибуты, от `duration` до `total_inval_msgs` в Таблица G.81.
- Атрибуты, которые управляют выводом в файл трассировки или файл системного журнала и не влияют на фильтрацию. В основном они определяют параметры `EXPLAIN`. Это атрибуты, начинающиеся с `explain_analyze` и до конца Таблицы G.81.

Выполнение запроса будет трассироваться, если выполняются оба следующих условия:

- Для всех указанных идентифицирующих атрибутов идентификации фильтра значения соответствующих характеристик запроса совпадают со значениями этих атрибутов.
- Для всех указанных пороговых атрибутов фильтра значения соответствующих характеристик запроса больше или равны пределам, указанным в этих атрибутах.

Можно не присваивать значения атрибутам. Для всех атрибутов, кроме `active`, можно также явно присвоить значение NULL. Если значение управляющего атрибута не определено или явно задан NULL, этому атрибуту будет присвоено значение по умолчанию, показанное в Таблица G.81. Для идентифицирующих и пороговых атрибутов неопределённые значения или NULL означают, что запросы не будут фильтроваться по соответствующей характеристике. Если для всех идентифицирующих и пороговых атрибутов не определено ни одного значения, все запросы считаются соответствующими этому фильтру и будут регистрироваться. Если атрибуты, задающие параметры `EXPLAIN`, не определены, будут использоваться значения по умолчанию, определённые параметрами конфигурации (за подробностями обратитесь к Подраздел G.2.8.2).

> **Предупреждение**
>
> Хотя при задании фильтра можно устанавливать значения для любой комбинации атрибутов фильтра, имейте в виду, что слишком общий фильтр приведёт к чрезмерному увеличению размера файла трассировки и заметно повлияет на производительность, поскольку основные издержки связаны с записью в файл трассировки, а не с проверкой условий фильтра.

G.2.8.1. Функции трассировки сеансов

Специальные функции позволяют создавать, изменять и удалять фильтры запросов:

`pgpro_stats_trace_insert(VARIADIC "any") returns integer`

Добавляет фильтр в список фильтров трассировки сеанса. Фильтр должен быть передан как последовательность чередующихся пар ключ-значение, где ключ — это имя атрибута фильтра из Таблица G.81, за исключением `filter_id`. Например:

```
pgpro_stats_trace_insert('pid', 42, 'database_name', 'main', 'explain_analyze',
 true)
```

Пары ключ-значение позиционно независимы: они могут быть указаны в любом порядке, независимо от порядка атрибутов в Таблица G.81. Некоторые атрибуты могут отсутствовать в последовательности. Таким атрибутам будут назначены значения по умолчанию. Возвращает уникальный `filter_id`, который функция назначает добавленному фильтру.

`pgpro_stats_trace_update(filter_id integer, VARIADIC "any") returns boolean`

Изменяет фильтр трассировки сеанса, определённый *`filter_id`*. Атрибуты фильтра для изменения должны передаваться как последовательность чередующихся пар ключ-значение, где ключ — это имя атрибута фильтра из Таблица G.81, за исключением `filter_id`. Пары ключ-значение позиционно независимы: они могут быть указаны в любом порядке, независимо от порядка атрибутов в Таблица G.81. Атрибуты фильтра, значения которых не указаны явно, не будут изменены. Значение NULL для идентифицирующих и пороговых атрибутов сбрасывает значения этих атрибутов, что означает, что запрос будет соответствовать условиям изменённого фильтра независимо от соответствующих характеристик. Возвращает `true` в случае успеха и `false` в противном случае.

`pgpro_stats_trace_delete(filter_id integer) returns boolean`

Удаляет фильтр трассировки сеанса, заданный параметром *`filter_id`*. При успешном удалении возвращает `true`, в противном случае — `false`.

`pgpro_stats_trace_reset() returns integer`

Удаляет все фильтры трассировки сеансов. Возвращает количество удалённых фильтров. По умолчанию эту функцию могут выполнять только суперпользователи. Другим пользователям можно дать доступ к ней, используя `GRANT`.

`pgpro_stats_trace_show() returns setof record`

Отображает содержимое таблицы фильтров, столбцы которой показаны в Таблице G.81.

G.2.8.2. Параметры конфигурации трассировки сеансов

Следующие параметры могут использоваться для конфигурации протоколирования трассировки сеансов. Обратите внимание, что параметры `pgpro_stats.explain_*_default` определяют поведение протоколирования, когда соответствующие атрибуты фильтра `explain_*` (из Таблица G.81) не указаны или для них явно заданы значения NULL.

`pgpro_stats.explain_analyze_default` (`boolean`)

При значении `true` вывод `EXPLAIN` будет протоколироваться с параметром `ANALYZE`, в противном случае (`false`) — без него. По умолчанию: `false`. Изменить этот параметр могут только суперпользователи.

`pgpro_stats.explain_verbose_default` (`boolean`)

При значении `true` вывод `EXPLAIN` будет протоколироваться с параметром `VERBOSE`, в противном случае (`false`) — без него. По умолчанию: `false`. Изменить этот параметр могут только суперпользователи.

`pgpro_stats.explain_costs_default` (`boolean`)

При значении `true` вывод `EXPLAIN` будет протоколироваться с параметром `COSTS`, в противном случае (`false`) — без него. По умолчанию: `true`. Изменить этот параметр могут только суперпользователи.

`pgpro_stats.explain_settings_default` (`boolean`)

При значении `true` вывод `EXPLAIN` будет протоколироваться с параметром `SETTINGS`, в противном случае (`false`) — без него. По умолчанию: `false`. Изменить этот параметр могут только суперпользователи.

`pgpro_stats.explain_buffers_default` (`boolean`)

При значении `true` вывод `EXPLAIN` будет протоколироваться с параметром `BUFFERS`, в противном случае (`false`) — без него. По умолчанию: `false`. Изменить этот параметр могут только суперпользователи.

`pgpro_stats.explain_wal_default` (`boolean`)

При значении `true` вывод `EXPLAIN` будет протоколироваться с параметром `WAL`, в противном случае (`false`) — без него. По умолчанию: `false`. Изменить этот параметр могут только суперпользователи.

`pgpro_stats.explain_timing_default` (`boolean`)

При значении `true` вывод `EXPLAIN` будет протоколироваться с параметром `TIMING`, в противном случае (`false`) — без него. По умолчанию: `true`. Изменить этот параметр могут только суперпользователи.

`pgpro_stats.trace_log_level` (`enum`)

Определяет уровень детализации, с которым вывод `EXPLAIN` будет записываться в системный файл журнала (если файл трассировки не указан). Допустимые значения: `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING` и `LOG`. Значение по умолчанию: `LOG`.

`pgpro_stats.trace_query_text_size` (`int`)

Определяет максимальный размер текста запроса в выводе трассировки сеансов, в килобайтах. Если размер текста запроса в килобайтах больше, запрос обрезается до максимального размера. Если параметр не задан, используется значение по умолчанию ноль, которое означает, что в вывод включаются полные тексты запросов.

G.2.8.3. Примеры, относящиеся к трассировке сеансов

Пример G.1. Использование функций трассировки сеансов

Добавьте фильтр `first`:

```
SELECT pgpro_stats_trace_insert('alias', 'first', 'pid', pg_backend_pid(),
 'explain_analyze', true);
```

Добавьте фильтр `second` и укажите протоколирование в файл трассировки `second_tf.trace`:

```
SELECT pgpro_stats_trace_insert('alias', 'second', 'database_name', current_database(),
 'explain_costs', false, 'tracefile', 'second_tf');
```

Таблицу с фильтрами можно посмотреть следующим образом:

```
\x auto
SELECT * from pgpro_stats_trace_show();
```

```
 -[ RECORD 1 ]-------+----------
 filter_id           | 1
 active              | t
 alias               | first
 tracefile           |
 pid                 | 243183
 database_name       |
 client_addr         |
 application_name    |
 username            |
 queryid             |
 planid              |
 duration            |
 plan_time           |
 exec_time           |
 user_time           |
 system_time         |
 rows                |
 shared_blks_hit     |
 shared_blks_read    |
 shared_blks_fetched |
 shared_blks_dirtied |
 shared_blks_written |
 local_blks_hit      |
 local_blks_read     |
 local_blks_fetched  |
 local_blks_dirtied  |
 local_blks_written  |
 temp_blks_read      |
 temp_blks_written   |
```

```
 wal_bytes             |
 total_wait_time       |
 total_inval_msgs      |
 explain_analyze       | t
 explain_verbose       | f
 explain_costs         | t
 explain_settings      | f
 explain_buffers       | f
 explain_wal           | f
 explain_timing        | t
 explain_format        | text
 time_info             | f
-[ RECORD 2 ]-------+----------
 filter_id             | 2
 active                | t
 alias                 | second
 tracefile             | second_tf
 pid                   |
 database_name         | postgres
 client_addr           |
 application_name      |
 username              |
 queryid               |
 planid                |
 duration              |
 plan_time             |
 exec_time             |
 user_time             |
 system_time           |
 rows                  |
 shared_blks_hit       |
 shared_blks_read      |
 shared_blks_fetched |
 shared_blks_dirtied |
 shared_blks_written |
 local_blks_hit        |
 local_blks_read       |
 local_blks_fetched  |
 local_blks_dirtied  |
 local_blks_written  |
 temp_blks_read        |
 temp_blks_written     |
 wal_bytes             |
 total_wait_time       |
 total_inval_msgs      |
 explain_analyze       | f
 explain_verbose       | f
 explain_costs         | f
 explain_settings      | f
 explain_buffers       | f
 explain_wal           | f
 explain_timing        | f
 explain_format        | text
 time_info             | f
```

Следующий запрос соответствует условиям обоих фильтров, поэтому он должен быть записан в файл системного журнала и в указанный файл трассировки:

```
SELECT 1 as result;
```

Вывод в файл системного журнала выглядит так:

```
-- :: MSK [] LOG:  Filter 1 triggered explain of the plan:
Query Text: SELECT 1 as result;
Result  (cost=0.00..0.01 rows=1 width=4) (actual time=0.003..0.003 rows=1 loops=1)
```

Ниже приведён вывод файла трассировки `second_tf.trace`:

```
Query Text: SELECT 1 as result;
Result
```

Удалите первый фильтр:

```
SELECT pgpro_stats_trace_delete(1);
```

Также измените значение `pid` на 2 для второго фильтра

```
SELECT pgpro_stats_trace_update(2, 'pid', 2);
```

При выполнении запроса

```
SELECT 2 as result;
```

он не записывается в `second_tf.trace`.

Удалите все фильтры из таблицы:

```
SELECT pgpro_stats_trace_reset();
```

Пример G.2. Вывод трассировки сеанса с `time_info` = `true`

Следующий пример показывает вывод трассировки сеанса в формате JSON при `time_info` = `true`:

```
{
  "Pid": 123456,
  "Duration ms": 0.094691,
  "Start time": "2025-01-12 15:39:27.580447+03",
  "Stop time": "2025-01-12 15:39:27.587802+03",
  "Query Text": "SELECT 2 AS Result;",
  "Plan": {
    "Node Type": "Result",
    "Parallel Aware": false,
    "Async Capable": false,
    "Startup Cost": 0.00,
    "Total Cost": 0.01,
    "Plan Rows": 1,
    "Plan Width": 4
  }
}
```

G.2.9. Метрики аннулирования кеша

Помимо прочего, pgpro_stats может собирать статистику аннулирования кеша. Представленная в этом разделе общая информация поможет лучше понимать показатели, связанные с этой статистикой.

У каждого обслуживающего процесса есть свой локальный кеш, что позволяет минимизировать обращения к метаинформации таблиц, в частности к системным каталогам. Если какой-либо обслуживающий процесс изменяет метаинформацию некоторого объекта, она должна обновиться в кешах других обслуживающих процессов. Для этого первый обслуживающий процесс отправляет сообщение аннулирования в специальную очередь для таких сообщений.

Все обслуживающие процессы получают сообщения аннулирования из очереди. Если объект, для которого получено сообщение аннулирования, находится в кеше, его метаданные в кеше обновляются, иначе это сообщение просто игнорируется. В pgpro_stats значения большинства счётчиков сообщений аннулирования, если явно не указано иное, увеличиваются, когда обслуживающие процессы только генерируют сообщения, которые попадут в очередь после фиксации соответствующей транзакции. Обратите внимание, что при откате транзакции счётчики не уменьшаются, хотя сообщения в этом случае не доходят до очереди и аннулирование не производится.

Когда обслуживающий процесс, добавляющий сообщения в очередь, обнаруживает, что размер очереди достиг некоторого предела, он запускает очистку, удаляя сообщения, уже обработанные всеми обслуживающими процессами. Если какие-то обслуживающие процессы не успевают обрабатывать сообщения и, таким образом, задерживают очистку, им передаётся сигнал, чтобы они сбросили свои кеши.

G.2.9.1. Представление `pgpro_stats_inval_status`

Представление `pgpro_stats_inval_status` содержит одну строку, показывающую текущее состояние глобальной очереди аннулирования кеша. Столбцы представления показаны в Таблице G.82.

Таблица G.82. Столбцы `pgpro_stats_inval_status`

Name	Тип	Описание
`num_inval_messages`	`int8`	Текущее число сообщений аннулирования в очереди
`num_inval_queue_cleanups`	`int8`	Количество очисток очереди аннулирования, выполненных во избежание её переполнения
`num_inval_queue_resets`	`int4`	Число операций сброса кеша для обслуживающих процессов, не успевающих обрабатывать сообщения достаточно быстро

В работающей системе `num_inval_messages` обычно имеет значение около 4000, то есть очередь почти полностью заполнена. Увеличение `num_inval_queue_cleanups` говорит о том, что в очередь поступает много сообщений аннулирования кеша. Обычно счётчик `num_inval_queue_resets` не увеличивается, а его увеличение свидетельствует о слишком быстром поступлении таких сообщений или о задержках в их обработке со стороны обслуживающих процессов. Отслеживание значений `num_inval_queue_cleanups` и `num_inval_queue_resets` в некоторых случаях может позволить обнаружить проблемные обслуживающие процессы, как описано ниже.

Если на протяжении некоторого времени счётчик `num_inval_queue_cleanups` заметно увеличивается, а `num_inval_queue_resets` — нет, это говорит о том, что обслуживающие процессы обрабатывают сообщения аннулирования медленнее, чем они генерируются, но всё же достаточно быстро для того, чтобы очередь не переполнялась.

Если на протяжении некоторого времени счётчик `num_inval_queue_cleanups` значительно не увеличивается, а счётчик `num_inval_queue_resets` — увеличивается, это определённо указывает на задержку обработки сообщений обслуживающими процессами. Какие именно обслуживающие процессы виноваты в задержке, позволяет определить столбец `cache_resets` представления `pgpro_stats_totals`.

Если же на протяжении некоторого времени оба счётчика значительно увеличиваются, это указывает на то, что обслуживающие процессы обрабатывают сообщения аннулирования настолько медленно, что происходит переполнение очереди. Какие именно процессы задерживают обработку сообщений, позволяет определить столбец `cache_reset` представления `pgpro_stats_total`. Что является причиной увеличения значения `num_inval_queue_resets`, большая скорость генерирования сообщений или слишком медленная их обработка, непосредственно определить нельзя. Однако можно проанализировать счётчик `total` из представления `pgpro_stats_inval_msgs`. Если в данном интервале времени счётчик изменился примерно на ту же величину, что и в предыдущем интервале той же длительности, увеличение счётчиков определённо вызвано задержками обслуживающих процессов.

Представление `pgpro_stats_inval_status` можно определить с помощью функции:

```
pgpro_stats_inval_status() returns record
```

Определяет представление `pgpro_stats_inval_status`, описанное подробно в Таблице G.82.

G.2.9.2. Тип `pgpro_stats_inval_msgs`

Представления `pgpro_stats_statements` и `pgpro_stats_totals` для каждого соответствующего объекта показывают запись типа `pgpro_stats_inval_msgs` со счётчиками сообщений аннулирования кеша. Поля этой записи показаны в Таблице G.83.

Таблица G.83. Поля `pgpro_stats_inval_msgs`

Name	**Тип**	**Описание**
`total`	`bigint`	Общее число сообщений аннулирования
`catcache`	`bigint`	Число сообщений избирательного аннулирования кеша каталога
`catalog`	`bigint`	Число сообщений полного аннулирования кеша каталога
`relcache`	`bigint`	Число сообщений избирательного аннулирования кеша отношений
`relcache_all`	`bigint`	Число сообщений полного аннулирования кеша отношений
`smgr`	`bigint`	Число сообщений, вызывающих аннулирование открытых файлов отношений. Увеличивается, когда сообщения попадают в очередь.
`relmap`	`bigint`	Число сообщений аннулирования кеша карты отношений. Увеличивается, когда сообщения отправляются в очередь.
`snapshot`	`bigint`	Число сообщений аннулирования снимков каталога

G.2.10. Авторы

Postgres Professional, Москва, Россия

Приложение H. Сторонние модули и расширения, поставляемые в виде отдельных пакетов

В этом приложении содержится информация о сторонних модулях и расширениях, которые доступны в Postgres Pro Standard как отдельные пакеты. Они перечислены в Подразделе 16.1.4.1, а в документации каждого модуля объясняется, какой пакет выбрать для его установки. В Приложении F и Приложении G описаны прочие модули и расширения, включённые в дистрибутив Postgres Pro Standard. Также обратите внимание на следующие расширения, которые поставляются в виде отдельных пакетов: orafce, pgvector, pldebugger и plv8 (за подробностями обратитесь к Таблице 16.3).

H.1. oracle_fdw — доступ к базам данных Oracle

Модуль oracle_fdw – это расширение Postgres Pro, которое предоставляет обёртку сторонних данных для простого и эффективного доступа к базам данных Oracle, включая вынесение наружу условий `WHERE` и обращений к столбцам, а также полноценную поддержку `EXPLAIN`.

H.1.1. Установка

oracle_fdw поставляется вместе с Postgres Pro Standard в виде отдельного пакета `oracle-fdw-std-17` (подробные инструкции по установке приведены в Главе 16). Для oracle_fdw требуется Oracle Instant Client версии 19.5.

Для систем на базе RPM (RHEL, SUSE, РЕД ОС, ROSA, Альт Линукс) загрузите RPM-пакет `oracle-instantclient` и установите его с помощью `rpm`. Если вы используете систему на базе Debian (Ubuntu, Astra Linux), нужно либо преобразовать RPM-пакет в Debian-пакет с помощью alien, а затем установить его с помощью dpkg, либо загрузить ZIP-архив с клиентом и извлечь содержимое архива в каталог `/opt/oracle`.

После установки Postgres Pro Standard создайте расширение oracle_fdw:

```
CREATE EXTENSION oracle_fdw;
```

При этом определяются необходимые функции и создаётся оболочка сторонних данных.

Обратите внимание, что версия расширения, выводимая командой psql `\dx` или в системном каталоге `pg_available_extensions`, *не* является установленной версией oracle_fdw. Чтобы узнать версию oracle_fdw, используйте функцию `oracle_diag`.

H.1.2. Внутреннее устройство

oracle_fdw задаёт для параметра `MODULE` сеанса Oracle значение `postgres`, а для параметра `ACTION` — номер внутреннего процесса. Это помогает идентифицировать сеанс Oracle и позволяет отслеживать его с помощью `DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE`.

oracle_fdw использует интерфейс массивов Oracle, чтобы минимизировать взаимодействие клиент-сервер. Размер пакета можно настроить с помощью параметра таблицы prefetch, который по умолчанию равен 50.

Вместо использования `PLAN_TABLE` для построения плана запроса Oracle (что потребовало бы создания соответствующей таблицы в базе данных Oracle), oracle_fdw применяет планы выполнения, хранящиеся в кеше библиотеки. Для этого запрос Oracle *описывается явно*, что инициирует его анализ. Сложность заключается в определении `SQL_ID` и `CHILD_NUMBER` оператора в `V$SQL`, поскольку столбец `SQL_TEXT` ограничен первыми 1000 байтами запроса. Поэтому oracle_fdw добавляет к запросу комментарий с хешем текста запроса. Этот хеш используется для поиска в `V$SQL`. Фактический план выполнения или информация о стоимости извлекается из `V$SQL_PLAN`.

oracle_fdw использует уровень изоляции транзакций `SERIALIZABLE` на стороне Oracle, который соответствует уровню `REPEATABLE READ` в Postgres Pro. Это необходимо для обеспечения согласованности данных, поскольку один запрос Postgres Pro может быть преобразован в несколько запросов к Oracle (например, при соединении вложенным циклом). Однако реализация `SERIALIZABLE` в Oracle имеет некоторые особенности — за подробностями обратитесь к разделу Проблемы.

Транзакция Oracle фиксируется непосредственно перед фиксацией локальной транзакции, поэтому завершённая транзакция Postgres Pro гарантирует завершение транзакции Oracle. Однако остаётся небольшая вероятность, что транзакция Postgres Pro не сможет завершиться, даже если транзакция Oracle уже была зафиксирована. Избежать этого можно только с использованием двухфазных транзакций и менеджера транзакций, что выходит за рамки возможностей обёртки сторонних данных. Подготовленные операторы с участием Oracle не поддерживаются по той же причине.

H.1.3. Простой пример

Ниже представлен пример использования oracle_fdw. Для получения более детальной информации обратитесь к разделам Параметры и Использование. Также следует ознакомиться с документацией по работе со сторонними данными и упомянутыми в ней командами.

В данном примере предполагается, что вы можете подключиться к Oracle, выполнив следующую команду от имени пользователя операционной системы, который может запускать сервер Postgres Pro (например, `postgres`):

```
sqlplus orauser/orapwd@//dbserver.mydomain.com:1521/ORADB
```

Если вы успешно подключились, значит клиент Oracle и среда настроены правильно. Также предполагается, что пакет с расширением oracle_fdw уже установлен (см. раздел Установка).

Необходим доступ к такой таблице:

```
SQL> DESCRIBE oratab
 Name                               Null?    Type
 -------------------------------- -------- ------------
 ID                                 NOT NULL NUMBER(5)
 TEXT                                        VARCHAR2(30)
 FLOATING                           NOT NULL NUMBER(7,2)
```

Настройте oracle_fdw от имени суперпользователя Postgres Pro следующим образом:

```
pgdb=# CREATE EXTENSION oracle_fdw;
pgdb=# CREATE SERVER oradb FOREIGN DATA WRAPPER oracle_fdw
          OPTIONS (dbserver '//dbserver.mydomain.com:1521/ORADB');
```

Можно использовать другие варианты указания имени или локальные соединения, см. описание dbserver ниже.

Рекомендуется работать от имени суперпользователя, только когда это действительно необходимо, поэтому предлагается разрешить обычному пользователю использовать сторонний сервер (это необязательно для работы примера, но рекомендуется):

```
pgdb=# GRANT USAGE ON FOREIGN SERVER oradb TO pguser;
```

Затем можно подключиться к Postgres Pro от имени `pguser` и задать следующие параметры:

```
pgdb=> CREATE USER MAPPING FOR pguser SERVER oradb
          OPTIONS (user 'orauser', password 'orapwd');
```

Чтобы не хранить пароли Oracle, можно использовать внешнюю аутентификацию.

```
pgdb=> CREATE FOREIGN TABLE oratab (
          id        integer           OPTIONS (key 'true')  NOT NULL,
          text      character varying(30),
          floating  double precision  NOT NULL
       ) SERVER oradb OPTIONS (schema 'ORAUSER', table 'ORATAB');
```

Помните, что имя таблицы и имя схемы (последнее необязательно), как правило, пишутся заглавными буквами.

Теперь можно обращаться к этой таблице, как к обычной таблице Postgres Pro.

H.1.4. Использование

H.1.4.1. Разрешения Oracle

Пользователю Oracle потребуется право `CREATE SESSION`, а также доступ к рассматриваемой таблице или представлению. Обратите внимание, что oracle_fdw обращается к таблице Oracle во время планирования запроса, чтобы получить её определение. Это происходит *до* проверки прав для внешней таблицы. Поэтому если вы попытаетесь получить доступ к внешней таблице без соответствующих прав в Postgres Pro, может возникнуть ошибка Oracle. Это ожидаемо и не представляет угрозы безопасности.

Для `EXPLAIN VERBOSE` пользователю также потребуются права `SELECT` для `V$SQL` и `V$SQL_PLAN`.

H.1.4.2. Соединения

oracle_fdw кеширует соединения Oracle, поскольку создавать сеанс Oracle для каждого отдельного запроса слишком дорого. Все соединения автоматически закрываются по завершении сеанса Postgres Pro.

Чтобы закрыть все кешированные соединения Oracle, можно использовать функцию `close_connections`. Это может быть полезно для длительных сеансов, которые время от времени обращаются к сторонним таблицам, чтобы избежать блокировки ресурсов, необходимых для открытого соединения Oracle. Вызывать эту функцию внутри транзакции, изменяющей данные Oracle, нельзя.

H.1.4.3. Столбцы

При определении сторонней таблицы выполняется сопоставление столбцов таблицы Oracle со столбцами Postgres Pro в порядке их определения.

oracle_fdw включит в запрос к Oracle только те столбцы, которые действительно необходимы запросу Postgres Pro.

В таблице Postgres Pro может быть больше или меньше столбцов, чем в таблице Oracle. Если в ней больше столбцов, и эти столбцы используются, выводится предупреждение и возвращаются значения NULL.

Прежде чем выполнить `UPDATE` или `DELETE`, проверьте, что параметр `key` установлен для всех столбцов, входящих в первичный ключ таблицы. Если это условие не соблюдено, выполнение операций приведёт к ошибкам.

H.1.4.4. Типы данных

Столбцы в Postgres Pro должны быть определены с типами данных, которые oracle_fdw может преобразовывать (см. таблицу преобразования ниже). Это ограничение действует только для используемых столбцов, поэтому можно определить «фиктивные» столбцы для непреобразуемых типов данных, пока к ним не производится обращение (этот подход работает только для `SELECT`, но не при изменении сторонних данных). Если значение Oracle превышает размер столбца Postgres Pro (например, длину столбца `varchar` или максимальное значение `integer`), это приведёт к ошибке времени выполнения.

Расширение oracle_fdw автоматически обрабатывает следующие преобразования:

```
Тип Oracle                | Возможные типы PostgreSQL
--------------------------+------------------------------------------------
CHAR                      | char, varchar, text
NCHAR                     | char, varchar, text
VARCHAR                   | char, varchar, text
VARCHAR2                  | char, varchar, text, json
NVARCHAR2                 | char, varchar, text
CLOB                      | char, varchar, text, json
LONG                      | char, varchar, text
RAW                       | uuid, bytea
BLOB                      | bytea
BFILE                     | bytea (read-only)
LONG RAW                  | bytea
NUMBER                    | numeric, float4, float8, char, varchar, text
NUMBER(n,m) with m<=0     | numeric, float4, float8, int2, int4, int8,
                          |    boolean, char, varchar, text
FLOAT                     | numeric, float4, float8, char, varchar, text
BINARY_FLOAT              | numeric, float4, float8, char, varchar, text
BINARY_DOUBLE             | numeric, float4, float8, char, varchar, text
DATE                      | date, timestamp, timestamptz, char, varchar, text
TIMESTAMP                 | date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH TIME ZONE  | date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH            | date, timestamp, timestamptz, char, varchar, text
   LOCAL TIME ZONE        |
INTERVAL YEAR TO MONTH    | interval, char, varchar, text
INTERVAL DAY TO SECOND    | interval, char, varchar, text
XMLTYPE                   | xml, char, varchar, text
MDSYS.SDO_GEOMETRY        | geometry (см. "поддержка PostGIS" ниже)
```

При преобразовании типа `NUMBER` в `boolean` значение 0 интерпретируется как `false`, а любое другое значение — как `true`.

Вставка или изменение данных `XMLTYPE` возможно только для значений, длина которых не превышает максимального размера типа `VARCHAR2` (4000 или 32767, в зависимости от значения параметра `MAX_STRING_SIZE`.

В настоящее время тип `NCLOB` не поддерживается, так как Oracle не выполняет автоматическое преобразование этого типа в клиентскую кодировку.

Если требуется преобразовать `TIMESTAMP WITH LOCAL TIMEZONE` в `timestamp`, рекомендуется установить параметр `set_timezone` на стороннем сервере.

Если требуются преобразования, не указанные выше, определите соответствующее представление в Oracle или Postgres Pro.

H.1.4.5. Условия `WHERE` и предложения `ORDER BY`

Postgres Pro будет использовать все подходящие части предложения `WHERE` в качестве фильтра для сканирования. Запрос Oracle, сформированный oracle_fdw, будет включать предложение `WHERE`, соответствующее этим фильтрам, если такое условие можно без проблем преобразовать в SQL Oracle. Эта функция, известная как *вынесение наружу* предложений `WHERE`, может значительно сократить количество строк, извлекаемых из Oracle, а также помочь оптимизатору Oracle выбрать хороший план доступа к необходимым таблицам.

Предложения `ORDER BY` будут аналогично выноситься наружу в Oracle, когда это возможно. Обратите внимание, что условия `ORDER BY`, сортирующие по строкам символов, не будут выноситься наружу, поскольку невозможно гарантировать, что порядок сортировки в Postgres Pro совпадёт с порядком в Oracle.

Для успешного вынесения наружу таких предложений рекомендуется использовать простые условия для сторонней таблицы. Выбирайте типы данных столбцов Postgres Pro, соответствующие типам Oracle, поскольку в противном случае условия не смогут быть преобразованы.

Выражения `now()`, `transaction_timestamp()`, `current_timestamp`, `current_date` и `localtimestamp` будут преобразованы корректно.

В выводе `EXPLAIN` будет показан запрос, отправленный в Oracle, что позволяет увидеть, какие условия были преобразованы в Oracle и каким образом.

H.1.4.6. Соединения сторонних таблиц

Расширение oracle_fdw поддерживает вынос соединений на сервер Oracle. Это означает, что соединение сторонних таблиц будет выполнено в рамках одного запроса Oracle на стороне Oracle.

Однако для выноса соединений наружу есть следующие ограничения:

- Обе таблицы должны быть определены на одном и том же стороннем сервере.
- Соединения между тремя и более таблицами не выносятся наружу.
- Соединение должно быть в рамках оператора `SELECT`.
- Расширение oracle_fdw должно иметь возможность вынести наружу все условия соединения и все предложения `WHERE`.
- Перекрёстные соединения без условий соединения не выносятся наружу.
- Если соединение передано на сервер, предложения `ORDER BY` не выносятся наружу.

Важно собирать статистику для обеих сторонних таблиц с помощью команды `ANALYZE`, тогда Postgres Pro сможет выбрать наиболее оптимальную стратегию соединения.

H.1.4.7. Изменение сторонних данных

Расширение oracle_fdw поддерживает команды `INSERT`, `UPDATE` и `DELETE` для сторонних таблиц. Они разрешены по умолчанию (как и в базах данных, обновлённых с более ранней версии Postgres Pro) и могут быть отключены в параметре таблицы readonly.

Чтобы `UPDATE` и `DELETE` работали, столбцы, соответствующие столбцам первичного ключа таблицы Oracle, должны быть помечены параметром key. Эти столбцы используются для идентификации строк сторонней таблицы, поэтому убедитесь, что этот параметр установлен для *всех* столбцов первичного ключа.

Если не указать столбец сторонней таблицы во время `INSERT`, для этого столбца устанавливается значение, определённое в предложении `DEFAULT` в сторонней таблице Postgres Pro (или NULL, если нет предложения `DEFAULT`). Предложения `DEFAULT` в соответствующих столбцах Oracle не используются. Если сторонняя таблица Postgres Pro не включает все столбцы таблицы Oracle, предложения `DEFAULT` из Oracle будут использоваться для столбцов, не включённых в определение сторонней таблицы.

Предложение `RETURNING` в командах `INSERT`, `UPDATE` и `DELETE` поддерживается, но не для столбцов с типами данных Oracle `LONG` и `LONG RAW` (Oracle не поддерживает эти типы данных в предложении `RETURNING`).

Триггеры для сторонних таблиц поддерживаются. Триггеры, определённые с помощью `AFTER` и `FOR EACH ROW`, требуют отсутствия в сторонней таблице столбцов с типом данных Oracle `LONG` или `LONG RAW`. Это связано с использованием в таких триггерах предложения `RETURNING`.

Хотя изменение сторонних данных поддерживается, производительность может быть низкой, особенно при обработке большого количества строк. Это связано с особенностями работы обёртки сторонних данных, которая требует обработки каждой строки отдельно.

Транзакции перенаправляются в Oracle, поэтому `BEGIN`, `COMMIT`, `ROLLBACK` и `SAVEPOINT` работают как обычно. Подготовленные операторы с перенаправлением в Oracle не поддерживаются. За подробной информацией обратитесь к разделу Внутреннее устройство.

Поскольку oracle_fdw по умолчанию использует сериализуемые транзакции, операторы изменения данных могут привести к сбою сериализации:

```
ORA-08177: can't serialize access for this transaction
```

Сбой может произойти, если параллельные транзакции изменяют одну и ту же таблицу. Вероятность сбоя возрастает в случае длительных транзакций. Такие сбои можно идентифицировать по коду `SQLSTATE` (40001). Приложение, использующее oracle_fdw, должно повторить транзакции, завершившиеся таким сбоем.

Можно использовать другой уровень изоляции транзакций, за подробностями обратитесь к разделу Параметры внешнего сервера.

H.1.4.8. EXPLAIN

Postgres Pro EXPLAIN показывает запрос, который фактически отправлен в Oracle. `EXPLAIN VERBOSE` показывает план выполнения Oracle (эта возможность недоступна для серверов Oracle 9i или более ранней версии, см. Проблемы).

H.1.4.9. ANALYZE

Для сбора статистик по сторонним таблицам можно использовать ANALYZE, которая поддерживается в oracle_fdw.

Без статистики Postgres Pro не может оценить количество строк, возвращаемых запросами к сторонней таблице, что может привести к выбору неоптимальных планов выполнения.

Postgres Pro *не* будет автоматически собирать статистику для сторонних таблиц с помощью демона автоочистки, как это происходит с обычными таблицами. Поэтому особенно важно запускать `ANALYZE` для сторонних таблиц после их создания и каждый раз, когда удалённая таблица значительно изменяется.

Имейте в виду, что анализ сторонней таблицы Oracle выполняется через полное последовательное сканирование таблицы. Чтобы ускорить процесс, можно задать параметр sample_percent, который позволяет использовать лишь выборку данных из таблицы Oracle.

H.1.4.10. Поддержка PostGIS

Тип данных `geometry` доступен, только если установлено расширение PostGIS.

Поддерживаются следующие геометрические типы данных: `POINT`, `LINE`, `POLYGON`, `MULTIPOINT`, `MULTILINE` и `MULTIPOLYGON` в двухмерном и трёхмерном представлении. Пустые геометрии PostGIS не поддерживаются, поскольку для них нет аналогов в Oracle Spatial.

Значения NULL для `SRID` в Oracle будут преобразованы в 0 и наоборот. Для других преобразований `SRID` Oracle и PostGIS создайте файл `srid.map` в каталоге `share` в Postgres Pro. Каждая строка этого файла должна содержать `SRID` Oracle и соответствующий `SRID` PostGIS, разделённые пробелом. Для обеспечения высокой производительности файл рекомендуется держать небольшим.

H.1.4.11. Поддержка команды `IMPORT FOREIGN SCHEMA`

Команда IMPORT FOREIGN SCHEMA поддерживает массовый импорт определений всех таблиц из схемы Oracle. В дополнение к документации команды `IMPORT FOREIGN SCHEMA` следует учитывать следующие особенности:

- `IMPORT FOREIGN SCHEMA` создаёт внешние таблицы для всех объектов, найденных в `ALL_TAB_COLUMNS`. Сюда входят таблицы, представления и материализованные представления, но не синонимы.
- Для команды `IMPORT FOREIGN SCHEMA` поддерживаются следующие параметры:
  - Переменная `case` управляет регистром для имён таблиц и столбцов во время импорта.

  Возможные значения:

  - `keep`: оставить имена такими же, как в Oracle, обычно в верхнем регистре.
  - `lower`: перевести все имена таблиц и столбцов в нижний регистр.
  - `smart`: перевести только те имена, которые в Oracle полностью написаны в верхнем регистре (это значение по умолчанию).

- `collation` — это правило сортировки, используемое для управления регистром `lower` и `smart` параметра `case`.

  Значение по умолчанию — `default`, которое соответствует правилу сортировки по умолчанию для базы данных. Поддерживаются только правила сортировки в схеме `pg_catalog`. Для получения списка возможных значений обратитесь к полю `collname` в каталоге `pg_collation` .

- `dblink` — это ссылка на базу данных Oracle, через которую осуществляется доступ к схеме.

  Написание этого имени должно быть таким же, как в системном каталоге Oracle, которое обычно состоит только из заглавных букв.

- `readonly` устанавливает параметр readonly для всех импортируемых таблиц.
- `skip_tables` (по умолчанию `false`): не импортировать таблицы.
- `skip_views` (по умолчанию `false`): не импортировать представления.
- `skip_matviews` (по умолчанию `false`): не импортировать материализованные представления.
- `max_long` устанавливает параметр max_long для всех импортируемых таблиц.
- `sample_percent` устанавливает параметр sample_percent для всех импортируемых таблиц.
- `prefetch` устанавливает параметр prefetch для всех импортируемых таблиц.
- `lob_prefetch` устанавливает параметр lob_prefetch для всех импортируемых таблиц.
- `nchar` устанавливает параметр nchar для всех импортируемых таблиц.
- `set_timezone` устанавливает параметр set_timezone для всех импортируемых таблиц.

- Имя схемы Oracle должно быть написано точно так же, как в Oracle, то есть, как правило, в верхнем регистре. Так как Postgres Pro переводит имена в нижний регистр перед обработкой, необходимо заключить имя схемы в двойные кавычки (например, `"SCOTT"`).
- Имена таблиц в предложении `LIMIT TO` или `EXCEPT` должны быть записаны в том виде, в котором они будут отображаться в Postgres Pro после применения правил учёта регистра, описанных выше.

Обратите внимание, что `IMPORT FOREIGN SCHEMA` не поддерживается для серверов Oracle версии 8i; за подробностями обратитесь к разделу Проблемы.

H.1.5. Справка

H.1.5.1. Создаваемые расширением объекты

```
oracle_fdw_handler() RETURNS fdw_handler
oracle_fdw_validator(text[], oid) RETURNS void
```

Эти функции представляют собой функцию-обработчик и функцию проверки, которые необходимы для создания обёртки сторонних данных.

```
FOREIGN DATA WRAPPER oracle_fdw
  HANDLER oracle_fdw_handler
  VALIDATOR oracle_fdw_validator
```

Расширение автоматически создаёт обёртку сторонних данных с именем `oracle_fdw`. Обычно после этого можно перейти к определению сторонних серверов. Можно также создать дополнительные обёртки сторонних данных Oracle, например, если нужно установить параметр nls_lang. Хотя можно изменить существующую обёртку `oracle_fdw`, следует помнить, что в этом случае все изменения будут потеряны после выгрузки/восстановления базы.

`oracle_close_connections() RETURNS void`

Эту функцию можно использовать для закрытия всех открытых соединений Oracle в этом сеансе. За дополнительные сведениями обратитесь к разделу Использование.

`oracle_diag(name DEFAULT NULL) RETURNS text`

Данная функция полезна только для диагностических целей. Она возвращает версии сервера oracle_fdw, Postgres Pro и клиента Oracle. При вызове без аргументов или с NULL также возвращаются значения некоторых переменных среды, используемых для установления соединений с Oracle. При вызове с именем стороннего сервера также возвращается версия сервера Oracle.

`oracle_execute(server name, stmt text) RETURNS void`

Эту функцию можно использовать для выполнения произвольных операторов SQL на удалённом сервере Oracle. Она будет работать только с операторами, которые не возвращают результаты (обычно операторы DDL).

Будьте осторожны при использовании этой функции, поскольку она может повлиять на процесс управления транзакциями oracle_fdw. Помните, что выполнение DDL-операторов в Oracle сопровождается неявной командой `COMMIT`. Не рекомендуется использовать эту функцию в транзакциях с несколькими операторами.

H.1.5.2. Параметры

H.1.5.2.1. Параметры обёртки сторонних данных

> **Важно**
>
> Если вы измените стандартную обёртку сторонних данных `oracle_fdw`, все изменения будут потеряны при выгрузке/восстановлении. Чтобы сохранить параметры, создайте новую обёртку сторонних данных. Установочный скрипт содержит оператор `CREATE FOREIGN DATA WRAPPER`, который можно взять за основу при создании собственной обёртки.

`[nls_lang]`

Устанавливает указанное значение для переменной среды `NLS_LANG` в Oracle. Формат `NLS_LANG` — *`язык_территория.наборсимволов`* (например, `AMERICAN_AMERICA.AL32UTF8`). Значение должно соответствовать кодировке вашей базы данных. Если это значение не установлено, oracle_fdw автоматически определит его, если сможет, и выдаст предупреждение в противном случае. Устанавливайте это значение, только если понимаете, как оно работает. См. также раздел Проблемы.

H.1.5.2.2. Параметры стороннего сервера

`dbserver`

Строка подключения базы данных Oracle для удалённой базы данных, которая может быть любого вида, поддерживаемого Oracle, если клиент Oracle настроен соответствующим образом. Для локальных (`BEQUEATH`) соединений задайте пустую строку.

`[isolation_level]`

Уровень изоляции транзакций, используемый в базе данных Oracle: `serializable`, `read_committed` или `read_only`. Значение по умолчанию — `serializable`.

Обратите внимание, что один оператор Postgres Pro может обращаться к таблице Oracle более одного раза (например, оператор соединения вложенным циклом). Чтобы избежать несогласованности данных в условиях гонки с параллельными транзакциями, уровень изоляции транзакций должен гарантировать стабильность чтения. Это возможно только с уровнями изоляции Oracle `SERIALIZABLE` или `READ ONLY`.

К сожалению, реализация уровня изоляции `SERIALIZABLE` в Oracle имеет ряд недостатков и может приводить к ошибкам сериализации (ORA-08177) в непредвиденных ситуациях, например при вставках в таблицу. Использование уровня `READ COMMITTED` решает эту проблему, но *с риском несогласованности данных*. Если необходимо использовать этот уровень изоляции, следует убедиться, что планы выполнения запросов не предполагают многократного выполнения внешнего сканирования.

`[nchar]`

Со значением `on` для этого параметра будет выбираться более дорогое преобразование символов на стороне Oracle. Это необходимо, когда в таблицах Oracle есть столбцы `NCHAR` или `NVARCHAR2`, содержащие символы, которые не могут быть представлены в наборе символов базы данных Oracle. Значение по умолчанию — `off`.

Установка значения `on` для `nchar` существенно снижает производительность и вызывает ошибки ORA-01461 при использовании операторов `UPDATE` для строк длиной более 2000 байт (или 16383, если `MAX_STRING_SIZE = EXTENDED`). Это проблема вызвана ограничением в Oracle.

`[set_timezone]`

При указании для этого параметра значения `on` часовой пояс сеанса Oracle при подключении будет соответствовать текущему значению параметра `timezone` в Postgres Pro. Это полезно, только если планируется использовать столбцы Oracle типа `TIMESTAMP WITH LOCAL TIME ZONE` и нужно преобразовать их в `timestamp without time zone` в Postgres Pro. Значение по умолчанию — `off`.

Обратите внимание, что если изменить `timezone` после подключения к Oracle, oracle_fdw не изменит часовой пояс сеанса Oracle. В этом случае, чтобы открыть новое соединение при следующем обращении к сторонней таблице, вызовите `oracle_close_connections() RETURNS void`.

Если Oracle не может распознать часовой пояс, попытки соединения завершаются ошибкой вида «ORA-01882: timezone region not found» (ORA-01882: регион часового пояса не найден).

В этом случае либо используйте другое значение `timezone`, либо оставьте значение `off` и задайте для переменной среды `ORA_SDTZ` соответствующее значение в конфигурации среды сервера Postgres Pro.

H.1.5.2.3. Параметры сопоставления пользователей

`user`

Имя пользователя Oracle для сеанса. Если вы не хотите хранить учётные данные Oracle в базе данных Postgres Pro, укажите пустую строку для *внешней аутентификации* (один из простых способов — *внешнее хранилище паролей*).

`password`

Пароль пользователя Oracle.

H.1.5.2.4. Параметры сторонней таблицы

`table`

Имя таблицы Oracle, которое должно быть указано в том же регистре, что и в системном каталоге Oracle, то есть обычно заглавными буквами.

Чтобы определить стороннюю таблицу на основе произвольного запроса Oracle, сделайте значением этого параметра текст запроса, заключённый в круглые скобки, например:

```
OPTIONS (table '(SELECT col FROM tab WHERE val = ''string'')')
```

В этом случае не устанавливайте параметр schema. Команды `INSERT`, `UPDATE` и `DELETE` будут работать со сторонними таблицами, определёнными в простых запросах. Если работать с такими таблицами не нужно (или хочется избежать запутанных сообщений об ошибках Oracle для более сложных запросов), установите для таблицы параметр readonly.

`dblink`

Ссылка на базу данных Oracle, через которую осуществляется доступ к таблице. Ссылка должна быть указана так же, как в системном каталоге Oracle, то есть обычно заглавными буквами.

`[schema]`

Схема (или владелец) таблицы. Используется для доступа к таблицам, владельцем которых не является подключающийся пользователь Oracle. Значение должно быть указано так же, как в системном каталоге Oracle, то есть обычно заглавными буквами.

`[max_long]`

Максимальная длина любых столбцов `LONG`, `LONG RAW` и `XMLTYPE` в таблице Oracle. Возможные значения — целые числа от 1 до 1073741823 (максимальный размер `bytea` в Postgres Pro). Этот объём памяти будет выделен как минимум дважды, так что большие значения потребуют выделения значительных объёмов памяти. Если max_long меньше длины самого длинного получаемого значения, будет выведено сообщение об ошибке «ORA-01406: fetched column value was truncated» (ORA-01406: значение извлечённого столбца было усечено). По умолчанию используется `32767`.

`[readonly]`

Команды `INSERT`, `UPDATE` и `DELETE` поддерживаются только для таблиц, где значение этого параметра отличается от yes/on/true. По умолчанию используется `false`.

`[sample_percent]`

Этот параметр влияет только на выполнение `ANALYZE` и может быть полезен для сокращения времени работы `ANALYZE` с очень большими таблицами.

Значение должно находиться в диапазоне от 0,000001 до 100. Оно определяет процент блоков таблицы Oracle, которые будут выбраны случайным образом для расчёта статистики таблицы Postgres Pro. В Oracle для выборки блоков используется предложение `SAMPLE BLOCK (x)`. Значение по умолчанию: `100`.

`ANALYZE` завершится ошибкой ORA-00933 для таблиц, определённых с помощью запросов Oracle, и может завершиться ошибкой ORA-01446 для таблиц, определённых с помощью сложных представлений Oracle.

`[prefetch]`

Задаёт количество строк, которые будут извлечены за один цикл передачи данных между Postgres Pro и Oracle во время сканирования сторонней таблицы. Значение должно находиться в диапазоне от 1 до 1000, где нулевое значение отключает предварительную выборку. Значение по умолчанию — `50`.

С более высокими значениями производительность может повыситься, но будет использоваться больше памяти на сервере Postgres Pro.

Обратите внимание, что предварительная выборка не выполняется, если таблица Oracle содержит столбцы типа `MDSYS.SDO_GEOMETRY`.

`[lob_prefetch]`

Устанавливает число байт, которые предварительно выбираются для значений `BLOB`, `CLOB` и `BFILE`. Если значения LOB превышают этот размер, потребуются дополнительные циклы передачи данных между Postgres Pro и Oracle, поэтому установка значения, превышающего обычно используемый размер LOB, может повысить производительность. Выбор больших значений для этого параметра увеличит потребление памяти на стороне сервера, но повысит производительность при работе с большими LOB. Значение по умолчанию — `1048576`.

H.1.5.2.5. Параметры столбцов

`[key]`

Если установлено значение yes/on/true, соответствующий столбец сторонней таблицы Oracle считается частью первичного ключа. Чтобы `UPDATE` и `DELETE` работали, необходимо установить этот параметр для всех столбцов, входящих в первичный ключ таблицы. Значение по умолчанию — `false`.

`[strip_zeros]`

Если установлено значение yes/on/true, символы ASCII 0 будут удалены из строки во время передачи. Такие символы допустимы в Oracle, но не в Postgres Pro, поэтому они вызовут ошибку при чтении oracle_fdw. Этот параметр имеет смысл использовать только для столбцов `character`, `character varying` и `text`. Значение по умолчанию — `false`.

H.1.6. Проблемы

H.1.6.1. Кодировка

Символы, хранящиеся в базе данных Oracle, которые не могут быть преобразованы в кодировку базы данных Postgres Pro, будут автоматически заменены на подстановочные символы Oracle, обычно это прямые или обратные вопросительные знаки. При этом не будет выводиться никаких предупреждений или сообщений об ошибках.

Если используется кодировка базы данных Postgres Pro, которая не поддерживается Oracle (в настоящее время это `EUC_CN`, `EUC_KR`, `LATIN10`, `MULE_INTERNAL`, `WIN874` и `SQL_ASCII`), символы, отличные от ASCII, не могут быть конвертированы. В этом случае будет выдано предупреждение, и символы будут заменены на подстановочные, как описано выше.

Можно установить параметр nls_lang внешней оболочки данных, чтобы принудительно использовать определённую кодировку Oracle, но итоговые символы в большинстве случаев будут некорректными и приведут к сообщениям об ошибках Postgres Pro. Вероятно, это может быть полезно только для кодирования `SQL_ASCII`, так что использовать это следует, только если вы точно понимаете, что делаете.

H.1.6.2. Ограниченная функциональность в старых версиях Oracle

Определение системных каталогов Oracle `V$QL` и `V$QL_PLAN` изменилось в Oracle 10.1. Использование `EXPLAIN VERBOSE` со старыми версиями сервера Oracle приведёт к ошибкам, например:

```
ERROR:  error describing query: OCIStmtExecute failed to execute
        remote query for sql_id
DETAIL:  ORA-00904: "LAST_ACTIVE_TIME": invalid identifier
```

Это не планируется исправлять, поскольку расширенная поддержка Oracle 9i отсутствует с 2010 года, и эта функциональность не является существенной.

`IMPORT FOREIGN SCHEMA` выдаёт следующую ошибку с сервером Oracle 8i:

```
ERROR:  error importing foreign schema: OCIStmtExecute failed to execute
        column query
```

```
DETAIL:  ORA-00904: invalid column name
```

Это связано с тем, что в представлении `ALL_TAB_COLUMNS` отсутствует столбец `CHAR_LENGTH`, который был добавлен в Oracle 9i.

H.1.6.3. Библиотеки LDAP

Общая библиотека клиента Oracle поставляется с собственной клиентской LDAP-реализацией, соответствующей *RFC 1823*, поэтому у этих функций те же имена, что и в OpenLDAP. Это приведёт к конфликту имён, если сервер Postgres Pro был сконфигурирован с параметром `--with-ldap`.

Конфликт имён не будет выявлен, поскольку oracle_fdw загружается во время выполнения. Однако если будет вызвана функция LDAP, возникнут проблемы. Как правило, сначала загружается OpenLDAP, поэтому если Oracle вызывает функцию LDAP (например, если используется разрешение имён по правилам *именования каталогов*), произойдёт сбой сервера. Это может привести к появлению в журнале сервера Postgres Pro сообщений подобного рода (например, в Linux):

```
../../../libraries/libldap/getentry.c:29: ldap_first_entry:
 Assertion `( (ld)->ld_options.ldo_valid == 0x2 )' failed.
```

Так как сборка Postgres Pro выполнена с указанием `--with-ldap`, она может работать, пока не используются какие-либо функциональные возможности клиента LDAP в Oracle. На некоторых платформах можно принудительно загрузить общую клиентскую библиотеку Oracle до запуска сервера Postgres Pro (`LD_PRELOAD` в Linux). Затем должны использоваться функции LDAP Oracle. В этом случае Oracle может использовать функциональные возможности LDAP, но использование LDAP из Postgres Pro приведёт к сбою сервера.

Использовать функциональность LDAP одновременно и в Postgres Pro, и в Oracle не допускается.

H.1.6.4. Ошибки сериализации

В Oracle версии 11.2 или выше при вставке первой строки в созданную таблицу Oracle с oracle_fdw возникнет ошибка сериализации.

Это связано с функциональностью Oracle *отложенное создание сегмента*, при котором выделение дискового пространства для новой таблицы откладывается до вставки первой строки. В результате при использовании сериализуемых транзакций возникает ошибка сериализации.

Это не серьёзная проблема. Её можно избежать, либо проигнорировав эту первую ошибку, либо создав таблицу с параметром `SEGMENT CREATION IMMEDIATE`.

Гораздо более серьёзная проблема заключается в том, что одновременные вставки могут иногда вызывать ошибки сериализации, когда разделение страницы индекса происходит одновременно с модифицируемой сериализуемой транзакцией.

Oracle утверждает, что это не ошибка, и предлагаемое решение — повторить транзакцию, которая получила ошибку сериализации.

H.1.6.5. Ошибки Oracle

Это список ошибок Oracle, которые влияют или влияли на oracle_fdw в прошлом.

Ошибка 2728408 может вызвать «ORA-8177 cannot serialize access for this transaction» (невозможно сериализовать доступ для этой транзакции), даже если не предпринимается никаких попыток изменения удалённых данных. Она может возникнуть на сервере Oracle 8.1.7.4 (установите промежуточное исправление 2728408) или сервере Oracle 9.2 (установите набор исправлений 9.2.0.4 или выше).

Известно, что клиент Oracle 21c не поддерживает работу со столбцами `CLOB` (они отображаются пустыми). Нет прямых доказательств, что это ошибка Oracle, но другие версии работают нормально.

H.1.7. Авторы

Лауренц Альбе, с заметным вкладом Винсента Мора из Oslandia и Тацуро Ямады из NTT OSS Center.

H.2. tds_fdw — подключение к базам данных, использующим протокол TDS

Модуль `tds_fdw` предоставляет обёртку сторонних данных tds_fdw, которая может подключаться к базам данных, использующим протокол TDS (Tabular Data Stream, поток табличных данных), например *Sybase* и *Microsoft SQL Server*.

Для этой обёртки требуется библиотека, реализующая интерфейс DB-Library, например FreeTDS. Обёртка была протестирована с FreeTDS, но не с проприетарными реализациями DB-Library.

H.2.1. Ограничения

При включённом `match_column_names` поддерживаются условия `WHERE` и вынос столбцов наружу. Однако вынос наружу `JOIN` и пишущие операции не поддерживаются.

H.2.2. Установка tds_fdw

tds_fdw поставляется вместе с Postgres Pro Standard в виде отдельного пакета `tds_fdw` (подробные инструкции по установке приведены в Главе 16).

Установите расширение `tds_fdw` с помощью команды CREATE EXTENSION.

H.2.3. Настройка tds_fdw

H.2.3.1. Управление наборами символов/кодировкой

Хотя многие новые версии протокола TDS для связи с сервером используют только стандарт USC-2, FreeTDS преобразует UCS-2 в клиентскую кодировку по вашему выбору. Чтобы установить клиентскую кодировку, можно задать для переменной `client charset` значение `freetds.conf`. За подробностями обратитесь к описанию *The `freetds.conf` File* (файла freetds.conf) и *Localization and TDS 7.0* (локализации и TDS 7.0).

При получении указанной ниже ошибки во время работы Microsoft SQL Server с данными Unicode может потребоваться дополнительная настройка:

```
NOTICE: DB-Library notice: Msg #: 4004, Msg state: 1, Msg: Unicode data in a Unicode-only collation or ntext data cannot be sent to clients using DB-Library (such as ISQL) or ODBC version 3.7 or earlier., Server: PILLIUM\SQLEXPRESS, Process: , Line: 1, Level: 16

ERROR: DB-Library error: DB #: 4004, DB Msg: General SQL Server error: Check messages from the SQL Server, OS #: -1, OS Msg: (null), Level: 16
```

В таком случае придётся вручную указать в файле `freetds.conf` версию tds 7.0 или выше. За подробностями обратитесь к описанию *Файла `freetds.conf`* и *Выбора версии протокола TDS*.

H.2.3.2. Настройка зашифрованных подключений к MSSQL

Такие подключения настраиваются в файле `freetds.conf`. Найдите "шифрование" в разделе `freetds.conf settings` *файла `freetds.conf`*.

H.2.4. Использование

Чтобы обеспечить доступ к базе данных с помощью tds_fdw, выполните следующие шаги:

1. Создайте объект стороннего сервера с помощью CREATE SERVER, который будет представлять каждую базу данных, доступную для подключения.

2. Создайте сопоставление пользователей с помощью CREATE USER MAPPING для каждого пользователя базы данных, которому нужно разрешить доступ к каждому стороннему серверу.
3. Создайте стороннюю таблицу с помощью CREATE FOREIGN TABLE или IMPORT FOREIGN SCHEMA для каждой таблицы, к которой нужно получить доступ.

H.2.4.1. Создание стороннего сервера

Чтобы создать сторонний сервер, выполните команду CREATE SERVER со следующими параметрами:

`servername`

Имя сервера, адрес или имя узла стороннего сервера. Значением может быть DSN, как указано в `freetds.conf`. За подробностями обратитесь к разделу *Поиск по имени FreeTDS*. Можно задать в этом параметре список имён серверов, разделённый запятыми, тогда будут выполняться попытки подключиться к каждому серверу до тех пор, пока не будет установлено первое подключение. Такой список может использоваться для автоматического переключения на резервный сервер.

Обязательный: да

Значение по умолчанию: 127.0.0.1

`port`

Порт стороннего сервера. Вместо этого его можно указать в `freetds.conf` (если `servername` является DSN).

Обязательный: нет

`database`

База данных для подключения на этом сервере

Обязательный: нет

`dbuse`

Если `dbuse` равен 0, tds_fdw подключится напрямую к `database`. Если `dbuse` не равен 0, tds_fdw подключится к базе данных сервера по умолчанию, а затем выберет базу данных, вызвав функцию `dbuse()` библиотеки DB-Library. Чтобы использовать Azure, для `dbuse` необходимо установить значение 0.

Обязательный: нет

По умолчанию: 0

`language`

Язык, используемый для сообщений, и локаль, используемая для форматов дат. В большинстве систем для FreeTDS по умолчанию может использоваться английский (США). Вероятно, их также можно изменить в `freetds.conf`. За дополнительной информацией по этой настройке для MS SQL Server обратитесь к разделу *`SET LANGUAGE` in MS SQL Server* (SET LANGUAGE в MS SQL Server), а для Sybase ASE — к разделам *Sybase ASE login options* (Параметры входа в Sybase ASE) и *`SET LANGUAGE` in Sybase ASE* (SET LANGUAGE в Sybase ASE).

Обязательный: нет

`character_set`

Клиентская кодировка, которая будет использоваться при подключении. В протоколе TDS версии 7.0 и выше для подключения всегда используется UCS-2, поэтому в таких случаях этот параметр ничего не делает. За подробностями обратитесь к разделу *Localization and TDS 7.0* (локализация и TDS 7.0).

Обязательный: нет

`tds_version`

Версия протокола TDS, которая будет использоваться для этого сервера. За подробностями обратитесь к разделам *Choosing a TDS protocol version* (Выбор версии протокола TDS) и *History of TDS Versions* (История версий TDS).

Обязательный: нет

`msg_handler`

Функция, используемая для обработчика сообщений TDS. Допустимо одно из следующих значений:

- `notice`: сообщения TDS преобразуются в замечания PostgreSQL
- `blackhole`: сообщения TDS игнорируются

Обязательный: нет

По умолчанию: `blackhole`

`fdw_startup_cost`

Стоимость, которая используется при планировании запросов для представления издержек использования этой обёртки сторонних данных.

Обязательный: нет

`fdw_tuple_cost`

Стоимость, которая используется при планировании запросов для представления издержек выборки строк с этого сервера.

Обязательный: нет

`sqlserver_ansi_mode`

Стоимость, которая используется для представления издержек выборки строк с этого сервера, используемых при планировании запросов.

Этот параметр поддерживается только для SQL Server. Установка значения `true` включит следующие параметры стороннего сервера после успешного подключения к нему:

- `CONCAT_NULLS_YIELDS_NULL ON`
- `ANSI_NULLS ON`
- `ANSI_WARNINGS ON`
- `QUOTED_IDENTIFIER ON`
- `ANSI_PADDING ON`
- `ANSI_NULL_DFLT_ON ON`

Эти параметры в целом сопоставимы с параметром SQL Server `ANSI_DEFAULTS`. Параметр `sqlserver_ansi_mode`, напротив, в настоящее время не активирует следующие параметры:

- `CURSOR_CLOSE_ON_COMMIT`
- `IMPLICIT_TRANSACTIONS`

Это соответствует поведению собственных драйверов ODBC и OLEDB для SQL-серверов, которые явно отключают эти параметры, если не настроено иначе.

Обязательный: нет

По умолчанию: `false`

Некоторые параметры сторонней таблицы также могут быть установлены на уровне сервера. К ним относятся:

- `use_remote_estimate`
- `row_estimate_method`

Пример H.1. Создание стороннего сервера

```
CREATE SERVER mssql_svr
    FOREIGN DATA WRAPPER tds_fdw
    OPTIONS (servername '127.0.0.1', port '1433', database 'tds_fdw_test', tds_version
 '7.1');
```

H.2.4.2. Создание сопоставлений пользователей

Чтобы создать сопоставление пользователя, выполните команду CREATE USER MAPPING со следующими параметрами:

`username`

Имя пользователя учётной записи на стороннем сервере

> **Важно**
>
> Если вы используете Azure SQL, имя пользователя для стороннего сервера должно быть в формате `username@servername`. Если используется только имя пользователя, аутентификация завершится ошибкой.

Обязательный: да

`password`

Пароль учётной записи на стороннем сервере

Обязательный: да

Пример H.2. Создание сопоставления пользователя

```
CREATE USER MAPPING FOR postgres
    SERVER mssql_svr
    OPTIONS (username 'sa', password '');
```

H.2.4.3. Создание сторонней таблицы

Чтобы создать стороннюю таблицу, выполните команду CREATE FOREIGN TABLE со следующими параметрами:

`query`

Строка запроса, используемая для обращения к сторонней таблице

Обязательный: Да (взаимоисключающий с `table_name`) `schema_name`

Схема, в которой находится таблица. Имя схемы также можно включить в `table_name`.

Обязательный: нет

`table_name`

Таблица на стороннем сервере, к которой выполняется запрос

Псевдонимы: *таблица*

Обязательный: Да (взаимоисключающий с `query`)

`match_column_names`

Сопоставить локальные столбцы с удалёнными, сравнивая их имена в таблицах вместо использования порядка, в котором они появляются в наборе результатов. Требуется для `WHERE` и выноса столбцов наружу.

Обязательный: нет

`use_remote_estimate`

Оценить размер таблицы, выполнив какую-нибудь операцию на удалённом сервере, как определено в `row_estimate_method`, вместо использования локальной оценки, как определено в `local_tuple_estimate`

Обязательный: нет

`local_tuple_estimate`

Локально заданная оценка количества кортежей, которая используется, когда отключён параметр `use_remote_estimate`

Обязательный: нет

`row_estimate_method`

Может принимать одно из следующих значений:

- `execute`: выполнить запрос на удалённом сервере и получить фактическое количество строк в запросе
- `showplan_all`: получить оценку количества строк с помощью *MS SQL Server* *`SET SHOWPLAN_ALL`*

Обязательный: нет

По умолчанию: `execute`

Пример H.3. Создание сторонней таблицы

Использование определения `table_name`:

```
CREATE FOREIGN TABLE mssql_table (
    id integer,
    data varchar)
    SERVER mssql_svr
    OPTIONS (table_name 'dbo.mytable', row_estimate_method 'showplan_all');
```

Использование определений `schema_name` и `table_name`:

```
CREATE FOREIGN TABLE mssql_table (
    id integer,
```

```
    data varchar)
    SERVER mssql_svr
    OPTIONS (schema_name 'dbo', table_name 'mytable', row_estimate_method
 'showplan_all');
```

Использование определения `query`:

```
CREATE FOREIGN TABLE mssql_table (
    id integer,
    data varchar)
    SERVER mssql_svr
    OPTIONS (query 'SELECT * FROM dbo.mytable', row_estimate_method 'showplan_all');
```

Установка имени расположенного удалённо столбца:

```
CREATE FOREIGN TABLE mssql_table (
    id integer,
    col2 varchar OPTIONS (column_name 'data'))
    SERVER mssql_svr
    OPTIONS (schema_name 'dbo', table_name 'mytable', row_estimate_method
 'showplan_all');
```

H.2.4.4. Импорт сторонней схемы

Чтобы импортировать стороннюю схему, выполните команду IMPORT FOREIGN SCHEMA со следующими параметрами:

`import_default`

Управляет включением выражений столбцов `DEFAULT` в определения сторонних таблиц

Обязательный: нет

По умолчанию: false

`import_not_null`

Управляет включением ограничений столбцов `NOT NULL` в определения сторонних таблиц

Обязательный: нет

По умолчанию: true

Пример H.4. Импорт сторонней схемы

```
IMPORT FOREIGN SCHEMA dbo
    EXCEPT (mssql_table)
    FROM SERVER mssql_svr
    INTO public
    OPTIONS (import_default 'true');
```

H.2.4.5. Задание переменных

Чтобы задать переменную, выполните команду SET.

Доступны следующие переменные:

`tds_fdw.show_before_row_memory_stats`

Выводить статистику контекста памяти в журнал Postgres Pro перед выборкой каждой строки

`tds_fdw.show_after_row_memory_stats`

Выводить статистику контекста памяти в журнал Postgres Pro после выборки каждой строки `tds_fdw.show_finished_memory_stats`

Вывести статистику контекста памяти в журнал Postgres Pro после завершения запроса

Пример H.5. Задание переменной

```
postgres=# SET tds_fdw.show_finished_memory_stats=1;
SET
```

H.2.4.6. Просмотр запроса, выданного на удалённо расположенной системе

Чтобы просмотреть запрос, выданный в удалённо расположенной системе, выполните команду EXPLAIN `[ VERBOSE ]`.

H.2.5. Автор

Джефф Монти

H.3. wal2json — преобразование изменений из WAL в JSON с помощью логического декодирования

Модуль wal2json — это расширение Postgres Pro для логического декодирования, которое преобразует изменения базы данных из журнала предзаписи (WAL) в формат `JSON`. Postgres Pro имеет доступ к кортежам, сгенерированным командами `INSERT` и `UPDATE`. В зависимости от выбора варианта идентификации реплики также могут быть доступны предыдущие версии строк для команд `UPDATE` и `DELETE`. Изменения можно получить либо по протоколу потоковой репликации (слоты логической репликации), либо через специальный SQL API.

Формат версии 1 создаёт JSON-объект для каждой транзакции. Этот объект содержит все новые и старые кортежи. Также есть возможность включить в него такие свойства, как метки времени транзакций, имена, дополненные именами схем, типы данных и идентификаторы транзакций.

Формат версии 2 создаёт JSON-объект для каждого кортежа с необязательными JSON-объектами, обозначающими начало и конец транзакции. Также можно включить разные свойства транзакции.

H.3.1. Установка и настройка

wal2json поставляется вместе с Postgres Pro Standard в виде отдельного пакета `wal2json-std-17` (подробные инструкции по установке приведены в Главе 16). После установки Postgres Pro Standard выполните следующие действия, чтобы подготовить wal2json к работе:

1. Включите логическое декодирование, задав для параметра `wal_level` значение `logical` в файле `postgresql.conf`.

   Дополнительно вы можете задать значения для параметров `max_replication_slots` и `max_wal_senders`.
2. Перезапустите сервер баз данных, чтобы изменения вступили в силу.

H.3.2. Параметры

Модуль wal2json предоставляет различные параметры для управления логическим декодированием:

`include-xids`

Добавить идентификатор транзакции к каждому набору изменений в JSON-выводе. Значение по умолчанию — `off`.

`include-timestamp`

Добавить `timestamp` к каждому набору изменений в JSON-выводе. Значение по умолчанию — `off`.

`include-schemas`

Добавить имя схемы к каждой записи об изменениях в JSON-выводе. Значение по умолчанию — `on`.

`include-types`

Добавить `type` к каждой записи об изменениях в JSON-выводе. Значение по умолчанию — `on`.

`include-typmod`

Добавить модификатор типа для столбцов, у которых они есть. Значение по умолчанию — `on`.

`include-type-oids`

Добавить OID типа. Значение по умолчанию — *off*.

`include-domain-data-type`

Заменить доменное имя нижележащим типом данных. Значение по умолчанию — `off`.

`include-column-positions`

Добавить позицию столбца (`pg_attribute.attnum`). Значение по умолчанию — `off`.

`include-origin`

Добавить источник каждого изменения. Значение по умолчанию — `off`.

`include-not-null`

Добавить информацию, помечен ли столбец как `not null` в поле `columnoptionals` JSON-вывода. Значение по умолчанию — `off`.

`include-default`

Добавить выражения по умолчанию в JSON-вывод. Значение по умолчанию — `off`.

`include-pk`

Добавить информацию о первичном ключе, включая имена столбцов и типы данных, в поле `pk` JSON-вывода. Значение по умолчанию — `off`.

`numeric-data-types-as-string`

Преобразовать значения числовых типов данных в строки в JSON-выводе. Спецификация JSON не поддерживает `Infinity` и `NaN` как допустимые числовые значения. Для чисел двойной точности могут возникнуть *потенциальные проблемы совместимости* . Значение по умолчанию — `off`.

`pretty-print`

Добавить пробелы и отступы в структуру JSON-вывода. Значение по умолчанию — `off`.

`write-in-chunks`

Включить запись JSON-вывода небольшими фрагментами вместо целого набора изменений. Этот параметр используется только когда для `format-version` задано значение `1`. Значение по умолчанию — `off`.

`include-lsn`

Добавить поле `nextlsn` к каждому набору изменений. Значение по умолчанию — `off`.

`include-transaction`

Добавить записи о начале и конце каждой транзакции в JSON-вывод. Значение по умолчанию — `off`.

`filter-origins`

Исключить изменения из указанных источников. По умолчанию этот параметр пуст, что означает, что источники не будут отфильтрованы. Задаётся в виде разделённого запятыми списка значений.

`filter-tables`

Исключить строки из указанных таблиц. По умолчанию этот параметр пуст, что означает, что таблицы не будут отфильтрованы. Задаётся в виде разделённого запятыми списка значений. Имена таблиц должны быть дополнены схемой. Имя вида `*.foo` означает таблицу `foo` во всех схемах, а имя вида `bar.*` означает все таблицы в схеме `bar`. Специальные символы (пробел, одинарная кавычка, запятая, точка, звёздочка) должны быть экранированы обратной косой чертой. Схема и таблица чувствительны к регистру. Например, таблица `"public"."Foo bar"` должна быть указана как `public.Foo\ bar`.

`add-tables`

Включить строки только из указанных таблиц. По умолчанию этот параметр пуст, что означает, что включены все таблицы из всех схем. Для этого параметра действуют те же правила, что и для `filter-tables`.

`filter-msg-prefixes`

Исключить сообщения, чей префикс указан в значении параметра. По умолчанию этот параметр пуст, что означает, что сообщения не будут отфильтрованы. Задаётся в виде разделённого запятыми списка значений.

`add-msg-prefixes`

Включить только те сообщения, чей префикс указан в значении параметра. По умолчанию используются все префиксы. Задаётся в виде разделённого запятыми списка значений. `wal2json` применяет `filter-msg-prefixes` до этого параметра.

`format-version`

Определить, какой формат вывода использовать. Значение по умолчанию — `1`.

`actions`

Определить, какие операции будут включены в JSON-вывод. По умолчанию используются все действия (`INSERT`, `UPDATE`, `DELETE` и `TRUNCATE`). Однако, если используется `format-version = 1`, операция `TRUNCATE` не включается (в целях обратной совместимости).

H.3.3. Примеры

Существует два способа получить изменения (JSON-объекты) в wal2json: через функции, вызываемые в SQL или с помощью pg_recvlogical.

H.3.3.1. pg_recvlogical

Ниже представлен пример, как получить JSON-объекты в wal2json с помощью pg_recvlogical. Помимо вышеуказанной настройки необходимо настроить подключение для репликации, чтобы использовать pg_recvlogical. Начиная с Postgres Pro версии 10, логическая репликация сопоставляет обычные записи с именем базы данных или ключевыми словами, такими как `all`.

Чтобы настроить подключение для репликации и параметры базы данных:

1. Добавьте правило подключения для репликации в файле `pg_hba.conf`:

```
local     mydatabase      myuser                          trust
```

2. Дополнительно можно задать `max_wal_senders` в файле `postgresql.conf`:

```
max_wal_senders = 1
```

3. Перезапустите сервер баз данных, если изменили `max_wal_senders`.

Чтобы получить JSON-объекты в wal2json:

1. Откройте терминал и подключитесь к базе данных:

```
 $ pg_recvlogical -d postgres --slot test_slot --create-slot -P wal2json
 $ pg_recvlogical -d postgres --slot test_slot --start -o pretty-print=1 -o
add-msg-prefixes=wal2json -f -
```

2. В другом терминале:

```
$ cat /tmp/example1.sql
CREATE TABLE table1_with_pk (a SERIAL, b VARCHAR(30), c TIMESTAMP NOT NULL,
PRIMARY KEY(a, c));
CREATE TABLE table1_without_pk (a SERIAL, b NUMERIC(5,2), c TEXT);

BEGIN;
INSERT INTO table1_with_pk (b, c) VALUES('Backup and Restore', now());
INSERT INTO table1_with_pk (b, c) VALUES('Tuning', now());
INSERT INTO table1_with_pk (b, c) VALUES('Replication', now());
SELECT pg_logical_emit_message(true, 'wal2json', 'this message will be
delivered');
SELECT pg_logical_emit_message(true, 'pgoutput', 'this message will be
filtered');
DELETE FROM table1_with_pk WHERE a < 3;
SELECT pg_logical_emit_message(false, 'wal2json', 'this non-transactional
message will be delivered even if you rollback the transaction');

INSERT INTO table1_without_pk (b, c) VALUES(2.34, 'Tapir');
-- эта запись не добавляется в поток, так как отсутствует первичный ключ или
репликационный идентификатор
UPDATE table1_without_pk SET c = 'Anta' WHERE c = 'Tapir';
COMMIT;

DROP TABLE table1_with_pk;
DROP TABLE table1_without_pk;

$ psql -At -f /tmp/example1.sql postgres
CREATE TABLE
CREATE TABLE
BEGIN
INSERT 0 1
INSERT 0 1
INSERT 0 1
3/78BFC828
3/78BFC880
DELETE 2
3/78BFC990
INSERT 0 1
UPDATE 1
COMMIT
DROP TABLE
DROP TABLE
```

3. Вывод в первом терминале может выглядеть так:

```
$ psql -At -f /tmp/example2.sql postgres
CREATE TABLE
```

```
         CREATE TABLE
         init
         BEGIN
         INSERT 0 1
         INSERT 0 1
         INSERT 0 1
         3/78C2CA50
         3/78C2CAA8
         DELETE 2
         3/78C2CBD8
         INSERT 0 1
         UPDATE 1
         COMMIT
         {
             "change": [
                 {
                     "kind": "message",
                     "transactional": false,
                     "prefix": "wal2json",
                     "content": "this non-transactional message will be delivered
even if you rollback the transaction"
                 }
             ]
         }
         psql:/tmp/example2.sql:17: WARNING:  table "table2_without_pk" without
primary key or replica identity is nothing
         {
           "change": [
             {
               "kind": "insert",
               "schema": "public",
               "table": "table2_with_pk",
               "columnnames": ["a", "b", "c"],
               "columntypes": ["integer", "character varying(30)", "timestamp
without time zone"],
               "columnvalues": [1, "Backup and Restore", "2018-03-27
12:05:29.914496"]
             }
             ,{
               "kind": "insert",
               "schema": "public",
               "table": "table2_with_pk",
               "columnnames": ["a", "b", "c"],
               "columntypes": ["integer", "character varying(30)", "timestamp
without time zone"],
               "columnvalues": [2, "Tuning", "2018-03-27 12:05:29.914496"]
             }
             ,{
               "kind": "insert",
               "schema": "public",
               "table": "table2_with_pk",
               "columnnames": ["a", "b", "c"],
               "columntypes": ["integer", "character varying(30)", "timestamp
without time zone"],
               "columnvalues": [3, "Replication", "2018-03-27 12:05:29.914496"]
             }
                 ,{
                     "kind": "message",
```

```
                    "transactional": true,
                    "prefix": "wal2json",
                    "content": "this message will be delivered"
                }
            ,{
              "kind": "delete",
              "schema": "public",
              "table": "table2_with_pk",
              "oldkeys": {
                "keynames": ["a", "c"],
                "keytypes": ["integer", "timestamp without time zone"],
                "keyvalues": [1, "2018-03-27 12:05:29.914496"]
              }
            }
            ,{
              "kind": "delete",
              "schema": "public",
              "table": "table2_with_pk",
              "oldkeys": {
                "keynames": ["a", "c"],
                "keytypes": ["integer", "timestamp without time zone"],
                "keyvalues": [2, "2018-03-27 12:05:29.914496"]
              }
            }
            ,{
              "kind": "insert",
              "schema": "public",
              "table": "table2_without_pk",
              "columnnames": ["a", "b", "c"],
              "columntypes": ["integer", "numeric(5,2)", "text"],
              "columnvalues": [1, 2.34, "Tapir"]
            }
          ]
        }
        stop
        DROP TABLE
        DROP TABLE
```

4. Чтобы удалить слот в первом терминале:

```
Ctrl+C
$ pg_recvlogical -d postgres --slot test_slot --drop-slot
```

H.3.3.2. Вызов SQL-функций

Ниже представлены примеры, как получить изменения в wal2json с помощью SQL.

Если для `format-version` задано значение `1`, скрипт может выглядеть так:

```
        $ cat /tmp/example2.sql
        CREATE TABLE table2_with_pk (a SERIAL, b VARCHAR(30), c TIMESTAMP NOT NULL,
 PRIMARY KEY(a, c));
        CREATE TABLE table2_without_pk (a SERIAL, b NUMERIC(5,2), c TEXT);

        SELECT 'init' FROM pg_create_logical_replication_slot('test_slot', 'wal2json');

        BEGIN;
        INSERT INTO table2_with_pk (b, c) VALUES('Backup and Restore', now());
```

```
        INSERT INTO table2_with_pk (b, c) VALUES('Tuning', now());
        INSERT INTO table2_with_pk (b, c) VALUES('Replication', now());
        SELECT pg_logical_emit_message(true, 'wal2json', 'this message will be
 delivered');
        SELECT pg_logical_emit_message(true, 'pgoutput', 'this message will be
 filtered');
        DELETE FROM table2_with_pk WHERE a < 3;
        SELECT pg_logical_emit_message(false, 'wal2json', 'this non-transactional
 message will be delivered even if you rollback the transaction');

        INSERT INTO table2_without_pk (b, c) VALUES(2.34, 'Tapir');
        -- эта запись не добавляется в поток, так как отсутствует первичный ключ или
 репликационный идентификатор
        UPDATE table2_without_pk SET c = 'Anta' WHERE c = 'Tapir';
        COMMIT;

        SELECT data FROM pg_logical_slot_get_changes('test_slot', NULL, NULL, 'pretty-
print', '1', 'add-msg-prefixes', 'wal2json');
        SELECT 'stop' FROM pg_drop_replication_slot('test_slot');

        DROP TABLE table2_with_pk;
        DROP TABLE table2_without_pk;
```

Ожидаемый вывод может выглядеть так:

```
        $ psql -At -f /tmp/example2.sql postgres
        CREATE TABLE
        CREATE TABLE
        init
        BEGIN
        INSERT 0 1
        INSERT 0 1
        INSERT 0 1
        3/78C2CA50
        3/78C2CAA8
        DELETE 2
        3/78C2CBD8
        INSERT 0 1
        UPDATE 1
        COMMIT
        {
            "change": [
                {
                    "kind": "message",
                    "transactional": false,
                    "prefix": "wal2json",
                    "content": "this non-transactional message will be delivered even
 if you rollback the transaction"
                }
            ]
        }
        psql:/tmp/example2.sql:17: WARNING:  table "table2_without_pk" without primary
 key or replica identity is nothing
        {
          "change": [
            {
              "kind": "insert",
```

```
             "schema": "public",
             "table": "table2_with_pk",
             "columnnames": ["a", "b", "c"],
             "columntypes": ["integer", "character varying(30)", "timestamp without
time zone"],
             "columnvalues": [1, "Backup and Restore", "2018-03-27 12:05:29.914496"]
           }
           ,{
             "kind": "insert",
             "schema": "public",
             "table": "table2_with_pk",
             "columnnames": ["a", "b", "c"],
             "columntypes": ["integer", "character varying(30)", "timestamp without
time zone"],
             "columnvalues": [2, "Tuning", "2018-03-27 12:05:29.914496"]
           }
           ,{
             "kind": "insert",
             "schema": "public",
             "table": "table2_with_pk",
             "columnnames": ["a", "b", "c"],
             "columntypes": ["integer", "character varying(30)", "timestamp without
time zone"],
             "columnvalues": [3, "Replication", "2018-03-27 12:05:29.914496"]
           }
               ,{
                   "kind": "message",
                   "transactional": true,
                   "prefix": "wal2json",
                   "content": "this message will be delivered"
               }
           ,{
             "kind": "delete",
             "schema": "public",
             "table": "table2_with_pk",
             "oldkeys": {
               "keynames": ["a", "c"],
               "keytypes": ["integer", "timestamp without time zone"],
               "keyvalues": [1, "2018-03-27 12:05:29.914496"]
             }
           }
           ,{
             "kind": "delete",
             "schema": "public",
             "table": "table2_with_pk",
             "oldkeys": {
               "keynames": ["a", "c"],
               "keytypes": ["integer", "timestamp without time zone"],
               "keyvalues": [2, "2018-03-27 12:05:29.914496"]
             }
           }
           ,{
             "kind": "insert",
             "schema": "public",
             "table": "table2_without_pk",
             "columnnames": ["a", "b", "c"],
             "columntypes": ["integer", "numeric(5,2)", "text"],
             "columnvalues": [1, 2.34, "Tapir"]
```

```
            }
          ]
        }
        stop
        DROP TABLE
        DROP TABLE
```

Если для `format-version` задано значение `2`, скрипт может выглядеть так:

```
        $ cat /tmp/example3.sql
        CREATE TABLE table3_with_pk (a SERIAL, b VARCHAR(30), c TIMESTAMP NOT NULL,
 PRIMARY KEY(a, c));
        CREATE TABLE table3_without_pk (a SERIAL, b NUMERIC(5,2), c TEXT);

        SELECT 'init' FROM pg_create_logical_replication_slot('test_slot', 'wal2json');

        BEGIN;
        INSERT INTO table3_with_pk (b, c) VALUES('Backup and Restore', now());
        INSERT INTO table3_with_pk (b, c) VALUES('Tuning', now());
        INSERT INTO table3_with_pk (b, c) VALUES('Replication', now());
        SELECT pg_logical_emit_message(true, 'wal2json', 'this message will be
 delivered');
        SELECT pg_logical_emit_message(true, 'pgoutput', 'this message will be
 filtered');
        DELETE FROM table3_with_pk WHERE a < 3;
        SELECT pg_logical_emit_message(false, 'wal2json', 'this non-transactional
 message will be delivered even if you rollback the transaction');

        INSERT INTO table3_without_pk (b, c) VALUES(2.34, 'Tapir');
        -- эта запись не добавляется в поток, так как отсутствует первичный ключ или
 репликационный идентификатор
        UPDATE table3_without_pk SET c = 'Anta' WHERE c = 'Tapir';
        COMMIT;

        SELECT data FROM pg_logical_slot_get_changes('test_slot', NULL, NULL, 'format-
version', '2', 'add-msg-prefixes', 'wal2json');
        SELECT 'stop' FROM pg_drop_replication_slot('test_slot');

        DROP TABLE table3_with_pk;
        DROP TABLE table3_without_pk;
```

Ожидаемый вывод может выглядеть так:

```
        $ psql -At -f /tmp/example3.sql postgres
        CREATE TABLE
        CREATE TABLE
        init
        BEGIN
        INSERT 0 1
        INSERT 0 1
        INSERT 0 1
        3/78CB8F30
        3/78CB8F88
        DELETE 2
        3/78CB90B8
        INSERT 0 1
        UPDATE 1
```

```
        COMMIT
        psql:/tmp/example3.sql:20: WARNING:  no tuple identifier for UPDATE in table
 "public"."table3_without_pk"
        {"action":"M","transactional":false,"prefix":"wal2json","content":"this non-
transactional message will be delivered even if you rollback the transaction"}
        {"action":"B"}
        {"action":"I","schema":"public","table":"table3_with_pk","columns":
[{"name":"a","type":"integer","value":1},{"name":"b","type":"character
 varying(30)","value":"Backup and Restore"},{"name":"c","type":"timestamp without time
 zone","value":"2019-12-29 04:58:34.806671"}]}
        {"action":"I","schema":"public","table":"table3_with_pk","columns":
[{"name":"a","type":"integer","value":2},{"name":"b","type":"character
 varying(30)","value":"Tuning"},{"name":"c","type":"timestamp without time
 zone","value":"2019-12-29 04:58:34.806671"}]}
        {"action":"I","schema":"public","table":"table3_with_pk","columns":
[{"name":"a","type":"integer","value":3},{"name":"b","type":"character
 varying(30)","value":"Replication"},{"name":"c","type":"timestamp without time
 zone","value":"2019-12-29 04:58:34.806671"}]}
        {"action":"M","transactional":true,"prefix":"wal2json","content":"this message
 will be delivered"}
        {"action":"D","schema":"public","table":"table3_with_pk","identity":
[{"name":"a","type":"integer","value":1},{"name":"c","type":"timestamp without time
 zone","value":"2019-12-29 04:58:34.806671"}]}
        {"action":"D","schema":"public","table":"table3_with_pk","identity":
[{"name":"a","type":"integer","value":2},{"name":"c","type":"timestamp without time
 zone","value":"2019-12-29 04:58:34.806671"}]}
        {"action":"I","schema":"public","table":"table3_without_pk","columns":
[{"name":"a","type":"integer","value":1},
{"name":"b","type":"numeric(5,2)","value":2.34},
{"name":"c","type":"text","value":"Tapir"}]}
        {"action":"C"}
        stop
        DROP TABLE
        DROP TABLE
```

Приложение I. Дополнительно поставляемые программы

В этом приложении, Приложении F и Приложении G содержится информация о дополнительных модулях, поставляемых в составе дистрибутива Postgres Pro Standard. За дополнительной информацией о серверных расширениях и подключаемых компонентах обратитесь к Приложению F и Приложению G.

В этом приложении описываются вспомогательные программы. После установки их можно найти в подкаталоге `bin` инсталляции Postgres Pro Standard и использовать как любую другую программу.

I.1. Дополнительные клиентские приложения PostgreSQL/Postgres Pro

В этом разделе рассматриваются дополнительные клиентские приложения PostgreSQL/Postgres Pro Standard, включённые в дистрибутив Postgres Pro Standard. Их можно запускать откуда угодно, независимо от того, где находится сервер баз данных. Обратитесь также к Справке: «Клиентские приложения Postgres Pro» за информацией о клиентских приложениях, относящихся к ядру дистрибутива Postgres Pro Standard.

oid2name

oid2name — преобразовать в имена OID и номера файловых узлов в каталоге данных Postgres Pro

Синтаксис

`oid2name` [*`параметр`*...]

Описание

oid2name — вспомогательная программа, помогающая администраторам исследовать структуру файлов Postgres Pro. Чтобы использовать её, необходимо понимать структуру базы данных на уровне файлов, описанную в Главе 63.

> **Примечание**
>
> Имя «oid2name» сложилось исторически и на самом деле вводит в заблуждение, так как чаще всего, используя её, вы будете иметь дело с номерами файловых узлов таблиц (эти номера образуют имена файлов в каталоге баз данных). Необходимо понимать, что OID таблиц отличаются от номеров файловых узлов!

Программа oid2name подключается к целевой базе данных и извлекает информацию об OID, файловых узлах и/или именах таблиц. С её помощью можно также просмотреть OID базы данных или табличного пространства.

Параметры

oid2name принимает следующие аргументы командной строки:

`-f` *`файловый_узел`*
`--filenode=`*`файловый_узел`*

показать информацию о таблице, к которой относится *`файловый_узел`*.

`-i`
`--indexes`

включить в вывод индексы и последовательности.

`-o` *`oid`*
`--oid=`*`oid`*

показать информацию о таблице с OID, равным *`oid`*.

`-q`
`--quiet`

не выводить заголовки (полезно для скриптов).

`-s`
`--tablespaces`

показать OID табличных пространств.

`-S`
`--system-objects`

включить в вывод системные объекты (те, что находятся в схемах `information_schema`, `pg_toast` и `pg_catalog`).

`-t` *`шаблон_имён_таблиц`*
`--table=`*`шаблон_имён_таблиц`*

показать информацию о таблицах, подпадающих под *`шаблон_имён_таблиц`*.

`-V`
`--version`

вывести версию oid2name и завершиться.

`-x`
`--extended`

вывести дополнительные сведения о каждом показываемом объекте: имя табличного пространства, имя схемы и OID.

`-?`
`--help`

вывести справку об аргументах командной строки oid2name и завершиться.

oid2name также принимает в командной строке следующие аргументы, задающие параметры подключения:

`-d` *`база_данных`*
`--dbname=`*`база_данных`*

целевая база данных.

`-h` *`сервер`*
`--host=`*`сервер`*

адрес сервера баз данных.

`-H` *`сервер`*

адрес сервера баз данных. Этот параметр считается *устаревшим* с 12 версии Postgres Pro.

`-p` *`порт`*
`--port=`*`порт`*

порт сервера баз данных.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

имя пользователя для подключения.

Чтобы получить информацию об определённых таблицах, выберите их с аргументом `-o`, `-f` и/или `-t`. Параметр `-o` принимает OID, `-f` — файловый узел, а `-t` — имя таблицы (на самом деле он принимает шаблон `LIKE`, так что в нём можно задать что-то вроде `foo%`). Вы можете использовать эти аргументы в любом количестве; в вывод будут включены все объекты, соответствующие любым из этих указаний. Но заметьте, что эти аргументы будут выбирать только объекты в базе данных, заданной ключом `-d`.

Если аргументы `-o`, `-f` и `-t` отсутствуют, но передаётся `-d`, будут выведены все таблицы в базе данных с именем, заданным в `-d`. В этом режиме набором выводимых данных управляют параметры `-S` и `-i`.

Если отсутствует и аргумент `-d`, выводится список OID баз данных. Также можно получить список табличных пространств, передав аргумент `-s`.

Переменные окружения

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию.

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Программе oid2name требуется работающий сервер баз данных с исправными системными каталогами. Таким образом, в ситуациях с катастрофическими повреждениями базы данных её использование ограничено.

Примеры

```
$ # Что вообще есть на этом сервере базы данных?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # Хорошо, давайте взглянем на базу alvherre
$ cd $PGDATA/base/17228

$ # Получим первые 10 объектов базы (отсортированных по размеру) в табличном
 пространстве по умолчанию
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751

$ # Поинтересуемся, что за файл 155173
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts

$ # Можно узнать сразу о нескольких объектах
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
-------------------------
```

```
    155173       accounts
   1155291  accounts_pkey

$ # Можно добавить другие параметры и получить дополнительные подробности с -x
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
------------------------------------------------------
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default

$ # Вычислить объём, который занимает на диске каждый объект БД
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16            1155287  branches_pkey
16            1155289  tellers_pkey
17561             1155291  accounts_pkey
...

$ # То же самое, но с сортировкой по размеру
$ du [0-9]* | sort -rn | while read SIZE FN
> do
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
133466             155173    accounts
17561            1155291  accounts_pkey
1177              16717  pg_proc_proname_args_nsp_index
...

$ # Просмотреть содержимое табличных пространств можно в каталоге pg_tblspc
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # Объекты каких баз данных находятся в табличном пространстве "fastdisk"?
$ ls -d 155151/*
155151/17228/  155151/PG_VERSION

$ # И что это за база данных 17228?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default
```

```
$ # Давайте посмотрим, какие объекты этой базы содержатся в данном табличном
 пространстве.
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156

$ # Мда, таблица невелика... и что это за таблица?
$ oid2name -d alvherre -f 155156
From database "alvherre":
  Filenode  Table Name
----------------------
    155156         foo
```

Автор

Б. Палмер `<bpalmer@crimelabs.net>`

pg_probackup

pg_probackup — управление резервным копированием и восстановлением кластеров баз данных Postgres Pro

Синтаксис

`pg_probackup version`

`pg_probackup help` [*команда*]

`pg_probackup init -B` *каталог_копий* `--skip-if-exists`

`pg_probackup add-instance -B` *каталог_копий* `-D` *каталог_данных* `--instance` *имя_экземпляра* `--skip-if-exists`

`pg_probackup del-instance -B` *каталог_копий* `--instance` *имя_экземпляра*

`pg_probackup set-config -B` *каталог_копий* `--instance` *имя_экземпляра* [*параметр*...]

`pg_probackup set-backup -B` *каталог_копий* `--instance` *имя_экземпляра* `-i` *ид_резервной_копии* [*параметр*...]

`pg_probackup show-config -B` *каталог_копий* `--instance` *имя_экземпляра* [*параметр*...]

`pg_probackup show -B` *каталог_копий* [*параметр*...]

`pg_probackup backup -B` *каталог_копий* `--instance` *имя_экземпляра* `-b` *режим_копирования* [*параметр*...]

`pg_probackup restore -B` *каталог_копий* `--instance` *имя_экземпляра* [*параметр*...]

`pg_probackup checkdb -B` *каталог_копий* `--instance` *имя_экземпляра* `-D` *каталог_данных* [параметр...]

`pg_probackup validate -B` *каталог_копий* [*параметр*...]

`pg_probackup merge -B` *каталог_копий* `--instance` *имя_экземпляра* `-i` *ид_резервной_копии* [*параметр*...]

`pg_probackup delete -B` *каталог_копий* `--instance` *имя_экземпляра* { `-i` *ид_резервной_копии* | `--delete-wal` | `--delete-expired` | `--merge-expired` } [*параметр*...]

`pg_probackup archive-push -B` *каталог_копий* `--instance` *имя_экземпляра* `--wal-file-path` *путь_файлов_wal* `--wal-file-name` *имя_файла_wal* [*параметр*...]

`pg_probackup archive-get -B` *каталог_копий* `--instance` *имя_экземпляра* `--wal-file-path` *путь_файлов_wal* `--wal-file-name` *имя_файла_wal* [*параметр*...]

`pg_probackup catchup -b` *режим_синхронизации* `--source-pgdata=`*путь_к_копируемому_каталогу_данных* `--destination-pgdata=`*путь_к_целевому_каталогу_данных* [*параметр*...]

`pg_probackup maintain -B` *каталог_копий* `--instance` *имя_экземпляра* [`--backup-id` *ид_резервной_копии*] `--status-sync` [*параметр*...]

Описание

pg_probackup — это утилита для управления резервным копированием и восстановлением кластеров баз данных Postgres Pro. Она предназначена для регулярного создания резервных копий экземпляра Postgres Pro, позволяющих восстанавливать сервер в случае необходимости. pg_probackup поддерживает PostgreSQL версии 11 и новее.

- Обзор
- Быстрый запуск
- Установка и подготовка
- Справка по командной строке
- Использование

Обзор

По сравнению с другими средствами резервного копирования pg_probackup имеет следующие преимущества, полезные для реализации различных стратегий резервного копирования и работы с базами данных большого объёма:

- Инкрементальное копирование: выбирая один из трёх режимов инкрементального копирования, вы можете реализовать стратегию резервного копирования, соответствующую вашему профилю транзакционной нагрузки. Это позволяет сэкономить место на диске и создавать копии быстрее, чем при полном копировании. Восстановление инкрементальных копий также осуществляется быстрее, чем воспроизведение файлов WAL.
- Инкрементальное восстановление: ускорение восстановления из копии благодаря повторному использованию неизменённых страниц, имеющихся в `PGDATA`.
- Проверка: автоматический контроль целостности данных и проверка резервных копий без восстановления данных кластера.
- Контроль целостности: выполняемая по запросу проверка экземпляра Postgres Pro с помощью команды `checkdb`.
- Политика хранения: управление архивами WAL и резервными копиями в соответствии с установленными правилами их хранения. Вы можете ограничить хранение резервных копий по времени или их количеству, а также переопределить время жизни (TTL) для избранных копий. Потерявшие актуальность резервные копии могут объединяться или удаляться.
- Параллельное выполнение: выполнение внутренних процессов команд `backup`, `restore`, `merge`, `delete`, `validate` и `checkdb` в несколько параллельных потоков.
- Сжатие: хранение копируемых данных в сжатом состоянии для экономии дискового пространства.
- Исключение дублирования: экономия дискового пространства за счёт фильтрации при инкрементальном копировании файлов, не содержащих непосредственно данные (например, файлов `_vm` или `_fsm`), если эти файлы не изменялись с момента создания предыдущей копии в цепочке инкрементальных копий.
- Удалённый режим работы: выполнение резервного копирования экземпляра Postgres Pro, находящегося в удалённой системе, и удалённое восстановление.
- Получение резервной копии с ведомого: исключение дополнительной нагрузки на ведущий сервер.
- Архивирование внешних каталогов: резервное копирование файлов и каталогов, расположенных вне каталога данных Postgres Pro (`PGDATA`), например скриптов, файлов конфигурации, журналов или SQL-дампов.
- Каталогизация резервных копий: получение списка резервных копий и соответствующей метаинформации в виде простого текста или JSON.
- Каталогизация архивов WAL: получение списка всех линий времени в WAL и соответствующей метаинформации в виде простого текста или JSON.
- Частичное восстановление: восстановление только избранных баз данных.
- Синхронизация: клонирование экземпляра Postgres Pro, в результате которого отставший ведомый сервер синхронизируется с ведущим.

Для управления резервными копиями pg_probackup создаёт *каталог резервных копий*. В этом каталоге сохраняются все файлы резервных копий с дополнительной метаинформацией, а также архивы WAL, необходимые для восстановления на момент времени. Вы можете хранить резервные копии разных экземпляров в отдельных подкаталогах одного каталога копий.

Используя pg_probackup, вы можете выполнять полное или инкрементальное резервное копирование:

- Полные резервные копии содержат все файлы данных, необходимые для восстановления кластера баз данных с нуля.
- Инкрементальные копии создаются на уровне страниц и включают только те данные, которые изменились со времени последнего копирования. Это позволяет сэкономить место на диске и создавать копии быстрее, чем при полном копировании. Восстановление инкрементальных копий также осуществляется быстрее, чем воспроизведение файлов WAL. pg_probackup поддерживает следующие режимы инкрементального копирования:

  - Разностное копирование. В режиме `DELTA` pg_probackup считывает все файлы данных в каталоге данных и копирует только те страницы, которые изменились со времени предыдущего копирования. Для использования данного режима не требуется производить непрерывное архивирование. В этом режиме объём ввода-вывода может равняться объёму при полном резервном копировании.
  - Страничное копирование. В режиме `PAGE` pg_probackup сканирует все файлы WAL в архиве с момента создания предыдущей полной или инкрементальной копии. Новая резервная копия будет содержать только страницы, фигурирующие в записях WAL. При этом необходимо, чтобы в архиве WAL сохранялись все файлы WAL, записанные после предыдущей копии. Если размер этих файлов сравним с общим размером файлов базы данных, ускорение будет менее значительным, но размер копии будет всё же меньше.
  - Копирование изменений. В режиме `PTRACK` Postgres Pro отслеживает изменения страниц на лету. Чтобы он работал, не требуется производить непрерывное архивирование WAL. При каждом изменении страницы отношения она помечается в специальной карте `PTRACK`. Это отслеживание привносит небольшие издержки в работу сервера, но значительно ускоряет инкрементальное резервное копирование.

Предупреждение

После повышения ведомого сервера до ведущего и переключения линий времени создайте первую резервную копию в режиме **FULL или DELTA** — другие режимы могут повредить данные.

pg_probackup может производить копирование только работающего экземпляра, а для обеспечения целостности таких копий требуется копировать WAL. Поэтому вне зависимости от выбранного режима копирования (FULL, PAGE или DELTA), чтобы pg_probackup мог получить полноценную копию, нужно выбрать один из следующих *режимов доставки WAL*:

- ARCHIVE. В таком режиме целостность копий обеспечивается посредством непрерывного архивирования. Это режим доставки WAL по умолчанию.
- STREAM. Такие резервные копии включают все файлы, необходимые для восстановления целостного состояния кластера на момент создания копии. Вне зависимости от того, осуществляется ли непрерывное архивирование, необходимые для восстановления сегменты WAL считываются по протоколу потоковой репликации во время резервного копирования и включаются в состав резервной копии. Поэтому такие резервные копии называются *автономными* или *самодостаточными*.

Ограничения

В настоящее время pg_probackup имеет следующие ограничения:

- Режим удалённого сервера на платформе Windows не поддерживается.
- В Unix-системах копию базы Postgres Pro можно сделать только от имени того пользователя, который запускает сервер Postgres Pro. Например, если сервер Postgres Pro запускает пользователь `postgres`, команду `backup` также должен выполнять пользователь `postgres`. Для удовлетворения этого требования в случае использования удалённого режима и SSH в параметре `--remote-user` необходимо передать `postgres`.

- На сервере Postgres Pro, где была сделана копия, и на сервере, где она будет восстанавливаться, должны быть одинаковые значения параметров block_size и wal_block_size и одинаковая основная версия. В зависимости от конфигурации кластера, Postgres Pro может накладывать дополнительные ограничения, например, по архитектуре процессора и версии libc/icu.

Быстрый запуск

Чтобы быстро начать работу с pg_probackup, выполните следующие действия. Это позволит настроить резервное копирование удалённо в режимах FULL и DELTA и увидеть некоторые базовые операции pg_probackup. Далее используются следующие обозначения:

- `backup` — роль в Postgres Pro, используемая для подключения к кластеру Postgres Pro.
- `backupdb` — база данных, через которую выполняется подключение к кластеру Postgres Pro.
- `backup_host` — система, где находится каталог резервных копий.
- `backup_user` — пользователь в системе `backup_host`, от имени которого выполняются все операции pg_probackup.
- `/mnt/backups` — путь к каталогу резервных копий в системе `backup_host`.
- `postgres_host` — система, в которой работает Postgres Pro.
- `postgres` — пользователь в системе `postgres_host`, от имени которого запускаются процессы кластера Postgres Pro.
- `/var/lib/pgpro/std-16/data` — путь к каталогу данных Postgres Pro в системе `postgres_host`.

Выполните следующие шаги:

1. Установите pg_probackup в обеих системах: `backup_host` (сервер резервного копирования) и `postgres_host` (сервер баз данных).
2. Настройте подключение через SSH с `backup_host` к `postgres_host`:
3. Настройте кластер баз данных для резервного копирования в режиме STREAM.
4. Проинициализируйте каталог резервных копий:

```
backup_user@backup_host:~$ pg_probackup init -B /mnt/backups
INFO: Backup catalog '/mnt/backups' successfully initialized
```

5. Добавьте экземпляр с именем `mydb` в каталог резервных копий:

```
backup_user@backup_host:~$ pg_probackup add-instance \
    -B /mnt/backups \
    -D /var/lib/pgpro/std-16/data \
    --instance=node \
    --remote-host=postgres_host \
    --remote-user=postgres
INFO: Instance 'node' successfully initialized
```

6. Сделайте полную резервную копию:

```
backup_user@backup_host:~$ pg_probackup backup \
    -B /mnt/backups \
    -b FULL \
    --instance=node \
    --stream \
    --compress-algorithm=zstd \
    --remote-host=postgres_host \
    --remote-user=postgres \
    -U backup \
    -d backupdb
INFO: Backup start, pg_probackup version: 2.7.3, instance: node, backup ID: SBOL6J,
 backup mode: FULL, wal mode: STREAM, remote: true, compress-algorithm: zstd,
 compress-level: 1
```

```
WARNING:  pgpro_edition() function is old-style and will be removed in future major
 release, use pgpro_edition GUC variable instead.
INFO: This PostgreSQL instance was initialized with data block checksums. Data
 block corruption will be detected
INFO: Database backup start
INFO: wait for pg_backup_start()
INFO: PGDATA size: 96MB
INFO: Current Start LSN: 0/8000028, TLI: 1
INFO: Start transferring data files
INFO: Data files are transferred, time elapsed: 2s
INFO: wait for pg_stop_backup()
INFO: pg_stop_backup() successfully executed
INFO: stop_stream_lsn 0/9000000 currentpos 0/9000000
INFO: backup->stop_lsn 0/8004E40
INFO: Getting the Recovery Time from WAL
INFO: Syncing backup files to disk
INFO: Backup files are synced, time elapsed: 1s
INFO: Validating backup SBOL6J
INFO: Backup SBOL6J data files are valid
INFO: Backup SBOL6J resident size: 53MB
INFO: Backup SBOL6J completed
```

7. Выведите список резервных копий экземпляра:

```
backup_user@backup_host:~$ pg_probackup show \
        -B /mnt/backups \
        --instance=node
=======================================================================================
 Instance  Version  ID      Recovery Time                  Mode  WAL Mode  TLI
 Time  Data   WAL  Zalg  Zratio  Start LSN  Stop LSN   Status
=======================================================================================
 node      17       SBOL6J  2024-04-09 18:18:21.970314+03  FULL  STREAM    1/0
 4s  37MB  16MB  zstd    2.57  0/8000028  0/8004E40  OK
```

8. Сделайте инкрементальную резервную копию в режиме DELTA:

```
backup_user@backup_host:~$ pg_probackup backup \
    -B /mnt/backups \
    -b DELTA \
    --instance=node \
    --stream \
    --compress-algorithm=zstd \
    --remote-host=postgres_host \
    --remote-user=postgres \
    -U backup \
    -d backupdb
INFO: Backup start, pg_probackup version: 2.7.3, instance: node, backup ID: SBOL6N,
 backup mode: DELTA, wal mode: STREAM, remote: true, compress-algorithm: zstd,
 compress-level: 1
WARNING:  pgpro_edition() function is old-style and will be removed in future major
 release, use pgpro_edition GUC variable instead.
INFO: This PostgreSQL instance was initialized with data block checksums. Data
 block corruption will be detected
INFO: Database backup start
INFO: wait for pg_backup_start()
INFO: Parent backup: SBOL6J
INFO: PGDATA size: 96MB
INFO: Current Start LSN: 0/9000028, TLI: 1
INFO: Parent Start LSN: 0/8000028, TLI: 1
```

```
INFO: Start transferring data files
INFO: Data files are transferred, time elapsed: 1s
INFO: wait for pg_stop_backup()
INFO: pg_stop_backup() successfully executed
INFO: stop_stream_lsn 0/A000000 currentpos 0/A000000
INFO: backup->stop_lsn 0/9000190
INFO: Getting the Recovery Time from WAL
INFO: Syncing backup files to disk
INFO: Backup files are synced, time elapsed: 0
INFO: Validating backup SBOL6N
INFO: Backup SBOL6N data files are valid
INFO: Backup SBOL6N resident size: 17MB
INFO: Backup SBOL6N completed
```

9. Добавьте или измените несколько параметров в файле конфигурации pg_probackup, чтобы их не надо было каждый раз указывать в командной строке:

```
backup_user@backup_host:~$ pg_probackup set-config \
    -B /mnt/backups \
    --instance=node \
    --remote-host=postgres_host \
    --remote-user=postgres \
    -U backup \
    -d backupdb
```

10. Проверьте конфигурацию экземпляра:

```
backup_user@backup_host:~$ pg_probackup show-config \
    -B /mnt/backups \
    --instance=node
# Backup instance information
pgdata = /var/lib/pgpro/std-16/data
system-identifier = 7355886958826772732
xlog-seg-size = 16777216
# Connection parameters
pgdatabase = backupdb
pghost = postgres_host
pguser = backup
# Archive parameters
archive-timeout = 5min
# Logging parameters
log-level-console = INFO
log-level-file = OFF
log-format-console = PLAIN
log-format-file = PLAIN
log-filename = pg_probackup.log
log-rotation-size = 0TB
log-rotation-age = 0d
# Retention parameters
retention-redundancy = 0
retention-window = 0
wal-depth = 0
# Compression parameters
compress-algorithm = none
compress-level = 1
# Remote access parameters
remote-proto = ssh
remote-host = postgres_host
remote-user = postgres
```

Обратите внимание, что параметры, не изменённые командой `set-config`, сохраняют значения по умолчанию.

11. Сделайте ещё одну инкрементальную копию в режиме DELTA, опустив параметры, ранее сохранённые в файле конфигурации:

```
backup_user@backup_host:~$ pg_probackup backup \
    -B /mnt/backups \
    -b DELTA \
    --instance=node \
    --stream \
    --compress-algorithm=zstd
INFO: Backup start, pg_probackup version: 2.7.3, instance: node, backup ID: SBOL6P,
 backup mode: DELTA, wal mode: STREAM, remote: true, compress-algorithm: zstd,
 compress-level: 1
WARNING:  pgpro_edition() function is old-style and will be removed in future major
 release, use pgpro_edition GUC variable instead.
INFO: This PostgreSQL instance was initialized with data block checksums. Data
 block corruption will be detected
INFO: Database backup start
INFO: wait for pg_backup_start()
INFO: Parent backup: SBOL6N
INFO: PGDATA size: 96MB
INFO: Current Start LSN: 0/A000028, TLI: 1
INFO: Parent Start LSN: 0/9000028, TLI: 1
INFO: Start transferring data files
INFO: Data files are transferred, time elapsed: 1s
INFO: wait for pg_stop_backup()
INFO: pg_stop_backup() successfully executed
INFO: stop_stream_lsn 0/B000000 currentpos 0/B000000
INFO: backup->stop_lsn 0/A000190
INFO: Getting the Recovery Time from WAL
INFO: Syncing backup files to disk
INFO: Backup files are synced, time elapsed: 0
INFO: Validating backup SBOL6P
INFO: Backup SBOL6P data files are valid
INFO: Backup SBOL6P resident size: 17MB
INFO: Backup SBOL6P completed
```

12. Вновь выведите список резервных копий экземпляра:

```
backup_user@backup_host:~$ pg_probackup show \
    -B /mnt/backups \
    --instance=node
=========================================================================================
 Instance  Version  ID      Recovery Time                  Mode   WAL Mode  TLI
 Time    Data   WAL  Zalg  Zratio  Start LSN  Stop LSN   Status
=========================================================================================
 node      17       SBOL6P  2024-04-09 18:18:26.630175+03  DELTA  STREAM    1/1
 1s  1147kB  16MB  zstd    1.00  0/A000028  0/A000190  OK
 node      17       SBOL6N  2024-04-09 18:18:25.015713+03  DELTA  STREAM    1/1
 2s  1160kB  16MB  zstd    1.04  0/9000028  0/9000190  OK
 node      17       SBOL6J  2024-04-09 18:18:21.970314+03  FULL   STREAM    1/0
 4s    37MB  16MB  zstd    2.57  0/8000028  0/8004E40  OK
```

13. Восстановите данные из последней доступной копии в произвольный каталог:

```
backup_user@backup_host:~$ pg_probackup restore \
    -B /mnt/backups \
    -D /var/lib/pgpro/std-16/staging-data \
```

```
    --instance=node
INFO: Validating parents for backup SBOL6P
INFO: Validating backup SBOL6J
INFO: Backup SBOL6J data files are valid
INFO: Validating backup SBOL6N
INFO: Backup SBOL6N data files are valid
INFO: Validating backup SBOL6P
INFO: Backup SBOL6P data files are valid
INFO: Backup SBOL6P WAL segments are valid
INFO: Backup SBOL6P is valid.
INFO: Restoring the database from the backup starting at 2024-04-09 18:18:25+03 on
 localhost
INFO: Start restoring backup files. PGDATA size: 112MB
INFO: Backup files are restored. Transferred bytes: 112MB, time elapsed: 2s
INFO: Restore incremental ratio (less is better): 100% (112MB/112MB)
INFO: Syncing restored files to disk
INFO: Restored backup files are synced, time elapsed: 1s
INFO: Restore of backup SBOL6P completed.
```

Установка и подготовка

Установив pg_probackup, выполните следующие действия:

- Инициализируйте каталог резервных копий.
- Добавьте копируемый экземпляр в каталог копий.
- Настройте кластер баз данных для использования pg_probackup.
- Настройте SSH для выполнения операций pg_probackup в удалённом режиме, если вы планируете его использовать.

Инициализация каталога резервных копий

pg_probackup сохраняет все файлы копируемых данных и WAL в соответствущих подкаталогах каталога резервных копий.

Для инициализации каталога резервных копий выполните команду:

```
pg_probackup init -B каталог_копий
```

здесь *`каталог_копий`* — это каталог, предназначенный для резервных копий. Если *`каталог_копий`* уже существует, он должен быть пустым. В противном случае pg_probackup выдаст ошибку.

Пользователь, запускающий pg_probackup, должен иметь полный доступ к *`каталогу_копий`*.

pg_probackup создаёт *`каталог_копий`* со следующими подкаталогами:

- `wal/` — каталог для файлов WAL.
- `backups/` — каталог для файлов резервных копий.

Проинициализировав каталог резервных копий, вы можете добавить определение копируемого экземпляра.

Определение копируемого экземпляра

pg_probackup может сохранять резервные копии разных кластеров баз данных в одном каталоге резервных копий. Для создания необходимых подкаталогов вы должны определить копируемый экземпляр в каталоге копий для каждого кластера баз данных, копию которого вы будете делать.

Для определения копируемого экземпляра выполните команду:

```
pg_probackup add-instance -B каталог_копий -D каталог_данных --instance=имя_экземпляра
 [параметры_удалённого_режима]
```

Здесь:

- *`каталог_данных`* — каталог, содержащий данные кластера, копию которого вы хотите сделать. Для подготовки и использования pg_probackup необходимо иметь право записи в этот каталог.
- *`имя_экземпляра`* — это имя подкаталогов, в которых будут храниться файлы копируемых данных и WAL для этого кластера.
- параметры_удалённого_режима должны задаваться дополнительно, если *`каталог_данных`* располагается удалённо.

pg_probackup создаёт подкаталоги *`имя_экземпляра`* в каталогах `backups/` и `wal/` внутри каталога резервных копий. Каталог `backups/`*`имя_экземпляра`* содержит файл конфигурации `pg_probackup.conf` с параметрами pg_probackup, относящимися к данному экземпляру копии. Если этой команде передать параметры_удалённого_режима, они будут добавлены в `pg_probackup.conf`.

Более подробно тонкая настройка pg_probackup описывается в Подразделе «Настройка pg_probackup».

Пользователь, запускающий pg_probackup, должен иметь полный доступ к *`каталогу_копий`* и как минимум доступ на чтение всего содержимого *`каталога_данных`*. Если вы зададите путь к каталогу копий в переменной окружения `BACKUP_PATH`, соответствующий параметр в командах pg_probackup можно не указывать.

> ### Примечание
>
> Для Postgres Pro версии 11 и новее рекомендуется использовать возможность Доступ группы, чтобы выполнить копирование мог любой пользователь ОС, включённый в группу владельца кластера. В этом случае пользователь должен иметь права на чтение каталога кластера.

Настройка кластера баз данных

Хотя pg_probackup можно использовать от имени суперпользователя, рекомендуется создать отдельную роль с минимальными правами, необходимыми для выбранной стратегии копирования. В этих инструкциях по настройке такой ролью служит роль `backup`.

Из соображений безопасности для выполнения следующих конфигурационных SQL-запросов рекомендуется использовать отдельную базу данных.

```
postgres=# CREATE DATABASE backupdb;
postgres=# \c backupdb
```

Для выполнения резервного копирования роль `backup` должна иметь следующие разрешения на сервере Postgres Pro (только в базе данных, к которой **производится подключение**).

Для версий Postgres Pro 11 — 14:

```
BEGIN;
CREATE ROLE backup WITH LOGIN;
GRANT USAGE ON SCHEMA pg_catalog TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.current_setting(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.set_config(text, text, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_is_in_recovery() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_start_backup(text, boolean, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_stop_backup(boolean, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_create_restore_point(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_switch_wal() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_last_wal_replay_lsn() TO backup;
```

```
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current_snapshot() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_snapshot_xmax(txid_snapshot) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_control_checkpoint() TO backup;
COMMIT;
```

Для Postgres Pro версии 15 или новее:

```
BEGIN;
CREATE ROLE backup WITH LOGIN;
GRANT USAGE ON SCHEMA pg_catalog TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.current_setting(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.set_config(text, text, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_is_in_recovery() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_backup_start(text, boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_backup_stop(boolean) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_create_restore_point(text) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_switch_wal() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_last_wal_replay_lsn() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_current_snapshot() TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.txid_snapshot_xmax(txid_snapshot) TO backup;
GRANT EXECUTE ON FUNCTION pg_catalog.pg_control_checkpoint() TO backup;
COMMIT;
```

В файле pg_hba.conf разрешите подключение к кластеру баз данных пользователю с ролью `backup`.

Программа pg_probackup должна читать непосредственно файлы кластера, поэтому запускать pg_probackup (или подключаться к нему удалённо) нужно от имени пользователя ОС, который имеет доступ на чтение всех файлов и каталогов внутри каталога данных кластера (`PGDATA`), подлежащего копированию.

В зависимости от того, будете ли вы делать самодостаточные или архивные копии, конфигурация кластера Postgres Pro будет разной (об особенностях рассказывается ниже). Чтобы получить копию кластера баз данных с ведомого сервера, запустить pg_probackup в удалённом режиме или создать резервную копию PTRACK, требуется дополнительная настройка.

Подробнее об этом рассказывается в подразделах Настройка потокового резервного копирования, Настройка непрерывного архивирования WAL, Настройка копирования с резервного сервера, Настройка удалённого режима, Настройка частичного восстановления и Настройка копирования PTRACK.

Настройка потокового резервного копирования

Чтобы настроить кластер для потокового резервного копирования, выполните следующие действия:

- Если роль `backup` не существует, создайте её с правом `REPLICATION` при Настройке кластера базы данных:

  ```
  CREATE ROLE backup WITH LOGIN REPLICATION;
  ```

- Если роль `backup` уже существует, дайте ей право `REPLICATION`:

  ```
  ALTER ROLE backup WITH REPLICATION;
  ```

- В файле pg_hba.conf разрешите выполнение репликации для роли `backup`.
- Установите для параметра max_wal_senders достаточно большое значение, предусматривающее минимум одно подключение для процесса резервного копирования.
- Задайте для параметра wal_level значение выше `minimal`.

Если вы намерены выполнять страничное резервное копирование в потоковом режиме или восстановление на момент времени с потоковыми копиями, вам тем не менее надо будет настроить архивирование WAL, как описано в подразделе Настройка непрерывного архивирования WAL.

После этих подготовительных действий вы можете делать резервные копии в режимах FULL, PAGE, DELTA и PTRACK, используя потоковую доставку WAL.

> **Примечание**
>
> Если вы намерены использовать .pgpass для прохождения аутентификации при выполнении копирования в потоковом режиме, файл `.pgpass` должен содержать учётные данные для подключения к базе данных `replication`. Например: `pghost:5432:replication:backup_user:my_strong_password`

Настройка непрерывного архивирования WAL

Для выполнения копирования в режиме PAGE, восстановления на момент времени и создания резервных копий с использованием режима доставки WAL ARCHIVE должно осуществляться непрерывное архивирование WAL. Чтобы настроить непрерывное архивирование, выполните следующие действия:

- Задайте для параметра wal_level значение выше `minimal`.
- Если вы настраиваете резервное копирование на ведущем сервере, параметр archive_mode должен иметь значение `on` или `always`. Для выполнения резервного копирования на ведомом требуется значение `always`.
- Установите параметр archive_command:

```
archive_command = '"путь_инсталляции/pg_probackup" archive-push -B "каталог_копий"
 --instance=имя_экземпляра --wal-file-name=%f [параметры_удалённого_режима]'
```

Здесь *путь_инсталляции* — путь к каталогу установленной версии pg_probackup, которую вы хотите использовать, *каталог_копий* и *имя_экземпляра* должны указывать на уже проинициализированный для данного кластера БД копируемый экземпляр, а параметры_удалённого_режима должны задаваться только в случае расположения архива WAL в удалённой системе. Подробнее все возможные параметры `archive-push` рассматриваются в описании archive-push.

После этих подготовительных действий вы сможете использовать режим доставки WAL ARCHIVE и делать резервные копии в режиме PAGE, а также выполнять восстановление на момент времени.

Вы можете просмотреть текущее состояние архива WAL, воспользовавшись командой show. За подробностями обратитесь к Подразделу «Просмотр оглавления архива WAL».

Если вы планируете выполнять страничное копирование и/или делать копии с ведомого сервера, используя режим доставки WAL ARCHIVE, при недостаточной транзакционной активности может потребоваться долго ждать заполнения очередного сегмента WAL. Чтобы ограничить время ожидания, вы можете воспользоваться параметром Postgres Pro archive_timeout на **ведущем** сервере. Значение этого параметра должно быть меньше значения `--archive-timeout` (по умолчанию 5 минут), чтобы заполненный сегмент успел передаться ведомому серверу и попасть в архив WAL, прежде чем копирование прервётся по тайм-ауту, заданному параметром `--archive-timeout`.

> **Примечание**
>
> Вместо использования команды pg_probackup archive-push вы можете воспользоваться любым другим средством, при условии, что в процессе непрерывного архивирования сегменты WAL будут попадать в каталог *каталог_копий*/wal/*имя_экземпляра*. Для сжатия сегментов,

если в нём есть потребность, должен использоваться алгоритм `gzip`, а сжатые файлы сегментов должны иметь расширение `.gz`.

Примечание

Организовать непрерывное архивирование можно не только с помощью параметров `archive_mode` и `archive_command`, но и применяя утилиту pg_receivewal. В этом случае аргумент pg_receivewal `-D` *`каталог`* должен указывать на каталог *`каталог_копий`*`/wal/`*`имя_экземпляра`*. Программа pg_probackup принимает сжатые WAL, которые может сохранять pg_receivewal. Стратегию архивирования «без потерь» можно реализовать только с использованием pg_receivewal.

Настройка копирования с ведомого сервера

Утилита pg_probackup может копировать данные с ведомого сервера. Для этого требуется дополнительная настройка:

- Установите на ведомом сервере для параметра hot_standby значение `on`.
- На ведущем сервере установите для параметра full_page_writes значение `on`.
- Для получения самодостаточных копий с ведомого сервера выполните действия, описанные в подразделе Настройка потокового копирования.
- Для получения архивных копий с ведомого сервера выполните действия, описанные в подразделе Настройка непрерывного архивирования WAL.

После этих подготовительных действий вы можете делать резервные копии с ведомого сервера в режимах FULL, PAGE, DELTA или PTRACK, используя режим доставки WAL ARCHIVE или STREAM.

Копирование с ведомого сервера имеет следующие ограничения:

- Если ведомый сервер повышается до ведущего во время копирования, копирование прерывается с ошибкой.
- Все необходимые для резервной копии записи WAL должны содержать полные страницы. Для этого требуется включить режим `full_page_writes` на ведущем сервере и отказаться от использования в archive_command утилит, удаляющих полные страницы из файлов WAL, как, например, pg_compresslog.

Настройка проверки целостности кластера

Для логической проверки кластера БД требуется дополнительная настройка (предполагается, что настройка производится для роли `backup`):

- Установите расширение amcheck или *amcheck_next* **в каждой базе** кластера:

  ```
  CREATE EXTENSION amcheck;
  ```

- Дайте роли `backup` **в каждой базе данных** кластера следующие права:

```
GRANT SELECT ON TABLE pg_catalog.pg_am TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_class TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_database TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_namespace TO backup;
GRANT SELECT ON TABLE pg_catalog.pg_extension TO backup;
GRANT EXECUTE ON FUNCTION bt_index_check(regclass) TO backup;
GRANT EXECUTE ON FUNCTION bt_index_check(regclass, bool) TO backup;
GRANT EXECUTE ON FUNCTION bt_index_check(regclass, bool, bool) TO backup;
```

Настройка частичного восстановления

Если вы планируете производить частичное восстановление, выполните следующее дополнительное действие:

- Дайте право на чтение `pg_catalog.pg_database` роли `backup` в той базе данных, через которую **выполняется подключение** к серверу Postgres Pro:

  ```
  GRANT SELECT ON TABLE pg_catalog.pg_database TO backup;
  ```

Настройка удалённого режима

Программа pg_probackup поддерживает удалённый режим, в котором резервное копирование, восстановление и архивирование WAL могут выполняться удалённо. В этом режиме каталог резервных копий находится в локальной системе, а целевой экземпляр Postgres Pro работает на удалённом компьютере. В настоящее время это поддерживается с использованием только одного сетевого протокола — SSH.

Настройте SSH

Если вы хотите использовать pg_probackup в удалённом режиме (применяя SSH), выполните следующие действия:

1. Установите pg_probackup в обеих системах: `backup_host` (сервер резервного копирования) и `postgres_host` (сервер баз данных).
2. Для организации связи между узлами настройте подключение по SSH пользователя `backup_user` в системе `backup_host` к системе `postrges_host` под именем пользователя `postgres` без указания пароля:

   ```
   backup_user@backup_host:~$ ssh-copy-id postgres@postgres_host
   ```

   Здесь:

   - `backup_host` — система, в которой находится *каталог копий*.
   - `postrges_host` — система, в которой функционирует кластер Postgres Pro.
   - `backup_user` — пользователь в системе `backup_host`, от имени которого запускается pg_probackup.
   - `postgres` — пользователь в системе `postgres_host`, от имени которого запускаются процессы кластера Postgres Pro. Для Postgres Pro версии 11 и новее можно применить более безопасный подход, воспользовавшись возможностью предоставления доступа группе.
3. Если вы будете использовать непрерывное архивирование WAL, настройте подключение по SSH пользователя `postgres` в системе `postgres_host` к системе `backup_host` под именем пользователя `backup` без указания пароля:

   ```
   postgres@postgres_host:~$ ssh-copy-id backup_user@backup_host
   ```
4. Убедитесь, что pg_probackup в системе `postgres_host` можно найти при установке соединения по SSH. Например, чтобы обеспечить это в Bash можно изменить `PATH` в файле `~/.bashrc` пользователя `postgres` (над строкой в `bashrc`, которая завершает скрипт для неинтерактивных оболочек). Либо для команд pg_probackup можно указать в параметре --remote-path путь к каталогу, содержащему программу pg_probackup в системе `postgres_host`.

pg_probackup работает в удалённом режиме по протоколу SSH следующим образом:

- В удалённом режиме поддерживаются только следующие команды: add-instance, backup, restore, delete, catchup, archive-push и archive-get.
- Для работы в удалённом режиме необходимо, чтобы программа pg_probackup была установлена и в локальной, и в удалённой системе, при этом одной и той же версии.
- При запуске в удалённом режиме основной процесс pg_probackup в локальной системе подключается к удалённой по протоколу SSH и запускает в удалённой системе один или несколько агентов, которые называются *удалёнными агентами*. Количество удалённых агентов определяется значением параметра `-j`/`--threads`.
- Основной процесс pg_probackup использует удалённые агенты для передачи данных между локальной и удалённой системой и для обращения к её файлам.

- Удалённые агенты стараются минимизировать сетевой трафик и количество операций взаимодействия узлов.
- Основной процесс обычно запускается в системе *`backup_host`* и подключается к системе *`postgres_host`*, но для команд `archive-push` и `archive-get` основной процесс запускается в системе *`postgres_host`* и подключается к *`backup_host`*.
- После завершения передачи данных удалённые агенты завершаются и подключения SSH закрываются.
- Если какой-либо удалённый агент столкнулся с ошибкой, все агенты завершаются, а основной процесс pg_probackup выдаёт информацию о возникшей ошибке и также нештатно завершается.
- Сжатие данных всегда осуществляется в системе *`postgres_host`*, а распаковка — в системе *`backup_host`*.

Примечание

Вы можете задать дополнительные *ограничительные параметры* SSH для защиты целевой системы в случае компрометации используемой учётной записи.

Примечание

Если для pg_probackup, работающего в удалённом режиме через SSH, задать число потоков больше 10 (параметр `-j`/`--threads`), то фактическое число SSH-соединений может превысить максимально допустимое число одновременных SSH-соединений на удалённом сервере и, следовательно, приведёт к ошибке «ERROR: Agent error: kex_exchange_identification: Connection closed by remote host» (ОШИБКА: Ошибка агента: kex_exchange_identification: Соединение закрыто удалённым сервером). Чтобы исправить ошибку, либо уменьшите количество потоков pg_probackup, либо измените значение параметра конфигурации `MaxStartups` удалённого SSH-сервера. Если SSH настроен как служба xinetd на удалённом сервере, измените значение параметра конфигурации xinetd `per_source`, а не `MaxStartups`.

Настройка копирования PTRACK

Примечание

Версии PTRACK ниже 2.0 признаны устаревшими и не поддерживаются. В Postgres Pro Standard и Postgres Pro Enterprise версии 11.9.1 и новее входит обновлённый PTRACK 2.0. Обновите ваш сервер, чтобы избежать проблем с резервными копиями, которые вы будете делать в дальнейшем, и обязательно сделайте копии ваших кластеров с обновлённым PTRACK, так как копии, полученные с использованием PTRACK 1.x, могут быть испорченными.

Если вы хотите копировать данные в режиме PTRACK, выполните следующие дополнительные действия.

Примечание

Для роли, которая будет выполнять резервное копирование в режиме PTRACK (роль `backup` в примерах ниже), требуемые права доступа указаны в Подразделе «Настройка кластера баз данных». Роль должна иметь права только в той базе данных, которая используется для подключения к серверу Postgres Pro.

Для Postgres Pro версии 11 или новее:

1. Создайте расширение PTRACK в базе данных, *используемой для подключения* к серверу Postgres Pro:

   ```
   CREATE EXTENSION ptrack;
   ```

2. Чтобы включить отслеживание изменений страниц, задайте для параметра `ptrack.map_size` положительное целое значение и перезапустите сервер.

   Для оптимальной производительности рекомендуется задавать `ptrack.map_size` равным `N / 1024`, где *N* — объём кластера Postgres Pro в мегабайтах. Если он будет иметь меньшее значение, это увеличит вероятность наложения информации разных блоков в карте PTRACK, что повлечёт ложные положительные результаты при определении изменённых блоков и, как следствие, увеличение размера инкрементальной копии, так как в копию будут попадать и фактически неизменённые блоки. Использовать значения `ptrack.map_size`, превышающие 1024, не рекомендуется, хотя PTRACK поддерживает большие карты.

Примечание

В случае изменения значения `ptrack.map_size` ранее созданный файл карты PTRACK очищается, и отслеживание новых блоков начинается с начала. Таким образом, прежде чем создавать новые инкрементальные копии в режиме PTRACK после изменения `ptrack.map_size` необходимо сделать новую полную копию кластера.

Использование

Создание резервной копии

Для создания резервной копии выполните следующую команду:

```
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b режим_копирования
```

Здесь *режим_копирования* может принимать следующие значения: `FULL`, `DELTA`, `PAGE`, и `PTRACK`.

При восстановлении кластера из инкрементальной копии pg_probackup использует родительскую полную копию и все инкрементальные копии между ними, которые в совокупности образуют «цепочку копий». Таким образом, прежде чем делать инкрементальные копии, необходимо сделать как минимум одну полную со статусом `OK` или `DONE` в каталоге. Если родительская полная копия имеет статус `MERGING` или `MERGED`, выполнить инкрементальное резервное копирование невозможно.

Например, если объединение уже было запущено с одной полной копией, попытка выполнить инкрементальное резервное копирование завершится выводом следующих сообщений:

```
ВНИМАНИЕ: Рабочая полная резервная копия не найдена в текущей линии времени, идёт поиск
 в предыдущих линиях времени
ВНИМАНИЕ: Невозможно найти рабочую резервную копию в предыдущих линиях времени
ОШИБКА: Создайте новую полную резервную копию до инкрементальной
```

Режим ARCHIVE

Режим ARCHIVE используется в качестве режима доставки WAL по умолчанию.

Например, чтобы сделать полную копию в режиме доставки WAL ARCHIVE, выполните:

```
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b FULL
```

Резервное копирование ARCHIVE требует организации непрерывного архивирования, посредством которого считываются сегменты WAL, требующиеся для восстановления согласованного состояния кластера на момент создания копии.

В процессе резервного копирования pg_probackup помещает файлы WAL, содержащие записи от `Start LSN` до `Stop LSN`, в каталог *каталог_копий*/wal/*имя_экземпляра*. pg_probackup также проверяет, что записи WAL между `Start LSN` и `Stop LSN` корректно читаются. Это позволяет исключить риск хранения в архиве повреждённого WAL.

Режим STREAM

Режим STREAM может использоваться в качестве альтернативного режима доставки WAL.

Например, чтобы сделать полную резервную копию в потоковом режиме, добавьте флаг `--stream` к команде, показанной выше:

```
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b FULL --stream --temp-slot
```

Необязательный параметр `--temp-slot` обеспечивает наличие необходимых сегментов в случае прокрутки WAL до завершения резервного копирования.

В отличие от копий ARCHIVE, копии типа STREAM включают все сегменты WAL, необходимые для восстановления согласованного состояния кластера на момент создания копии.

В процессе выполнения команды backup программа pg_probackup передаёт файлы WAL, содержащие записи от `Start LSN` до `Stop LSN` в каталог *каталог_копий*/backups/*имя_экземпляра*/*ид_резервной_копии*/database/pg_wal. Чтобы исключить риск архивирования испорченных файлов WAL, pg_probackup также проверяет, что записи WAL от `Start LSN` до `Stop LSN` прочитываются корректно.

Даже если вы используете непрерывное архивирование, копирование в режиме STREAM может быть полезно в следующих случаях:

- Копии типа STREAM могут быть восстановлены на сервере, не имеющем файлового доступа к архиву WAL.
- Копии типа STREAM позволяют восстановить состояние кластера на тот момент времени, для которого уже нет файлов WAL.
- Копирование в режиме STREAM можно производить с ведомого сервера, не дожидаясь заполнения сегментов WAL, что могло бы занять длительное время при небольшом объёме трафика WAL.

Проверка страниц

Когда в кластере БД включены контрольные суммы, pg_probackup использует их для проверки целостности файлов данных в процессе резервного копирования. При чтении каждой страницы pg_probackup проверяет, совпадает ли вычисленная сумма с контрольной суммой, хранящейся в заголовке страницы. Это гарантирует, что в кластере Postgres Pro и самой резервной копии не содержатся испорченные страницы. Заметьте, что pg_probackup читает файлы данных непосредственно из файловой системы, поэтому при активной записи в момент копирования возможны ложные выявления некорректных контрольных сумм из-за частичной записи. В случае несовпадения контрольной суммы страница считывается повторно, и контрольная сумма проверяется ещё раз.

Страница признаётся испорченной, если проверка контрольной суммы не проходит более 300 раз. В этом случае резервное копирование прерывается.

Даже если контрольные суммы не включены, pg_probackup всегда проверяет целостность заголовков страниц.

Внешние каталоги

Чтобы заархивировать каталог, размещённый вне каталога данных, воспользуйтесь необязательным параметром `--external-dirs`, в котором можно задать путь к нужному каталогу. Если вы хотите заархивировать несколько внешних каталогов, их пути нужно разделить двоеточиями в Linux или точками с запятой в Windows.

Например, чтобы в системе Linux включить каталоги `/etc/dir1` и `/etc/dir2` в полную копию экземпляра *имя_экземпляра*, которая будет размещаться в *каталоге_копий*, выполните:

```
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b FULL --external-
dirs=/etc/dir1:/etc/dir2
```

Например, чтобы включить каталоги `C:\dir1` и `C:\dir2` в полную копию, в Windows нужно выполнить:

```
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b FULL --external-
dirs=C:\dir1;C:\dir2
```

Для каждого внешнего каталога pg_probackup создаёт отдельный подкаталог в каталоге резервной копии и рекурсивно копирует в него всё содержимое внешнего каталога. Так как внешние каталоги, попадающие в разные резервные копии, не обязательно должны быть одинаковыми, при восстановлении кластера из инкрементальной копии будут восстановлены только те каталоги, которые относятся именно к ней. Внешние каталоги, сохранённые в предыдущих копиях, восстановлены не будут.

Чтобы нужные каталоги включались в каждую резервную копию вашего кластера, их список можно сохранить в файле конфигурации `pg_probackup.conf`, воспользовавшись командой set-config с ключом `--external-dirs`.

Проверка целостности кластера

Чтобы убедиться в отсутствии повреждений в кластере Postgres Pro, выполните следующую команду:

```
pg_probackup checkdb [-B каталог_копий [--instance=имя_экземпляра]] [-D каталог_данных]
 [параметры_подключения]
```

Эта команда выполняет физическую проверку всех файлов данных в указанном каталоге данных, проводя проверки целостности заголовков страниц, а также проверяя контрольные суммы на уровне блоков, если в кластере включены контрольные суммы. В случае обнаружения испорченной страницы `checkdb` продолжает работу, пока не будут проверены все страницы в кластере.

По умолчанию pg_probackup выполняет подобную проверку страницы автоматически в процессе создания копии. Команда `checkdb` позволяет при желании проводить проверки страниц в кластере, не создавая резервные копии (при этом даже необязательно настраивать копирование кластера в pg_probackup).

Чтобы провести проверку кластера, приложение pg_probackup должно подключиться к нему. Обычно для получения требуемых параметров подключения достаточно указать связанный с данным кластером копируемый экземпляр. Однако если параметры `-B` и `--instance` опускаются, параметры_соединения и *каталог_данных* необходимо задать в командной строке или в переменных окружения.

Важно понимать, что проверка на физическом уровне не позволяет выявить логические несоответствия, потерю или обнуление блоков или целых файлов и другие подобные аномалии. Провести логическую проверку в некотором объёме позволяют расширения amcheck и *amcheck_next*.

Если вы хотите помимо физической проверки проверить также все индексы во всех базах, используя эти расширения, вы можете передать команде checkdb флаг `--amcheck`:

```
pg_probackup checkdb -D каталог_данных --amcheck [параметры_соединения]
```

Проверку на физическом уровне можно отключить, воспользовавшись флагом `--skip-block-validation`. В этом случае задавать *каталог_копий* и *каталог_данных* не нужно, требуется задать только параметры подключения:

```
pg_probackup checkdb --amcheck --skip-block-validation [параметры_соединения]
```

Логическая проверка может быть более доскональной, если запустить её с ключом `--heapallindexed`. В этом случае будет дополнительно проверено, что в индексе действительно представлены все кортежи кучи, которые должны в него попасть, но это создаст дополнительную нагрузку на процессор, память и подсистему ввода-вывода.

Проверка резервных копий

pg_probackup вычисляет контрольные суммы для всех файлов копии в ходе резервного копирования. Процесс проверки контрольных сумм файлов называется *проверкой целостности копии*. По умолчанию проверка выполняется сразу после создания резервной копии и непосредственно перед восстановлением для выявления возможных повреждений резервных копий.

Если вы хотите пропустить проверку резервной копии, вы можете передать командам backup и restore флаг `--no-validate`.

Чтобы убедиться, что все необходимые файлы резервных копий имеются в наличии и что, используя их, можно восстановить кластер баз данных, вы можете запустить команду validate с теми же параметрами точки восстановления, с которыми вы будете производить восстановление.

Например, чтобы убедиться, что вы можете восстановить кластер баз данных из резервной копии, остановившись на транзакции с идентификатором 4242, выполните команду:

```
pg_probackup validate -B каталог_копий --instance=имя_экземпляра --recovery-target-
xid=4242
```

Если проверка проходит успешно, pg_probackup выдаёт сообщение об этом. В случае же неудачи вы получите сообщение об ошибке с указанием точного времени, идентификатора транзакции и значения LSN, до которого возможно восстановление.

Если вы укажете *идентификатор копии* в ключе `-i`/`--backup-id`, будет проверена только резервная копия с указанным идентификатором. Если *идентификатор копии* указывается вместе с параметрами точки восстановления, команда `validate` проверит, возможно ли восстановить указанную резервную копию до заданной точки.

Например, чтобы убедиться, что можно восстановить кластер баз данных из резервной копии с идентификатором `SBOL6P` до заданного момента времени, выполните команду:

```
pg_probackup validate -B каталог_копий --instance=имя_экземпляра -i SBOL6P --recovery-
target-time="2024-04-10 18:18:26+03"
```

Если вы укажете *ид_резервной_копии*, относящийся к инкрементальной копии, будут проверены все нужные ей родительские копии, начиная с полной.

Если вы опустите все параметры, будут проверены все резервные копии.

Восстановление кластера

Чтобы восстановить кластер баз данных из резервной копии, выполните команду restore как минимум со следующими параметрами:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра -i ид_резервной_копии
```

Здесь:

- *каталог_копий* — каталог, в котором хранятся все файлы резервных копий и метаданные.
- *имя_экземпляра* — имя экземпляра резервной копии кластера, который будет восстановлен.
- *ид_резервной_копии* определяет, из какой резервной копии будет восстановлен кластер. Если этот параметр опускается, pg_probackup использует последнюю подходящую копию для заданного экземпляра. Если вы выбираете для восстановления инкрементальную копию, pg_probackup автоматически восстанавливает нижележащую полную копию и затем последовательно применяет все необходимые добавления.

После окончания работы команды `restore` запустите службу базы данных.

Если вы восстанавливаете копию ARCHIVE, выполняете восстановление PITR или передаёте флаг `--restore-as-replica` команде `restore` для настройки ведомого сервера, pg_probackup создаёт файл конфигурации восстановления после копирования всех файлов данных в целевой каталог. Этот файл включает необходимые для восстановления параметры, за исключением пароля, заданного в primary_conninfo; если он требуется, его нужно дополнительно задать вручную или воспользоваться параметром `--primary-conninfo`. Для Postgres Pro версии 11 параметры восстановления сохраняются в файле `recovery.conf`, но для версий Postgres Pro, начиная с 12, pg_probackup сохраняет эти параметры в файле `probackup_recovery.conf` в каталоге данных и подключает его в `postgresql.auto.conf`.

Если восстанавливалась копия типа STREAM, восстановление завершается сразу, и кластер возвращается в согласованное состояние на момент времени, в который была сделана резервная копия. Для копий типа ARCHIVE Postgres Pro воспроизводит все имеющиеся в архиве сегменты WAL, в результате чего восстанавливается самое последнее состояние кластера на текущей линии времени. Это поведение можно изменить, определив параметры точки восстановления для команды `restore` как описывается в Подразделе «Выполнение восстановления на момент времени (PITR)».

Если кластер, подлежащий восстановлению, содержит табличные пространства, pg_probackup по умолчанию восстанавливает их в исходные расположения. Чтобы сменить расположения табличных пространств, воспользуйтесь параметром `--tablespace-mapping`/`-T`. В противном случае при восстановлении кластера на том же сервере произойдёт ошибка, если эти табличные пространства будут использоваться, так как восстанавливаемые данные нужно будет записать в те же каталоги.

Используя параметр `--tablespace-mapping`/`-T`, вы должны задать абсолютные пути к старому и новому каталогу табличного пространства. Если путь содержит знак равно (=), экранируйте его обратной косой чертой. Данный параметр может указываться неоднократно для перемещения нескольких табличных пространств. Например:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра -D каталог_данных -j 4
 -i ид_резервной_копии -T каталог_табл_пространства1=новый_каталог_табл_пространства1 -
T каталог_табл_пространства2=новый_каталог_табл_пространства2
```

Восстановление кластера на удалённом компьютере описано в Подразделе «Использование pg_probackup в удалённом режиме».

> **Примечание**
>
> По умолчанию команда restore проверяет указанную резервную копию перед восстановлением кластера. Если вы проводите проверку копий регулярно и хотели бы сэкономить время при восстановлении кластера, вы можете добавить ключ `--no-validate` для пропуска проверки и ускорения восстановления.

Инкрементальное восстановление

Скорость восстановления резервной копии можно значительно увеличить, заменяя в существующем каталоге данных Postgres Pro только некорректные или изменённые страницы. Это можно реализовать, используя параметры инкрементального восстановления с командой restore.

Чтобы восстановить кластер баз данных из резервной копии в инкрементальном режиме, выполните команду restore со следующими параметрами:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра -D каталог_данных -
I инкрементальный_режим
```

Здесь *инкрементальный_режим* может быть следующим:

- `CHECKSUM` — прочитать все файлы данных в целевом каталоге, проверить заголовок и контрольную сумму каждой страницы и заменить только некорректные страницы, а также страницы, в которых контрольная сумма и LSN отличаются от значений в соответствующей странице в копии. Это самый простой и надёжный инкрементальный режим. Его рекомендуется использовать по умолчанию.
- `LSN` — прочитать файл *`pg_control`* в каталоге данных, получить из него значения REDO LSN и REDO TLI, позволяющие определить точку в истории (точку сдвига), в которой состояние каталога данных сдвинулись с цепочки резервных копий. Если точка сдвига находится за пределами истории резервных копий, восстановление прерывается. Если же точка сдвига достижима, прочитываются все файлы данных в каталоге данных, проверяется заголовок и контрольная сумма на каждой странице, а затем заменяются только страницы с неверной контрольной суммой или с LSN, превышающим позицию точки сдвига. В этом режиме обеспечивается увеличенная скорость по сравнению с режимом CHECKSUM, но требуется выполнение двух условий. Во-первых, в целевом каталоге должны быть включены контрольные суммы (см. data_checksums), чтобы не произошло повреждения данных из-за изменения вспомогательных битов. Это условие будет проверяться перед инкрементальным восстановлением — в случае отсутствия контрольных сумм операция прерывается. Во-вторых, необходима синхронность файла `pg_control` с состоянием каталога данных. Это условие нельзя проверить в начале восстановления, так что гарантировать правильность информации в `pg_control` должен пользователь. Поэтому использовать режим LSN рекомендуется не во всех ситуациях, например, он противопоказан, когда файл `pg_control` повреждён или его содержимому нельзя доверять: после выполнения `pg_resetxlog`, после восстановления из копии без процедуры воспроизведения журнала и т. д.
- `NONE` — обычное восстановление без инкрементальных оптимизаций.

Вне зависимости от выбранного инкрементального режима, pg_probackup будет проверять, что с заданным целевым каталогом не работает процесс postmaster и значения `system-identifier` в целевом экземпляре и копии совпадают.

Предположим, вы хотите восстановить старый ведущий сервер в виде реплики после переключения, используя инкрементальное восстановление в режиме LSN:

```
============================================================================================
 Instance  Version  ID      Recovery Time                  Mode    WAL Mode  TLI  Time
  Data   WAL  Zalg  Zratio  Start LSN   Stop LSN    Status
============================================================================================
 node      17       SBOL8S  2024-04-09 18:19:43.707720+03  DELTA   STREAM    16/15
 3s  114MB  64MB   lz4    1.42  0/3C003020  0/3E8D4930  OK
 node      17       SBOL8G  2024-04-09 18:19:32.594670+03  PTRACK  STREAM    15/15
 4s   30MB  16MB  zlib    2.23  0/31000028  0/310029E0  OK
 node      17       SBOL83  2024-04-09 18:19:22.269595+03  PAGE    STREAM    15/15
 7s   46MB  32MB  pglz    1.44  0/29000028  0/2A0000F8  OK
 node      17       SBOL7P  2024-04-09 18:19:06.557301+03  FULL    STREAM    15/0
 6s  144MB  16MB  zstd    2.47  0/22000028  0/220001C8  OK

backup_user@backup_host:~$ pg_probackup restore -B /mnt/backups --instance=node -R -I
 lsn
INFO: Destination directory and tablespace directories are empty, disable incremental
 restore
INFO: Validating parents for backup SBOL8S
INFO: Validating backup SBOL7P
INFO: Backup SBOL7P data files are valid
INFO: Validating backup SBOL83
INFO: Backup SBOL83 data files are valid
INFO: Validating backup SBOL8G
INFO: Backup SBOL8G data files are valid
INFO: Validating backup SBOL8S
INFO: Backup SBOL8S data files are valid
```

```
INFO: Backup SBOL8S WAL segments are valid
INFO: Backup SBOL8S is valid.
INFO: Restoring the database from the backup starting at 2024-04-09 18:19:40+03
INFO: Start restoring backup files. PGDATA size: 616MB
INFO: Backup files are restored. Transferred bytes: 616MB, time elapsed: 2s
INFO: Restore incremental ratio (less is better): 100% (616MB/616MB)
INFO: Syncing restored files to disk
INFO: Restored backup files are synced, time elapsed: 2s
INFO: Restore of backup SBOL8S completed.
```

Примечание

Инкрементальное восстановление возможно только для копий, у которых `program_version` больше или равна 2.4.0.

Частичное восстановление

Если вы подготовились к частичному восстановлению до того, как сделали резервную копию, вы можете восстановить только некоторые базы данных, используя параметры частичного восстановления с командой restore.

Чтобы восстановить только некоторые базы данных, выполните команду `restore` как минимум со следующими параметрами:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра --db-include=имя_базы
```

Ключ `--db-include` может добавляться многократно. Например, чтобы восстановить только базы `db1` и `db2`, выполните следующую команду:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра --db-include=db1 --db-include=db2
```

Чтобы исключить одну или несколько баз из числа восстанавливаемых, используйте ключ `--db-exclude`:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра --db-exclude=имя_базы
```

Ключ `--db-exclude` может добавляться многократно. Например, чтобы при восстановлении исключить базы данных `db1` и `db2`, выполните следующую команду:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра --db-exclude=db1 --db-exclude=db2
```

Механизм частичного восстановления рассчитывает на то, что в процессе восстановления Postgres Pro наличие пустых файлов не будет проблемой, так как файлы исключённых баз данных восстанавливаются пустыми. После успешного запуска кластера Postgres Pro восстановленные определения исключённых баз данных можно удалить с помощью команды `DROP DATABASE`.

Чтобы максимально быстро разделить один кластер, содержащий несколько баз данных, на разные кластеры, можно выполнить частичное восстановление исходного кластера в виде ведомого, передав ключ `--restore-as-replica` для определённых баз данных.

Примечание

Базы `template0` и `template1` восстанавливаются всегда.

Примечание

Из-за особенностей восстановления, присущих версиям Postgres Pro до 12, при частичном восстановлении кластера Postgres Pro этих версий рекомендуется установить для параметра hot_standby значение `off`. В противном случае при восстановлении возможен сбой.

Выполнение восстановления на момент времени (PITR)

Если вы настраивали непрерывное архивирование WAL до создания резервных копий, вы можете восстановить состояние кластера на любой момент времени (до заданной точки восстановления), используя с командой restore параметры точки восстановления.

Для восстановления на момент времени может использоваться копия типа STREAM или ARCHIVE, но при этом обязательно наличие архива WAL с момента создания копии или более раннего. Если параметр `-i`/`--backup-id` не задан, pg_probackup автоматически выбирает резервную копию, ближайшую к заданной цели восстановления, и начинает процесс восстановления. В противном случае pg_probackup попытается восстановить до заданной цели восстановления именно копию с заданным *`ид_резервной_копии`*.

- Чтобы восстановить состояние кластера на определённый момент времени, укажите это время в параметре `--recovery-target-time`, в формате timestamp. Например:

  ```
  pg_probackup restore -B каталог_копий --instance=имя_экземпляра --recovery-target-
  time="2024-04-10 18:18:26+03"
  ```

- Чтобы восстановить состояние кластера до определённой транзакции, воспользуйтесь ключом `--recovery-target-xid`:

  ```
  pg_probackup restore -B каталог_копий --instance=имя_экземпляра --recovery-target-
  xid=687
  ```

- Чтобы восстановить состояние кластера до определённой позиции в журнале (LSN), воспользуйтесь ключом `--recovery-target-lsn`:

  ```
  pg_probackup restore -B каталог_копий --instance=имя_экземпляра --recovery-target-
  lsn=16/B374D848
  ```

- Чтобы восстановить состояние кластера до заданной именованной точки восстановления, воспользуйтесь ключом `--recovery-target-name`:

  ```
  pg_probackup restore -B каталог_копий --instance=имя_экземпляра --recovery-target-
  name="before_app_upgrade"
  ```

- Чтобы восстановить последнее возможное состояние, исходя из содержимого архива WAL, передайте в параметре `--recovery-target` значение `latest`:

  ```
  pg_probackup restore -B каталог_копий --instance=имя_экземпляра --recovery-
  target="latest"
  ```

- Чтобы восстановить самое раннее из возможных согласованное состояние кластера, передайте в параметре `--recovery-target` значение `immediate`:

  ```
  pg_probackup restore -B каталог_копий --instance=имя_экземпляра --recovery-
  target='immediate'
  ```

Использование pg_probackup в удалённом режиме

Программа pg_probackup поддерживает работу в удалённом режиме, то есть может выполнять операции `backup` и `restore` удалённо, используя SSH. В этом режиме каталог резервных копий располагается в локальной системе, а целевой экземпляр Postgres Pro работает в удалённой. При этом pg_probackup должен быть установлен в обеих системах.

Примечание

pg_probackup рассчитывает на то, что узлы будут взаимодействовать между собой с использованием SSH без пароля.

Примечание

Помимо подключения по SSH, pg_probackup использует для управления удалёнными операциями обычное подключение к базе данных. За подробной информацией о настройке подключения к базе данных обратитесь к разделу Настройка кластера базы данных.

Типичная схема его использования выглядит так:

- В системе резервного копирования настройте pg_probackup, как описывается в подразделе Установка и настройка. Для команд add-instance и set-config необходимо задать параметры удалённого режима, указывающие на сервер с экземпляром Postgres Pro.
- Если вы хотите в удалённом режиме выполнять страничное копирование (PAGE), использовать доставку WAL в режиме ARCHIVE или осуществлять восстановление PITR, настройте непрерывное архивирование WAL с сервера БД в систему резервного копирования, как описано в подразделе Настройка непрерывного архивирования WAL. Для этого в командах archive-push и archive-get требуется задать параметры удалённого режима, указывающие на сервер, где находится каталог резервных копий.
- Запустите команду backup или restore с параметрами удалённого режима **в системе резервного копирования**. pg_probackup подключится к удалённой системе по SSH и соответственно создаст резервную копию в локальной системе или восстановит в удалённой системе ранее сделанную копию.

Например, чтобы создать полную архивную копию кластера Postgres Pro, работающего в удалённой системе с адресом `192.168.0.2`, подключившись к серверу по SSH через порт `2302` с именем пользователя `postgres`, выполните:

```
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b FULL --remote-
user=postgres --remote-host=192.168.0.2 --remote-port=2302
```

Чтобы восстановить последнюю имеющуюся копию в удалённой системе с адресом `192.168.0.2`, подключившись к серверу по SSH через порт `2302` с именем пользователя `postgres`, выполните:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра --remote-user=postgres
 --remote-host=192.168.0.2 --remote-port=2302
```

Для восстановления архивных копий или восстановления на момент времени в удалённом режиме требуется дополнительная информация: целевой адрес, порт и имя пользователя для установления SSH-подключения **с** сервера БД **к** системе, где находится каталог копий. Эту информацию будет использовать команда `restore_command` для копирования сегментов WAL из архива в каталог Postgres Pro `pg_wal`.

Для передачи этой информации вы можете задать параметры удалённого архива WAL.

Например, для восстановления последней копии в удалённой системе, настроенной для приёма подключения пользователя `postgres` по адресу `192.168.0.2` через порт `2302`, с использованием каталога резервных копий в системе, настроенной для подключения пользователя `backup` по адресу `192.168.0.3` через порт `2303`, выполните:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра --remote-user=postgres
 --remote-host=192.168.0.2 --remote-port=2302 --archive-host=192.168.0.3 --archive-
port=2303 --archive-user=backup
```

Переданные аргументы будут использоваться при составлении команды `restore_command`:

```
restore_command = '"путь_инсталляции/pg_probackup" archive-get -B "каталог_копий" --
instance=имя_экземпляра --wal-file-path=%p --wal-file-name=%f --remote-host=192.168.0.3
 --remote-port=2303 --remote-user=backup'
```

Также вы можете воспользоваться ключом `--restore-command` и передать всю команду *`restore_command`* целиком:

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра --remote-user=postgres
 --remote-host=192.168.0.2 --remote-port=2302 --restore-command='"путь_инсталляции/
pg_probackup" archive-get -B "каталог_копий" --instance=имя_экземпляра --wal-file-
path=%p --wal-file-name=%f --remote-host=192.168.0.3 --remote-port=2303 --remote-
user=backup'
```

Примечание

Использование удалённого режима на платформе Windows в настоящее время не поддерживается.

Запуск pg_probackup в параллельных потоках

Команды backup, restore, merge, delete, checkdb и validate могут выполняться в несколько параллельных потоков. Это может существенно ускорять работу pg_probackup при наличии достаточных ресурсов (ядер процессора, производительности дисковой подсистемы и сети).

Параллельным выполнением управляет ключ командной строки `-j/--threads`. Например, чтобы запустить резервное копирование в четыре параллельных потока, выполните:

```
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b FULL -j 4
```

Примечание

Восстановление происходит в параллельном режиме только на этапе копирования данных из каталога копий в каталог данных кластера. При запуске сервера Postgres Pro он должен будет воспроизвести записи из WAL, а это может происходить только последовательно.

Важно

Для выполнения pg_probackup в параллельных потоках реализован механизм, исключающий копирование одного файла несколько раз. Но при его использовании в системах с медленными дисками или слишком высокой нагрузкой копирование может завершиться ошибкой. Чтобы исключить такую ситуацию, устраните проблемы своей рабочей среды.

Настройка pg_probackup

Проинициализировав каталог резервных копий и добавив определение копируемого экземпляра, вы можете использовать файл `pg_probackup.conf` в каталоге *`каталог_копий`*`/backups/`*`имя_экземпляра`* для тонкой настройки конфигурации pg_probackup.

Например, команды backup и checkdb работают через обычное подключение Postgres Pro. Чтобы каждый раз не задавать параметры подключения в командной строке, вы можете определить их в файле конфигурации `pg_probackup.conf` с помощью команды set-config.

> **Примечание**
>
> Редактировать `pg_probackup.conf` вручную **не рекомендуется**.

Изначально `pg_probackup.conf` содержит следующие параметры:

- `PGDATA` — путь к каталогу данных кластера, который будет копироваться.
- `system-identifier` — уникальный идентификатор экземпляра Postgres Pro.

Вы можете дополнительно установить параметры удалённого режима, хранения копий, ведения журнала и сжатия, используя команду `set-config`:

```
pg_probackup set-config -B каталог_копий --instance=имя_экземпляра
[--external-dirs=путь_внешнего_каталога] [параметры_удалённого_режима]
 [параметры_соединения] [параметры_хранения] [параметры_журнала]
```

Чтобы просмотреть текущие параметры, выполните эту команду:

```
pg_probackup show-config -B каталог_копий --instance=имя_экземпляра
```

Параметры, заданные в `pg_probackup.conf`, можно переопределить, задавая соответствующие переменные окружения или параметры командной строки при вызове команд pg_probackup.

Указание параметров подключения

Если вы определите свойства соединения в файле конфигурации `pg_probackup.conf`, вы можете не указывать параметры соединения во всех последующих командах pg_probackup. Однако если установлены соответствующие переменные окружения, они имеют больший приоритет. Параметры, заданные в командной строке, переопределяют как переменные окружения, так и свойства в файле конфигурации.

Если не задано ничего, используются значения по умолчанию. В частности, pg_probackup пытается подключиться к Unix-сокету локального сервера (или к `localhost` в Windows), а в качестве имени базы данных и имени пользователя выбирает значение переменной окружения `PGUSER` либо имя текущего пользователя ОС.

Управление каталогом резервных копий

С помощью pg_probackup вы можете управлять резервными копиями в командной строке:

- Просматривать имеющиеся резервные копии
- Просматривать оглавление архива WAL
- Проверять резервные копии
- Объединять резервные копии
- Удалять резервные копии

Просмотр информации о резервных копиях

Чтобы просмотреть список существующих копий для каждого экземпляра, выполните команду:

```
pg_probackup show -B каталог_копий
```

pg_probackup выводит список всех имеющихся резервных копий. Например:

```
BACKUP INSTANCE 'node'
============================================================================================
 Instance  Version  ID      Recovery Time                  Mode    WAL Mode  TLI  Time
  Data   WAL  Zalg  Zratio  Start LSN   Stop LSN    Status
============================================================================================
```

```
 node      17       SBOL94  2024-04-09 18:19:56.603355+03  FULL    ARCHIVE   1/0    6s
 377MB  16MB   lz4    1.46  0/41000028  0/420000C0  OK
 node      17       SBOL8S  2024-04-09 18:19:43.707720+03  DELTA   STREAM    1/1    3s
 114MB  64MB   lz4    1.42  0/3C003020  0/3E8D4930  OK
 node      17       SBOL8G  2024-04-09 18:19:32.594670+03  PTRACK  STREAM    1/1    4s
  30MB  16MB  zlib    2.23  0/31000028  0/310029E0  OK
 node      17       SBOL83  2024-04-09 18:19:22.269595+03  PAGE    STREAM    1/1    7s
  46MB  32MB  pglz    1.44  0/29000028  0/2A0000F8  OK
 node      17       SBOL7P  2024-04-09 18:19:06.557301+03  FULL    STREAM    1/0    6s
 144MB  16MB  zstd    2.47  0/22000028  0/220001C8  OK
```

Для каждой копии выдаются следующие сведения:

- `Instance` — имя экземпляра.
- `Version` — базовая версия Postgres Pro.
- `ID` — идентификатор резервной копии.
- `Recovery time` — самое ранее время, на которое можно восстановить кластер из данной копии.
- `Mode` — режим, в котором была сделана копия. Возможные значения: `FULL` (полная), `PAGE` (страничная), `DELTA` (инкрементальная), `PTRACK` (копирование изменений).
- `WAL Mode` — режим доставки WAL. Возможные значения: `STREAM` (потоковый) и `ARCHIVE` (архивный).
- `TLI` — идентификаторы линии времени текущей копии и её родителя.
- `Time` — время, за которое была выполнена данная копия.
- `Data` — объём файлов данных в этой копии. Это значение не включает в себя объём файлов WAL. Для копий, сделанных в режиме STREAM, общий размер можно рассчитать, сложив значения `Data` и `WAL`.
- `WAL` — размер несжатых файлов WAL, которые должны быть применены в процессе восстановления копии для достижения согласованного состояния.
- `compress-alg` — алгоритм сжатия, используемый при получении резервной копии. Возможные значения: `zlib`, `pglz`, `lz4`, `zstd` и `none` (сжатие не производилось).
- `Zratio` — коэффициент сжатия, вычисленный как отношение «uncompressed-bytes» (объём несжатых данных в байтах) к «data-bytes» (итоговый объём данных).
- `Start LSN` — последовательный номер в журнале WAL, соответствующий началу процесса копирования. С этой позиции накатываются изменения (REDO) в процессе восстановления Postgres Pro.
- `Stop LSN` — последовательный номер в журнале WAL, соответствующий окончанию процесса копирования. Это позиция точки согласованности при восстановлении Postgres Pro.
- `Status` — состояние резервной копии. Возможные варианты:
  - `OK` — резервная копия сделана и пригодна к использованию.
  - `DONE` — резервная копия сделана, но не проверена.
  - `RUNNING` — резервное копирование выполняется.
  - `MERGING` — резервная копия объединяется.
  - `MERGED` — файлы резервной копии были успешно обработаны в процессе объединения копий, но её метаданные ещё изменяются. Это состояние могут иметь только полные резервные копии.
  - `DELETING` — файлы резервной копии удаляются.
  - `CORRUPT` — некоторые файлы резервной копии повреждены.
  - `ERROR` — резервное копирование было прервано из-за неожиданной ошибки.

  - `ORPHAN` — резервная копия непригодна к использованию, так как её родительская копия испорчена или отсутствует.
  - `HIDDEN_FOR_TEST` — скрипт теста пометил копию как несуществующую. (Собственно pg_probackup никогда не устанавливает это состояние.)

Восстановить кластер из копии можно только для копий с состоянием `OK` или `DONE`.

Чтобы получить более подробную информацию о копии, укажите в команде `show` её идентификатор:

```
pg_probackup show -B каталог_копий --instance=имя_экземпляра -i ид_резервной_копии
```

Пример вывода:

```
#Configuration
backup-mode = FULL
stream = false
compress-alg = lz4
compress-level = 1
from-replica = false

#Compatibility
block-size = 8192
xlog-block-size = 8192
checksum-version = 1
program-version = 2.7.3
server-version = 17

#Result backup info
timelineid = 1
start-lsn = 0/41000028
stop-lsn = 0/420000C0
start-time = '2024-04-09 18:19:52+03'
end-time = '2024-04-09 18:19:58+03'
end-validation-time = '2024-04-09 18:19:59+03'
recovery-xid = 757
recovery-time = '2024-04-09 18:19:56.603355+03'
data-bytes = 395651278
wal-bytes = 16777216
uncompressed-bytes = 578552566
pgdata-bytes = 578552248
status = OK
primary_conninfo = 'user=backup channel_binding=prefer host=localhost
 port=5432 sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres gssdelegation=0
 target_session_attrs=any load_balance_hosts=disable'
content-crc = 3862224379
```

В расширенном выводе представлены дополнительные атрибуты:

- `compress-alg` — алгоритм сжатия, используемый при получении резервной копии. Возможные значения: `zlib`, `pglz`, `lz4`, `zstd` и `none` (сжатие не производилось).
- `compress-level` — уровень сжатия, применяемый в процессе резервного копирования.
- `from-replica` — признак того, что копия была сделана с ведомого сервера. Возможные значения: `1`, `0`.
- `block-size` — значение параметра block_size, установленное в кластере Postgres Pro, с которого была сделана копия.
- `checksum-version` — признак включения параметра data_checksums в исходном кластере Postgres Pro. Возможные значения: `1`, `0`.

- `program-version` — полная версия программы pg_probackup, которая создала эту копию.
- `start-time` — время начала резервного копирования.
- `end-time` — время окончания резервного копирования.
- `end-validation-time` — время окончания проверки резервной копии.
- `expire-time` — время, когда закреплённая копия может быть ликвидирована в соответствии с политикой хранения. Этот атрибут имеется только у закреплённых копий.
- `uncompressed-bytes` — размер файлов данных до добавления заголовков страниц и сжатия. В случае использования сжатия оценить его эффективность можно, сопоставив объём `uncompressed-bytes` (байтов несжатых данных) с `data-bytes` (байтов данных).
- `pgdata-bytes` — размер файлов данных Postgres Pro на момент копирования. Эффективность инкрементального метода копирования можно оценить, сопоставив объём `pgdata-bytes` (байтов в PGDATA) с `uncompressed-bytes` (байтов несжатых данных).
- `recovery-xid` — идентификатор транзакции, соответствующей моменту окончания резервного копирования.
- `parent-backup-id` — идентификатор родительской копии. Определён только для инкрементальных копий.
- `primary_conninfo` — параметры подключения libpq, с использованием которых производилось подключение к кластеру Postgres Pro для получения этой резервной копии. Пароль в эти параметры не включается.
- `note` — текстовое примечание, связанное с копией.
- `content-crc` — контрольная сумма файла `backup_content.control`, рассчитанная по алгоритму CRC32. Она позволяет выявить повреждение метаданных копии.

Вы также можете получить подробную информацию о резервной копии в формате JSON:

```
pg_probackup show -B каталог_копий --instance=имя_экземпляра --format=json -i backup_id
```

Пример вывода:

```
[
    {
        "instance": "node",
        "backups": [
            {
                "id": "SBOL94",
                "status": "OK",
                "start-time": "2024-04-09 18:19:52+03",
                "backup-mode": "FULL",
                "wal": "ARCHIVE",
                "compress-alg": "lz4",
                "compress-level": 1,
                "from-replica": "false",
                "block-size": 8192,
                "xlog-block-size": 8192,
                "checksum-version": 1,
                "program-version": "2.7.3",
                "server-version": "17",
                "current-tli": 16,
                "parent-tli": 2,
                "start-lsn": "0/41000028",
                "stop-lsn": "0/420000C0",
                "end-time": "2024-04-09 18:19:58+03",
                "end-validation-time": "2024-04-09 18:19:59+03",
                "recovery-xid": 757,
                "recovery-time": "2024-04-09 18:19:56.603355+03",
                "data-bytes": 395651278,
                "wal-bytes": 16777216,
```

```
                "uncompressed-bytes": 578552566,
                "pgdata-bytes": 578552248,
                "primary_conninfo": "user=backup channel_binding=prefer
 host=localhost port=5432 sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres gssdelegation=0
 target_session_attrs=any load_balance_hosts=disable",
                "content-crc": 3862224379
            }
        ]
    }
]
```

Просмотр оглавления архива WAL

Чтобы получить информацию об архиве WAL для каждого экземпляра, выполните команду:

```
pg_probackup show -B каталог_копий [--instance=имя_экземпляра] --archive
```

pg_probackup выводит список всех имеющихся файлов WAL, сгруппированных по линиям времени. Например:

```
ARCHIVE INSTANCE 'node'
============================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size   Zratio  N backups  Status
============================================================================================
 1    0           0/0          00000001000000000000001B  000000010000000000000042  40
        640MB  1.00    5          OK
```

Для каждой линии времени выдаются следующие сведения:

- `TLI` — идентификатор линии времени.
- `Parent TLI` — идентификатор линии времени, от которой была ответвлена данная.
- `Switchpoint` — LSN момента, когда эта линия времени ответвилась от родительской.
- `Min Segno` — первый сегмент WAL, относящийся к этой линии времени.
- `Max Segno` — последний сегмент WAL, относящийся к этой линии времени.
- `N segments` — количество сегментов WAL, относящихся к этой линии времени.
- `Size` — объём, который занимают файлы на диске.
- `Zalg` — алгоритм сжатия, используемый при получении резервной копии. Возможные значения: `zlib`, `pglz`, `lz4`, `zstd` и `none` (сжатие не производилось).
- `Zratio` — коэффициент сжатия, вычисляемый по формуле `N segments` * *wal_segment_size* * *wal_block_size* / `Size`.
- `N backups` — число копий, относящихся к этой линии времени. Для получения подробных сведений об этих копиях воспользуйтесь форматом JSON.
- `Status` — состояние архива WAL для этой линии времени. Возможные значения:
  - `OK` — в архиве присутствуют все сегменты WAL между `Min Segno` и `Max Segno`.
  - `DEGRADED` — отсутствуют некоторые сегменты WAL между `Min Segno` и `Max Segno`. Понять, какие файлы утрачены, можно, посмотрев этот отчёт в формате JSON. Такой статус может возникать, если несколько файлов WAL (в середине последовательности) были удалены командой delete с параметром `--delete-wal` в соответствии с политикой хранения. Данный статус не влияет на корректность восстановления, но восстановить кластер до некоторых точек восстановления может быть невозможно.

Чтобы получить более подробную информацию об архиве WAL в формате JSON, выполните команду:

```
pg_probackup show -B каталог_копий [--instance=имя_экземпляра] --archive --format=json
```

Пример вывода:

```
[
    {
        "instance": "node",
        "timelines": [
            {
                "tli": 1,
                "parent-tli": 0,
                "switchpoint": "0/0",
                "min-segno": "00000001000000000000001B",
                "max-segno": "000000010000000000000042",
                "n-segments": 40,
                "size": 671088640,
                "zratio": 1.00,
                "closest-backup-id": "",
                "status": "OK",
                "lost-segments": [],
                "backups": [
                    {
                        "id": "SBOL94",
                        "status": "OK",
                        "start-time": "2024-04-09 18:19:52+03",
                        "backup-mode": "FULL",
                        "wal": "ARCHIVE",
                        "compress-alg": "lz4",
                        "compress-level": 1,
                        "from-replica": "false",
                        "block-size": 8192,
                        "xlog-block-size": 8192,
                        "checksum-version": 1,
                        "program-version": "2.7.3",
                        "server-version": "17",
                        "current-tli": 2,
                        "parent-tli": 0,
                        "start-lsn": "0/41000028",
                        "stop-lsn": "0/420000C0",
                        "end-time": "2024-04-09 18:19:58+03",
                        "end-validation-time": "2024-04-09 18:19:59+03",
                        "recovery-xid": 757,
                        "recovery-time": "2024-04-09 18:19:56.603355+03",
                        "data-bytes": 395651278,
                        "wal-bytes": 16777216,
                        "uncompressed-bytes": 578552566,
                        "pgdata-bytes": 578552248,
                        "primary_conninfo": "user=backup channel_binding=prefer
 host=localhost port=5432 sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
 ssl_min_protocol_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres gssdelegation=0
 target_session_attrs=any load_balance_hosts=disable",
                        "content-crc": 3862224379
                    },
                    {
                        "id": "SBOL8S",
                        "status": "OK",
                        "start-time": "2024-04-09 18:19:40+03",
                        "parent-backup-id": "SBOL8G",
                        "backup-mode": "DELTA",
                        "wal": "STREAM",
```

```
                       "compress-alg": "lz4",
                       "compress-level": 1,
                       "from-replica": "false",
                       "block-size": 8192,
                       "xlog-block-size": 8192,
                       "checksum-version": 1,
                       "program-version": "2.7.3",
                       "server-version": "17",
                       "current-tli": 1,
                       "parent-tli": 1,
                       "start-lsn": "0/3C003020",
                       "stop-lsn": "0/3E8D4930",
                       "end-time": "2024-04-09 18:19:43+03",
                       "end-validation-time": "2024-04-09 18:19:44+03",
                       "recovery-xid": 757,
                       "recovery-time": "2024-04-09 18:19:43.707720+03",
                       "data-bytes": 119350434,
                       "wal-bytes": 67108864,
                       "uncompressed-bytes": 170044286,
                       "pgdata-bytes": 578552248,
                       "primary_conninfo": "user=backup channel_binding=prefer
host=localhost port=5432 sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
ssl_min_protocol_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres gssdelegation=0
target_session_attrs=any load_balance_hosts=disable",
                       "content-crc": 1259851036
                   },
                   {
                       "id": "SBOL8G",
                       "status": "OK",
                       "start-time": "2024-04-09 18:19:28+03",
                       "parent-backup-id": "SBOL83",
                       "backup-mode": "PTRACK",
                       "wal": "STREAM",
                       "compress-alg": "zlib",
                       "compress-level": 1,
                       "from-replica": "false",
                       "block-size": 8192,
                       "xlog-block-size": 8192,
                       "checksum-version": 1,
                       "program-version": "2.7.3",
                       "server-version": "17",
                       "current-tli": 1,
                       "parent-tli": 1,
                       "start-lsn": "0/31000028",
                       "stop-lsn": "0/310029E0",
                       "end-time": "2024-04-09 18:19:32+03",
                       "end-validation-time": "2024-04-09 18:19:33+03",
                       "recovery-xid": 756,
                       "recovery-time": "2024-04-09 18:19:32.594670+03",
                       "data-bytes": 31218302,
                       "wal-bytes": 16777216,
                       "uncompressed-bytes": 69610366,
                       "pgdata-bytes": 510263736,
                       "primary_conninfo": "user=backup channel_binding=prefer
host=localhost port=5432 sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
ssl_min_protocol_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres gssdelegation=0
target_session_attrs=any load_balance_hosts=disable",
                       "content-crc": 2293248595
```

```
                    },
                    {
                        "id": "SBOL83",
                        "status": "OK",
                        "start-time": "2024-04-09 18:19:15+03",
                        "parent-backup-id": "SBOL7P",
                        "backup-mode": "PAGE",
                        "wal": "STREAM",
                        "compress-alg": "pglz",
                        "compress-level": 1,
                        "from-replica": "false",
                        "block-size": 8192,
                        "xlog-block-size": 8192,
                        "checksum-version": 1,
                        "program-version": "2.7.3",
                        "server-version": "17",
                        "current-tli": 1,
                        "parent-tli": 1,
                        "start-lsn": "0/29000028",
                        "stop-lsn": "0/2A0000F8",
                        "end-time": "2024-04-09 18:19:22+03",
                        "end-validation-time": "2024-04-09 18:19:22+03",
                        "recovery-xid": 755,
                        "recovery-time": "2024-04-09 18:19:22.269595+03",
                        "data-bytes": 48394744,
                        "wal-bytes": 33554432,
                        "uncompressed-bytes": 69577598,
                        "pgdata-bytes": 441975224,
                        "primary_conninfo": "user=backup channel_binding=prefer
host=localhost port=5432 sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
ssl_min_protocol_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres gssdelegation=0
target_session_attrs=any load_balance_hosts=disable",
                        "content-crc": 343227200
                    },
                    {
                        "id": "SBOL7P",
                        "status": "OK",
                        "start-time": "2024-04-09 18:19:01+03",
                        "backup-mode": "FULL",
                        "wal": "STREAM",
                        "compress-alg": "zstd",
                        "compress-level": 1,
                        "from-replica": "false",
                        "block-size": 8192,
                        "xlog-block-size": 8192,
                        "checksum-version": 1,
                        "program-version": "2.7.3",
                        "server-version": "17",
                        "current-tli": 1,
                        "parent-tli": 0,
                        "start-lsn": "0/22000028",
                        "stop-lsn": "0/220001C8",
                        "end-time": "2024-04-09 18:19:07+03",
                        "end-validation-time": "2024-04-09 18:19:09+03",
                        "recovery-xid": 754,
                        "recovery-time": "2024-04-09 18:19:06.557301+03",
                        "data-bytes": 151177272,
                        "wal-bytes": 16777216,
```

```
                        "uncompressed-bytes": 373695222,
                        "pgdata-bytes": 373694904,
                        "primary_conninfo": "user=backup channel_binding=prefer
host=localhost port=5432 sslmode=prefer sslcompression=0 sslcertmode=allow sslsni=1
ssl_min_protocol_version=TLSv1.2 gssencmode=prefer krbsrvname=postgres gssdelegation=0
target_session_attrs=any load_balance_hosts=disable",
                        "content-crc": 1636300818
                    }
                ]
            }
        ]
    }
]
```

В основном в этом формате представлены те же поля, что и в текстовом формате, с некоторыми исключениями:

- Размер выражается в байтах.
- Атрибут `closest-backup-id` содержит идентификатор самой последней доступной копии, принадлежащей к одной из предыдущих линий времени. Вы можете использовать эту копию для восстановления на момент, относящийся к этой линии времени. Если такой копии не существует, данный атрибут будет пустым.
- В массиве `lost-segments` представлены интервалы отсутствующих сегментов на линиях времени в непригодном состоянии (`DEGRADED`). На линиях времени в целостном состоянии `OK` массив `lost-segments` пуст.
- В массиве `backups` перечисляются все резервные копии, относящиеся к данной линии времени. Если к линии времени не относятся резервные копии, этот массив пуст.

Настройка политики хранения

Используя pg_probackup, вы можете реализовать политики хранения резервных копий, в соответствии с которыми могут удаляться лишние копии, очищаться ненужные сегменты WAL. Кроме того, можно закреплять определённые копии, чтобы они сохранялись независимо от политики, как описано ниже. Эти подходы можно комбинировать произвольным образом.

Удаление ненужных копий

По умолчанию все резервные копии, которые создаёт pg_probackup, сохраняются в предназначенном для них каталоге. Для экономии дискового пространства вы можете настроить политику сохранения копий, чтобы ненужные копии удалялись.

Чтобы настроить политику хранения, задайте одну или несколько следующих переменных в файле `pg_probackup.conf` с помощью команды set-config:

`--retention-redundancy=`*`избыточность`*

Определяет **количество полных резервных копий**, которое должно сохраняться в каталоге копий.

`--retention-window=`*`окно`*

Определяет самый ранний момент времени, на который pg_probackup может выполнить восстановление. В этом параметре задаётся **количество дней** от текущего момента. Например, если `retention-window=7`, должна сохраниться минимум одна копия старее 7 дней, вместе с соответствующими файлами WAL и всеми последующими копиями.

Если установлены параметры `--retention-redundancy` и `--retention-window`, при очистке каталога от ненужных копий принимаются во внимание оба заданных ими условия. Например, если задать параметры `--retention-redundancy=2` и `--retention-window=7`, программа pg_probackup должна сохранить две полные резервные копии, а также все копии, необходимые для восстановления данных, за последние семь дней:

```
pg_probackup set-config -B каталог_копий --instance=имя_экземпляра --retention-
redundancy=2 --retention-window=7
```

Рекомендуется всегда хранить как минимум две последние полные родительские копии, чтобы избежать ошибок при создании инкрементальных резервных копий.

Чтобы очистить каталог копий в соответствии с политикой хранения, нужно запустить команду delete с флагами сохранения, как показано ниже, или выполнить с этими флагами команду backup. В последнем случае ненужные копии будут удалены сразу после создания новой копии.

Например, чтобы удалить все резервные копии, считающиеся ненужными согласно установленной политике хранения, нужно выполнить следующую команду с флагом `--delete-expired`:

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра --delete-expired
```

Если вы хотите также удалить файлы WAL, которые больше не требуются ни для каких копий, укажите дополнительно `--delete-wal`:

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра --delete-expired --
delete-wal
```

Вы также можете установить или переопределить текущую политику хранения, добавив параметры `--retention-redundancy` и `--retention-window` непосредственно при выполнении команд `delete` или `backup`:

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра --delete-expired --
retention-window=7 --retention-redundancy=2
```

Так как для инкрементальных копий требуется наличие всех родительских полных копий и всех предыдущих инкрементальных копий, даже по истечении срока их хранения эти копии нельзя удалить, пока минимум одна инкрементальная копия в цепочке удовлетворяет политике хранения. Чтобы не хранить устаревшие копии, которые всё ещё нужны для восстановления нужной инкрементальной копии, вы можете объединить их с ней, воспользовавшись ключом `--merge-expired` команды backup или delete.

Предположим, что вы заархивировали экземпляр *узла* в *каталог_копий* со значением параметра `--retention-window`, равным `6`, и значением параметра `--retention-redundancy`, равным `2`, и на 1 августа 2024 г. у вас есть следующие копии:

```
BACKUP INSTANCE 'node'
============================================================================================
 Instance  Version  ID      Recovery Time          Mode   WAL Mode  TLI  Time    Data
  WAL  Zalg  Zratio  Start LSN   Stop LSN    Status
============================================================================================
 node      17       SHJ1N9  2024-08-01 09:50:00+03  FULL   ARCHIVE   1/0    5s    13MB
 16MB  zstd    2,81  0/1D000028  0/1E0000C0  OK
 node      17       SHJ1N8  2024-08-01 09:49:59+03  DELTA  ARCHIVE   1/1    5s  6432kB
 16MB  zstd    1,06  0/1A000028  0/1B0000C0  OK
 node      17       SHH6Z8  2024-07-31 09:49:59+03  PAGE   ARCHIVE   1/1    5s  6431kB
 16MB  zstd    1,06  0/17000028  0/180000C0  OK
 node      17       SHFCB6  2024-07-30 09:49:57+03  FULL   ARCHIVE   1/0    5s    12MB
 16MB  zstd    2,83  0/14000028  0/150000C0  OK
 node      17       SH9SB5  2024-07-27 09:49:56+03  PAGE   ARCHIVE   1/1    5s  6432kB
 16MB  zstd    1,06  0/11000028  0/120000C0  OK
 ----------------------------------------------------------retention
 window-----------------------------------------------------------
 node      17       SH62Z5  2024-07-25 09:49:56+03  DELTA  ARCHIVE   1/1    5s  6431kB
 16MB  zstd    1,06  0/E000028   0/F0000C0   OK
 node      17       SH48B3  2024-07-24 09:49:54+03  FULL   ARCHIVE   1/0    5s    12MB
 16MB  zstd    2,86  0/B000028   0/C0000C0   OK
 node      17       SGWTN3  2024-07-20 09:49:54+03  PAGE   ARCHIVE   1/1    5s  6432kB
 16MB  zstd    1,06  0/8000028   0/90000C0   OK
```

```
 node      17       SGUYZ2  2024-07-19 09:49:53+03  DELTA  ARCHIVE   1/1    5s  6442kB
 16MB  zstd    1,07  0/5000028   0/60000C0   OK
 node      17       SGT4B1  2024-07-18 09:49:52+03  FULL   ARCHIVE   1/0    5s    11MB
 16MB  zstd    2,89  0/2000028   0/3003A28   OK
```

При запуске команды delete с флагом `--delete-expired` резервные копии с идентификаторами `SGT4B1`, `SGUYZ2` и `SGWTN3` будут удалены как устаревшие по критериям окна хранения и избыточности (требуемый набор полных резервных копий уже сохранён). `SGUYZ2` и `SGWTN3` также будут удалены, поскольку базовая полная копия потеряла актуальность.

При запуске команды delete с флагом `--merge-expired` резервные копии `SH48B3` и `SH62Z5` будут объединены с `SH9SB5`. Объединение произойдёт именно с `SH9SB5`, так как это первая актуальная разностная копия, которую можно объединить с неактуальной разностной копией `SH62Z5` и неактуальной полной копией `SH48B3`.

```
pg_probackup delete -B каталог_копий --instance=узел --delete-expired --merge-expired
pg_probackup show -B каталог_копий
```

```
BACKUP INSTANCE 'node'
===========================================================================================
 Instance  Version  ID      Recovery Time           Mode   WAL Mode  TLI  Time    Data
  WAL  Zalg  Zratio  Start LSN   Stop LSN    Status
===========================================================================================
 node      17       SHJ1N9  2024-08-01 09:50:00+03  FULL   ARCHIVE   1/0    5s    13MB
 16MB  zstd    2,81  0/1D000028  0/1E0000C0  OK
 node      17       SHJ1N8  2024-08-01 09:49:59+03  DELTA  ARCHIVE   1/1    5s  6432kB
 16MB  zstd    1,06  0/1A000028  0/1B0000C0  OK
 node      17       SHH6Z8  2024-07-31 09:49:59+03  PAGE   ARCHIVE   1/1    5s  6431kB
 16MB  zstd    1,06  0/17000028  0/180000C0  OK
 node      17       SHFCB6  2024-07-30 09:49:57+03  FULL   ARCHIVE   1/0    5s    12MB
 16MB  zstd    2,83  0/14000028  0/150000C0  OK
 node      17       SH9SB5  2024-07-27 09:49:56+03  FULL   ARCHIVE   1/0    1s    12MB
 16MB  zstd    2,84  0/11000028  0/120000C0  OK
```

Поле `Time` (Время) для объединённой копии показывает время, которое заняла процедура объединения.

Закрепление резервных копий

Если вам нужно хранить отдельные копии дольше, чем допускает установленная политика хранения, вы можете дополнительно закрепить их на определённое время. Например:

```
pg_probackup set-backup -B каталог_копий --instance=имя_экземпляра -
i ид_резервной_копии --ttl=30d
```

Эта команда задаёт срок хранения заданной резервной копии 30 дней, начиная с момента, указанного в атрибуте `recovery-time` этой копии.

Вы также можете явно задать время истечения срока хранения копии, воспользовавшись ключом `--expire-time`. Например:

```
pg_probackup set-backup -B каталог_копий --instance=имя_экземпляра -
i ид_резервной_копии --expire-time="2027-04-09 18:21:32+00"
```

Также можно воспользоваться ключами `--ttl` и `--expire-time` команды backup и сразу закрепить создаваемую копию:

```
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b FULL --ttl=30d
pg_probackup backup -B каталог_копий --instance=имя_экземпляра -b FULL --expire-
time="2027-04-09 18:21:32+00"
```

Определить, закреплена ли копия, можно с помощью команды show:

```
pg_probackup show -B каталог_копий --instance=имя_экземпляра -i ид_резервной_копии
```

Если копия закреплена, у неё есть атрибут `expire-time`, содержащий время окончания срока её хранения:

```
...
recovery-time = '2024-04-09 18:21:32+00'
expire-time = '2027-04-09 18:21:32+00'
data-bytes = 22288792
...
```

Отменить закрепление копии можно, передав в параметре `--ttl` ноль:

```
pg_probackup set-backup -B каталог_копий --instance=имя_экземпляра -
i ид_резервной_копии --ttl=0
```

> **Примечание**
>
> Для закреплённой инкрементальной копии неявным образом закрепляются все нужные ей родительские копии. Если вы позже снимите с неё закрепление, последние также автоматически перестанут быть закреплёнными.

Настройка политики хранения архива WAL

Когда осуществляется непрерывное архивирование WAL, сегменты WAL могут занимать много места на диске. Даже если вы время от времени удаляете старые резервные копии, с флагом `--delete-wal` могут быть удалены только те сегменты WAL, которые не относятся ни к какой копии из остающихся в каталоге. Однако если возможность восстановления на момент времени нужна только для последних копий, можно настроить политику хранения архива WAL, чтобы ограничить глубину архива и сэкономить место на диске.

Чтобы настроить политику хранения архива WAL, запустите команду set-config с параметром `--wal-depth`, задающим количество копий, которые могут быть использованы для PITR. Этот параметр действует для всех линий времени, поэтому вы сможете выполнять PITR для одинакового количества копий на каждой линии времени, при их наличии. Закреплённые копии в этом числе не учитываются: если закрепляется одна из последних копий, pg_probackup обеспечивает возможность PITR для каждой дополнительной копии.

Чтобы удалить сегменты WAL, не удовлетворяющие заданной политике хранения архива WAL, вы должны просто выполнить команду delete или backup с флагом `--delete-wal`. Для архивных копий сегменты WAL между `Start LSN` и `Stop LSN` всегда сохраняются, так что эти копии остаются рабочими вне зависимости от значения `--wal-depth` и при необходимости могут быть восстановлены.

Вы также можете использовать параметр `--wal-depth` с командами delete и backup для переопределения ранее заданной политики сохранения WAL и удаления старых сегментов WAL «на лету».

Предположим, что вы заархивировали экземпляр `node` в *`каталог_копий`* и настроили непрерывное архивирование WAL:

```
pg_probackup show -B каталог_копий --instance=узел

============================================================================================
 Instance  Version  ID      Recovery Time                  Mode   WAL Mode  TLI  Time
  Data   WAL  Zalg  Zratio  Start LSN   Stop LSN    Status
============================================================================================
 node      17       SBOLDA  2024-04-09 18:22:23.147138+03  DELTA  STREAM    1/1    1s
1165kB   16MB  zstd    1.09  0/6F000028  0/6F000190  OK
 node      17       SBOLCY  2024-04-09 18:22:16.079841+03  FULL   STREAM    1/0   10s
 278MB  16MB  zstd    2.46  0/6D000028  0/6D000190  OK
```

```
 node      17       SBOLCW  2024-04-09 18:22:10.154022+03  DELTA  STREAM    1/1    2s
 1364kB  16MB  zstd    1.01  0/6B000028  0/6B000190  OK
 node      17       SBOLCS  2024-04-09 18:22:07.521646+03  DELTA  STREAM    1/1    4s
  78MB  16MB  zstd    2.41  0/69000028  0/69000190  OK
 node      17       SBOLCC  2024-04-09 18:21:55.830115+03  FULL   STREAM    1/0   15s
 278MB  96MB  zstd    2.46  0/600060C8  0/64FE6640  OK
 node      17       SBOLBW  2024-04-09 18:21:38.399702+03  FULL   STREAM    1/0   12s
 278MB  96MB  zstd    2.46  0/54001830  0/589E5908  OK
```

Вы можете проверить состояние архива WAL, запустив команду show с ключом `--archive`:

```
pg_probackup show -B каталог_копий --instance=node --archive
```

```
ARCHIVE INSTANCE 'node'
============================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size   Zratio  N backups  Status
============================================================================================
 1    0           0/0          000000010000000000000052  00000001000000000000006F  30
        480MB  1.00    6          OK
```

Операция очистки WAL без ключа `--wal-depth` может удалить лишь один сегмент:

```
pg_probackup delete -B каталог_копий --instance=node --delete-wal
```

```
ARCHIVE INSTANCE 'node'
============================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size   Zratio  N backups  Status
============================================================================================
 1    0           0/0          000000010000000000000054  00000001000000000000006F  28
        448MB  1.00    6          OK
```

Например, если вы хотите оставить только те сегменты, которые могут применяться к самой последней копии, передайте в параметре `--wal-depth` значение 1:

```
pg_probackup delete -B каталог_копий --instance=node --delete-wal --wal-depth=1
```

```
ARCHIVE INSTANCE 'node'
============================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size  Zratio  N backups  Status
============================================================================================
 1    0           0/0          000000010000000000000002  000000010000000000000047  70
        34MB  32.00   6          OK
```

Также вы можете использовать параметр `--wal-depth` с командой backup:

```
pg_probackup backup -B каталог_данных --instance=node -b DELTA --wal-depth=1 --delete-
wal
```

```
ARCHIVE INSTANCE 'node'
============================================================================================
 TLI  Parent TLI  Switchpoint  Min Segno                 Max Segno                 N
 segments  Size  Zratio  N backups  Status
============================================================================================
 1    0           0/0          000000010000000000000071  000000010000000000000071  1
        16MB  1.00    7          OK
```

Объединение резервных копий

По мере того как вы будете делать новые и новые инкрементальные копии, общий размер каталога резервных копий может существенно увеличиться. Для экономии места на диске вы можете объединить инкрементальные копии с родительской полной копией, выполнив команду `merge` и передав ей идентификатор копии самой последней резервной копии, подлежащей объединению:

```
pg_probackup merge -B каталог_копий --instance=имя_экземпляра -i ид_резервной_копии
```

Эта команда объединяет копии, относящиеся к одной цепочке инкрементальных копий. Если выбирается полная копия, она будет объединена с первой инкрементальной копией после неё. Если выбрана инкрементальная копия, он будет объединена с родительской полной копией, включая все инкрементальные копии между ними. После выполнения команды полная копия содержит все объединённые данные, а инкрементальные копии удаляются как ненужные. Таким образом, операция объединения по сути равнозначна созданию новой полной копии с удалением всех устаревших копий, но выполняется она быстрее, особенно с большими объёмами данных, и не нагружает подсистему ввода-вывода и сеть (если pg_probackup работает в удалённом режиме).

Перед объединением pg_probackup проверяет все задействуемые резервные копии, чтобы удостовериться в их целостности. Вы можете проверить текущее состояние резервной копии, передав её идентификатор команде show.

```
pg_probackup show -B каталог_копий --instance=имя_экземпляра -i ид_резервной_копии
```

Если процесс объединения ещё не закончен, вы увидите состояние `MERGING`. Для полных копий также можно увидеть состояние `MERGED` в процессе изменения метаданных на последнем этапе объединения. В случае прерывания операции объединения она может быть перезапущена.

Предупреждение

Не рекомендуется принудительно завершать операции объединения — это может нарушить дальнейшую работу команды `merge` и проверку резервных копий.

Удаление резервных копий

Для удаления резервной копии, ставшей ненужной, выполните команду:

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра -i ид_резервной_копии
```

Эта команда удалит резервную копию с заданным *`ид_резервной_копии`* вместе со всеми инкрементальными копиями, которые от неё зависят (если таковые найдутся). Таким образом вы можете удалить некоторые последние инкрементальные копии, сохранив предыдущую полную копию и некоторые следующие за ней инкрементальные копии.

Для удаления старых файлов WAL, которые не нужны для восстановления никаких из оставшихся резервных копий, воспользуйтесь ключом `--delete-wal`:

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра --delete-wal
```

Чтобы удалить резервные копии, считающиеся ненужными согласно текущей политике хранения, воспользуйтесь ключом `--delete-expired`:

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра --delete-expired
```

Копии с истёкшим сроком нельзя удалить, пока на них базируется минимум одна инкрементальная копия, удовлетворяющая политике хранения. Если вы хотите сократить число резервных копий, требующихся для восстановления инкрементальных копий, укажите параметр `--merge-expired` при выполнении этой команды:

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра --delete-expired --
merge-expired
```

В этом случае pg_probackup ищет самую старую инкрементальную копию, удовлетворяющую политике хранения и объединяет её с базовыми для неё полной и инкрементальными копиями, срок хранения которых истёк, тем самым превращая её в полную копию. После завершения объединения ставшие ненужными копии удаляются.

Прежде чем объединять или удалять резервные копии, вы можете выполнить команду `delete` с параметром `--dry-run` и получить в результате состояние всех имеющихся копий в соответствии с текущей политикой хранения; никакие необратимые действия при этом выполняться не будут.

Для удаления всех копий с определённым состоянием, воспользуйтесь параметром `--status`:

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра --status=ERROR
```

При удалении копий по критерию состояния установленные политики сохранения не учитываются.

Клонирование и синхронизация экземпляра Postgres Pro

В pg_probackup реализована команда catchup, которая позволяет создать копию экземпляра сервера Postgres Pro напрямую, не используя каталог резервных копий. Эта команда может быть полезна:

- Для добавления нового ведомого сервера.

  Как правило, для создания копии экземпляра Postgres Pro используется pg_basebackup. Команда `catchup`, если указан пустой целевой каталог, сделает то же самое, но в параллельном режиме может сработать гораздо быстрее.

- Для синхронизации отставшего ведомого сервера с ведущим.

  При активной записи на ведущем сервере реплики могут не успевать воспроизводить WAL с достаточной скоростью и в результате отставать. Обычно в этом случае создаётся новая реплика, а для этого нужно передать и сохранить на диске большой объём данных. Команда `catchup` напрямую переносит различия с ведущего сервера, что намного быстрее позволяет обновить данные на уже существующей реплике.

Операция `catchup` отличается от других операций pg_probackup:

- Не требуется каталог резервных копий.
- В качестве метода доставки WAL поддерживается только STREAM.
- Копирование внешних каталогов не поддерживается.
- Одновременно с `catchup` нельзя выполнять команды DDL CREATE TABLESPACE/DROP TABLESPACE.
- При синхронизации `catchup` берёт файлы конфигурации, такие как `postgresql.conf`, `postgresql.auto.conf` или `pg_hba.conf`, с исходного сервера и заменяет ими соответствующие файлы на целевом сервере. Чтобы оставить файлы конфигурации без изменений, используйте параметр `--exclude-path`.

Чтобы подготовить экземпляр Postgres Pro к клонированию/синхронизации, настройте исходный сервер БД следующим образом:

- Настройте кластер баз данных для копирования экземпляра сервера.
- Для копирования с удалённого сервера настройте удалённый режим.
- Для использования режима PTRACK настройте копирование PTRACK.

Перед клонированием/синхронизацией экземпляра Postgres Pro убедитесь, что исходный сервер запущен и принимает подключения. Чтобы клонировать/синхронизировать экземпляр Postgres Pro, в системе с целевым сервером выполните команду catchup:

```
pg_probackup catchup -b режим_синхронизации --source-
pgdata=путь_к_копируемому_каталогу_данных --destination-
```

```
pgdata=путь_к_целевому_каталогу_данных --stream [параметры_подключения]
 [параметры_удалённого_режима]
```

Здесь *режим_синхронизации* может принимать следующие значения:

- `FULL` — создаётся полная копия экземпляра Postgres Pro, для этого целевой каталог данных БД должен быть пустым.
- `DELTA` — считываются все файлы данных в каталоге данных и создаётся инкрементальная копия для страниц, изменённых с момента остановки целевого экземпляра.
- `PTRACK` — изменения в страницах отслеживаются на лету, считываются и копируются только страницы, изменённые с точки расхождения исходного и целевого экземпляров.

Предупреждение

Чтобы синхронизировать экземпляр в режиме PTRACK, необходим PTRACK версии 2.0 или выше, а значит, и Postgres Pro версии не ниже 11.

Указав параметр `--stream`, можно задать режим STREAM, при котором все необходимые файлы WAL передаются с исходного сервера по протоколу репликации.

Вы можете задать параметры_подключения к исходной БД, а если она находится на другом сервере, также укажите параметры_удалённого_режима.

Если в исходной БД есть табличные пространства, которые должны располагаться в других каталогах в целевой системе, задайте также параметр `--tablespace-mapping`:

```
pg_probackup catchup -b режим_синхронизации --source-
pgdata=путь_к_копируемому_каталогу_данных --destination-
pgdata=путь_к_целевому_каталогу_данных --stream --tablespace-
mapping=СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ
```

Чтобы операция `catchup` выполнялась в несколько параллельных потоков, задайте число потоков с помощью параметра `--threads`:

```
pg_probackup catchup -b режим_синхронизации --source-
pgdata=путь_к_копируемому_каталогу_данных --destination-
pgdata=путь_к_целевому_каталогу_данных --stream --threads=число_потоков
```

Перед клонированием/синхронизацией экземпляра Postgres Pro вы можете запустить команду `catchup` с флагом `--dry-run`, чтобы оценить размер передаваемых файлов данных без внесения изменений на диск:

```
pg_probackup catchup -b режим_синхронизации --source-
pgdata=путь_к_копируемому_каталогу_данных --destination-
pgdata=путь_к_целевому_каталогу_данных --stream --dry-run
```

Например, предположим, что отстал удалённый ведомый экземпляр Postgres Pro, данные которого хранятся в каталоге `/replica-pgdata`. Чтобы синхронизировать этот экземпляр с экземпляром в каталоге данных `/master-pgdata`, вы можете выполнить команду `catchup`, включив режим `PTRACK` и используя четыре параллельных потока, следующим образом:

```
pg_probackup catchup --source-pgdata=/master-pgdata --destination-pgdata=/replica-
pgdata -p 5432 -d postgres -U remote-postgres-user --stream --backup-mode=PTRACK
 --remote-host=remote-hostname --remote-user=remote-unix-username -j 4 --exclude-
path=postgresql.conf --exclude-path=postgresql.auto.conf --exclude-path=pg_hba.conf --
exclude-path=pg_ident.conf
```

Обратите внимание, что в этом примере файлы конфигурации не будут перезаписаны во время синхронизации.

В примере ниже показано, как можно добавить ещё один удалённый ведомый сервер Postgres Pro с каталогом данных `/replica-pga`, запустив `catchup` в режиме `FULL` в четыре параллельных потока:

```
pg_probackup catchup --source-pgdata=/master-pgdata --destination-pgdata=/replica-
pgdata -p 5432 -d postgres -U remote-postgres-user --stream --backup-mode=FULL --
remote-host=remote-hostname --remote-user=remote-unix-username -j 4
```

Справка по командной строке

Команды

В этом подразделе описываются команды pg_probackup. Необязательные параметры этих команд заключаются в квадратные скобки. В подробностях все параметры описываются в подразделе Параметры.

version

```
pg_probackup version [--format=json]
```

Выводит версию и редакцию pg_probackup, а также версию и редакцию сервера Postgres Pro.

При указании `--format=json` вывод будет получен в формате JSON. Это может потребоваться для внутренней интеграции с приложениями на основе JSON, например PPEM. Пример вывода JSON:

```
pg_probackup version --format=json
{
"pg_probackup":
{
"version": "2.8.2",
"edition": "community"
},
"database":
{
"type": "Postgres Pro Standard",
"version": "16.3.1"
},
"compressions": [zlib, pglz, lz4, zstd]
}
```

help

```
pg_probackup help [команда]
```

Выдаёт справку по командам pg_probackup. Если в параметрах задаётся одна из команд pg_probackup, выводит подробную информацию по параметрам, которые принимает эта команда.

init

```
pg_probackup init -B каталог_копий [--skip-if-exists] [--help]
[параметры_журнала]
```

Инициализирует *`каталог_копий`*, в котором будут храниться резервные копии, архив WAL и метаинформация о скопированных кластерах баз данных. Если заданный *`каталог_копий`* уже существует, он должен быть пустым. В противном случае pg_probackup выдаст соответствующее сообщение об ошибке. Вы можете отключить вывод этого сообщения, указав `--skip-if-exists`. Хотя каталог не будет инициализирован, приложение вернёт 0.

За подробностями обратитесь к подразделу Инициализация каталога копий.

add-instance

```
pg_probackup add-instance -B каталог_копий -D каталог_данных --instance=имя_экземпляра
[--skip-if-exists] [--help] [--wal-tree] [параметры_журнала]
```

Инициализирует новый копируемый экземпляр в каталоге *`каталог_копий`* и создаёт файл конфигурации `pg_probackup.conf`, управляющий параметрами pg_probackup, относящимися к кластеру в указанном *`каталоге_данных`*.

`--skip-if-exists`

Позволяет пропустить инициализацию без вызова ошибки, если экземпляр уже существует.

`--wal-tree`

Организует хранение файлов WAL в подкаталогах, оптимизируя дисковое пространство. Это предотвращает перегрузку файловых систем, в особенности NFS (Network File System, сетевая файловая система), чья архитектура неэффективно обрабатывает большое количество файлов в едином каталоге.

За подробной информацией о команде `add-instance` обратитесь к подразделу Определение копируемого экземпляра.

del-instance

```
pg_probackup del-instance -B каталог_копий --instance=имя_экземпляра [--help]
[параметры_журнала]
```

Удаляет все резервные копии и файлы WAL, связанные с указанным экземпляром.

set-config

```
pg_probackup set-config -B каталог_копий --instance=имя_экземпляра
[--help] [--pgdata=путь_к_pgdata]
[--retention-redundancy=избыточность][--retention-window=окно][--wal-depth=глубина_wal]
[--compress-algorithm=алгоритм_сжатия] [--compress-level=уровень_сжатия]
[-d имя_базы] [-h сервер] [-p порт] [-U имя_пользователя]
[--archive-timeout=тайм-аут] [--external-dirs=путь_внешнего_каталога]
[--restore-command=команда]
[параметры_удалённого_режима] [параметры_удалённого_архива_wal] [параметры_журнала]
```

Добавляет заданные параметры соединения, сжатия, хранения, ведения журнала и указания внешних каталогов в конфигурационный файл `pg_probackup.conf` либо изменяет ранее заданные значения.

Все поддерживаемые параметры описываются в подразделе Параметры.

Редактировать `pg_probackup.conf` вручную **не рекомендуется**.

set-backup

```
pg_probackup set-backup -B каталог_копий --instance=имя_экземпляра -
i ид_резервной_копии
{--ttl=время_жизни | --expire-time=время}
[--note=заметка_к_копии] [--help] [параметры_журнала]
```

Устанавливает заданные для конкретной резервной копии параметры в конфигурационном файле `backup.control` или изменяет ранее определённые значения.

`--note=`*`заметка_к_копии`*

Задаёт текстовую заметку для резервной копии. Если *`заметка_к_копии`* содержит символы перевода строки, сохранена будет только подстрока до первого перевода строки. Максимальный размер заметки равен 1 КБ. Значение *`'none'`* удаляет текущую заметку.

Все поддерживаемые варианты закрепления описываются в подразделе Параметры закрепления.

show-config

```
pg_probackup show-config -B каталог_копий --instance имя_экземпляра [--format=plain|
json]
[--default-units][параметры_журнала]
```

Выводит все текущие параметры конфигурации pg_probackup, в том числе те, что содержатся в файле `pg_probackup.conf`, размещённом в каталоге `каталог_копий/backups/имя_экземпляра`, и те, что были заданы в командной строке. Вы можете добавить параметр `--format=json` для получения результата в формате JSON. По умолчанию параметры конфигурации выводятся обычным текстом.

Также можно добавить параметр `--no-scale-units`, чтобы выводить значения конфигурационных параметров времени и объёма памяти в единицах измерения по умолчанию. В противном случае эти значения преобразуются в более крупные единицы для более удобного отображения. Например, значение параметра `archive-timeout` 300 будет выводиться как `5min`, а значение `archive-timeout` 301 — как `301s`. Кроме того, с указанием `--no-scale-units` параметры конфигурации выводятся без единиц измерения, а значения формата JSON, а также числовые и логические значения не заключаются в кавычки, что упрощает разбор.

Чтобы изменить содержимое `pg_probackup.conf`, используйте команду set-config.

show

```
pg_probackup show -B каталог_копий
[--help] [--instance=имя_экземпляра [-i ид_резервной_копии | --archive]] [--
format=plain|json] [--no-color] [--show-symlinks]
[параметры_журнала]
```

Показывает содержимое каталога копий. Если задано `имя_экземпляра` и `ид_резервной_копии`, выводится подробная информация об этой копии. С указанием `--archive` эта команда показывает содержимое архива WAL в данном каталоге.

По умолчанию содержимое каталога представляется в виде обычного текста. Вы можете передать параметр `--format=json`, чтобы получить результат в формате JSON. С параметром `--no-color` выводимые сообщения не выделяются цветом.

Если указан параметр `--show-symlinks`, команда также показывает ссылки между объединяемыми резервными копиями и исходными полными копиями, с которыми были объединены инкрементальные копии.

Более подробно использование этой команды описывается в подразделах Управление каталогом резервных копий и Просмотр оглавления архива WAL.

backup

```
pg_probackup backup -B каталог_копий -b режим_копирования --instance=имя_экземпляра
[--help] [-j число_потоков] [--progress]
[--backup-threads число_потоков] [--validate-threads число_потоков]
[-C] [--stream [-S slot_name] [--temp-slot[=true|false|on|off]]] [--backup-pg-log]
[--no-validate] [--skip-block-validation]
[-w --no-password] [-W --password] [--allow-group-access]
[--write-rate-limit=скорость_записи]
[--archive-timeout=тайм-аут] [--external-dirs=путь_внешнего_каталога]
[--no-sync] [--note=заметка_к_копии]
[параметры_подключения] [параметры_сжатия] [параметры_удалённого_режима]
[параметры_хранения] [параметры_закрепления] [параметры_журнала]
```

Создаёт резервную копию экземпляра Postgres Pro.

`-b режим`
`--backup-mode=режим`

> Выбирает режим резервного копирования. Поддерживаются следующие режимы: `FULL`, `DELTA`, `PAGE` и `PTRACK`.

`--backup-threads число_потоков`

> Указывает количество потоков для копирования файлов. Переопределяет параметр `j/--threads` для копирования файлов.

`--validate-threads` *`число_потоков`*

Указывает количество потоков для проверки резервной копии. Переопределяет параметр `j`/`--threads` для проверки резервной копии.

`-C`
`--smooth-checkpoint`

Растягивает выполнение контрольной точки во времени. По умолчанию pg_probackup пытается произвести контрольную точку максимально быстро.

`--stream`

Создаёт потоковую резервную копию (STREAM), включая в неё все необходимые файлы WAL, получаемые от сервера по протоколу репликации.

`--temp-slot[=true|false|on|off]`

Создаёт *временный* слот физической репликации для передачи WAL с архивируемого экземпляра Postgres Pro. По умолчанию `--temp-slot` включён. Это гарантирует, что все нужные сегменты WAL будут доступны, если в процессе копирования произойдёт переключение сегментов WAL. Этот параметр может использоваться только вместе с параметром `--stream`. По умолчанию имя слота — `pg_probackup_slot`. Чтобы его поменять, воспользуйтесь параметром `--slot`/`-S` и явно укажите `--temp-slot` или `--temp-slot=true|on`.

`-S` *`имя_слота`*
`--slot=`*`имя_слота`*

Задаёт слот репликации, к которому будет выполнено подключение для передачи WAL. Этот параметр можно указать только вместе с параметром `--stream`.

`--backup-pg-log`

Включает в резервную копию каталог `log`. Этот каталог обычно содержит журналы сообщений сервера. По умолчанию каталог `log` в копию не включается.

`-E` *`путь_внешнего_каталога`*
`--external-dirs=`*`путь_внешнего_каталога`*

Включает в создаваемую копию указанный каталог, рекурсивно копируя его содержимое в отдельный подкаталог каталога резервной копии. Этот параметр полезен для архивирования скриптов, SQL-дампов и файлов конфигурации, расположенных вне каталога данных. Если вы хотите архивировать несколько внешних каталогов, их пути нужно разделять двоеточием в Linux или точкой с запятой в Windows.

`--allow-group-access`

Предоставляет доступ на чтение резервных копий на сервере резервного копирования каждому пользователю операционной системы сервера, включённому в ту же группу, что и пользователь, от имени которого выполняются все операции pg_probackup.

`--write-rate-limit=`*`скорость_записи`*

Задаёт скорость записи данных на диск (в мегабайтах/гигабайтах в секунду). По умолчанию указывается в мегабайтах в секунду. Например: `--write-rate-limit=1GBps` или `--write-rate-limit=100` (мегабайты в секунду). Значение по умолчанию: 0 (без ограничений).

Если значение параметра указано, в конце резервного копирования будет выведена следующая информация:

- `written` — объём записанных данных, в МБ.
- `total time` — время между первой и последней записью (в секундах). Обратите внимание, что это не общее время создания резервной копии.

- `sleep time` — время принудительной задержки, в секундах.
- `average rate` — фактическая средняя скорость записи, в мегабайтах в секунду.

Например:

```
INFO: Rate limit: written 14975.445 MB, total time 17.163 s, sleep time 2.370 s,
 average rate 872.560715 MBps
```

`--archive-timeout=`*`время_ожидания`*

Задаёт тайм-аут для архивирования сегментов WAL и потоковой передачи (в секундах). По умолчанию pg_probackup ждёт выполнения этих операций 300 секунд.

`--skip-block-validation`

Отключает проверку контрольных сумм на уровне блоков в процессе резервного копирования.

`--no-validate`

Пропускает автоматическую проверку созданной резервной копии. Этот ключ может быть полезен, если вы регулярно проверяете резервные копии и хотите сократить время создания копии.

`--no-sync`

Не сбрасывать копируемые файлы на диск. Этот флаг позволяет несколько ускорить процесс копирования. Использование этого флага может привести к повреждению данных в случае аварии операционной системы или аппаратного сбоя. Если вы используете его, рекомендуется выполнить команду validate сразу после завершения копирования, чтобы убедиться в отсутствии проблем.

`--note=`*`заметка_к_копии`*

Задаёт текстовую заметку для резервной копии. Если *`заметка_к_копии`* содержит символы перевода строки, сохранена будет только подстрока до первого перевода строки. Максимальный размер заметки равен 1 КБ. Значение *`'none'`* удаляет текущую заметку.

За подробной информацией о параметрах команды обратитесь к подразделам Общие параметры, Параметры соединения, Параметры хранения, Параметры закрепления, Параметры удалённого режима, Параметры сжатия и Параметры ведения журнала.

За подробностями обратитесь к подразделу Создание резервной копии.

restore

```
pg_probackup restore -B каталог_копий --instance=имя_экземпляра
[--help] [--dry-run] [-D каталог_данных] [-i ид_резервной_копии]
[-j число_потоков] [--progress]
[-T СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ] [--external-mapping=СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ] [--
skip-external-dirs]
[-R | --restore-as-replica] [--no-validate] [--skip-block-validation]
[--force] [--no-sync] [--allow-group-access]
[--restore-command=команда]
[--primary-conninfo=строка_подключения]
[-S | --primary-slot-name=имя_слота]
[-X каталог_wal | --waldir=каталог_wal]
[параметры_точки_восстановления] [параметры_журнала] [параметры_удалённого_режима]
[параметры_частичного_восстановления] [параметры_удалённого_архива_wal]
```

Восстанавливает экземпляр Postgres Pro из резервной копии, расположенной в каталоге *`каталог_копий`*. Если вы укажете один или несколько параметров точки восстановления, pg_probackup найдёт ближайшую к этой точке копию и выполнит восстановление до заданной точки. Если ни идентификатор копии, ни параметры точки восстановления не задаются, pg_probackup выполняет восстановление, используя самую последнюю копию.

`-R`
`--restore-as-replica`

Создаёт минимальный файл конфигурации восстановления для облегчения настройки ведомого сервера. Если для соединения репликации требуется пароль, его нужно дополнительно задать вручную в primary_conninfo. Для Postgres Pro версии 11 и ниже параметры восстановления сохраняются в каталоге данных в файле `recovery.conf`, но для версий Postgres Pro, начиная с 12, pg_probackup сохраняет эти параметры в файле `probackup_recovery.conf`.

`--primary-conninfo=`*`строка_подключения`*

Устанавливает заданное значение для параметра primary_conninfo. Это значение учитывается только при использовании флага `-R`.

Пример: `--primary-conninfo="host=192.168.1.50 port=5432 user=foo password=foopass"`

`-S`
`--primary-slot-name=`*`имя_слота`*

Устанавливает заданное значение для параметра primary_slot_name. Это значение учитывается только при использовании флага `-R`.

`-T` *`СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ`*
`--tablespace-mapping=`*`СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ`*

Перемещает табличное пространство из каталога *`СТАРЫЙ_КАТАЛОГ`* в *`НОВЫЙ_КАТАЛОГ`* во время восстановления. И *`СТАРЫЙ_КАТАЛОГ`*, и *`НОВЫЙ_КАТАЛОГ`* должны задаваться абсолютными путями. Если путь содержит знак равно (=), экранируйте этот знак обратной косой чертой. Данный параметр может указываться неоднократно для перемещения нескольких табличных пространств.

`--external-mapping=`*`СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ`*

Перемещает внешний каталог, включённый в резервную копию, из каталога *`СТАРЫЙ_КАТАЛОГ`* в *`НОВЫЙ_КАТАЛОГ`* во время восстановления. И *`СТАРЫЙ_КАТАЛОГ`*, и *`НОВЫЙ_КАТАЛОГ`* должны задаваться абсолютными путями. Если путь содержит знак равно (=), экранируйте этот знак обратной косой чертой. Данный параметр может указываться неоднократно для нескольких каталогов.

`--skip-external-dirs`

Пропускать внешние каталоги, включённые в резервную копию указанием `--external-dirs`. Содержимое этих каталогов не будет восстановлено.

`--skip-block-validation`

Отключает проверку контрольных сумм на уровне блоков для ускорения проверки целостности. При автоматической проверке перед восстановлением будут проверяться только контрольные суммы на уровне файлов.

`--no-validate`

Пропускает проверку резервного копирования. Этот ключ может быть полезен, если вы регулярно проверяете копии и хотите сократить время восстановления данных.

`--allow-group-access`

Предоставляет каждому пользователю операционной системы, включённому в ту же группу, что и владелец кластера, те же права доступа к директории `PGDATA` на сервере баз данных, что и у владельца. Запуск команды `restore` с этим параметром задаёт те же права доступа, что устанавливаются при создании кластера командой

`initdb --Доступ группы`

`--restore-command=команда`

Задаёт значение для параметра restore_command. Например: `--restore-command='cp /mnt/server/archivedir/%f "%p"'`

`--force`

Позволяет игнорировать ошибочное состояние копии. Этот флаг можно использовать, когда требуется восстановить кластер Postgres Pro из повреждённой или некорректной копии. Применяйте его с осторожностью. Если в целевом каталоге `PGDATA` идентификатор системы отличается от того, что указан в копии, при инкрементальном восстановлении с этим флагом содержимое каталога будет перезаписано (тогда как без флага возникнет ошибка). Также в случае перенаправления указанием `--tablespace-mapping` табличных пространств в непустые каталоги содержимое этих каталогов будет удалено.

`--no-sync`

Не сбрасывать восстанавливаемые файлы на диск. Этот флаг позволяет несколько ускорить процесс восстановления. Использование этого флага может привести к повреждению данных в случае аварии операционной системы или аппаратного сбоя. Если такое событие произойдёт, вам потребуется запустить команду restore ещё раз.

`-X каталог_wal`
`--waldir=каталог_wal`

Задать каталог, в который будут записаны файлы WAL. По умолчанию файлы WAL будут записываться в подкаталог `pg_wal` целевого каталога, но с помощью этого параметра их можно поместить в любое место. Путь *`каталог_wal`* должен быть абсолютным; этот путь может не существовать, но если он существует, он должен быть пустым.

За подробной информацией о параметрах команды обратитесь к подразделам Общие параметры, Параметры точки восстановления, Параметры удалённого режима, Параметры удалённого архива WAL, Параметры ведения журнала и Параметры частичного восстановления.

За подробностями обратитесь к подразделу Восстановление кластера.

checkdb

```
pg_probackup checkdb
[-B каталог_копий] [--instance=имя_экземпляра] [-D каталог_данных]
[--help] [-j число_потоков] [--progress]
[--amcheck [--skip-block-validation] [--checkunique] [--heapallindexed]]
[параметры_соединения] [параметры_журнала]
```

Проверяет целостность кластера баз данных Postgres Pro, выявляя физические и логические повреждения.

`--amcheck`

Производит логическую проверку индексов для указанного экземпляра Postgres Pro, если не были выявлены повреждения при проверке файлов данных. Для проверки индексов в базе данных в ней должно быть установлено расширение amcheck или amcheck_next. В базах данных, где amcheck отсутствует, индексы проверяться не будут. Дополнительные параметры `--checkunique` и `--heapallindexed` действуют в зависимости от установленной версии amcheck.

`--checkunique`

Проверяет ограничения уникальности во время логической проверки индексов. Вы можете использовать этот флаг только вместе с флагом `--amcheck`, когда в базе данных установлено расширение amcheck.

Эта проверка возможна, только если в используемой вами версии расширения amcheck функция `bt_index_check` принимает параметр *`checkunique`*.

`--heapallindexed`

Проверяет, отражены ли в индексах все кортежи кучи, которые должны быть проиндексированы. Этот ключ можно использовать только вместе с `--amcheck`.

Эта проверка возможна, только если в используемой вами версии расширения amcheck/amcheck_next функция `bt_index_check` принимает параметр *`heapallindexed`*.

`--skip-block-validation`

Пропустить проверку файлов данных. Вы можете использовать этот ключ вместе с `--amcheck`, чтобы произвести только логическую проверку индексов.

За подробной информацией о параметрах команды обратитесь к подразделам Общие параметры, Параметры соединения и Параметры ведения журнала.

За подробностями обратитесь к подразделу Проверка кластера.

validate

```
pg_probackup validate -B каталог_копий
[--help] [--instance=имя_экземпляра] [-i ид_резервной_копии]
[-j число_потоков] [--progress] [--wal]
[--skip-block-validation]
[параметры_точки_восстановления] [параметры_журнала]
```

Проверяет наличие и целостность всех файлов, необходимых для восстановления кластера. Если *`имя_экземпляра`* не задаётся, pg_probackup проверяет все резервные копии, имеющиеся в каталоге копий. Если вы зададите *`имя_экземпляра`* без дополнительных параметров, pg_probackup проверит все резервные копии, которые имеются для этого экземпляра. Если задать *`имя_экземпляра`* с указанием точки восстановления и/или *`ид_резервной_копии`*, pg_probackup проверит, возможно ли восстановить кластер с этими параметрами. Если задан параметр `--wal`, будет произведена полная проверка архива WAL, а не только сегментов WAL, требующихся для восстановления кластера.

За подробностями обратитесь к подразделу Проверка копии.

merge

```
pg_probackup merge -B каталог_копий --instance=имя_экземпляра -i ид_резервной_копии
[--dry-run] [--help] [-j число_потоков] [--progress] [--no-validate] [--no-sync]
[параметры_журнала]
```

Объединяет копии, относящиеся к одной цепочке инкрементальных копий. Если выбрана полная копия, она будет объединена с первой инкрементальной копией после неё. Если выбрана инкрементальная копия, она будет объединена с родительской полной копией, включая все инкрементальные копии между ними. После выполнения команды полная копия содержит все объединённые данные, а инкрементальные копии удаляются как ненужные.

`--no-validate`

Пропускает автоматическую проверку до и после объединения.

`--no-sync`

Не сбрасывать объединяемые файлы на диск. Этот флаг позволяет несколько ускорить процесс объединения. Использование этого флага может привести к повреждению данных в случае аварии операционной системы или аппаратного сбоя.

За подробностями обратитесь к подразделам Общие параметры и Объединение резервных копий.

delete

```
pg_probackup delete -B каталог_копий --instance=имя_экземпляра
```

```
[--help] [-j число_потоков] [--progress]
[--retention-redundancy=избыточность][--retention-window=окно][--wal-depth=глубина_wal]
 [--delete-wal]
{-i ид_резервной_копии | --delete-expired [--merge-expired] | --merge-expired | --
status=состояние}
[--dry-run] [--no-validate] [--no-sync] [параметры_журнала]
```

Удаляет копию с заданным идентификатором (*ид_резервной_копии*) или запускает процедуру удаления резервных копий или заархивированных файлов WAL, не удовлетворяющих текущей политике хранения.

`--no-validate`

> Пропускает автоматическую проверку до и после объединения копий в соответствии с политикой хранения.

`--no-sync`

> Не сбрасывать объединяемые файлы на диск. Этот флаг позволяет несколько ускорить процесс объединения копий в соответствии с политикой хранения. Использование этого флага может привести к повреждению данных в случае аварии операционной системы или аппаратного сбоя.

За подробностями обратитесь к подразделам Удаление резервных копий, Параметры хранения и Настройка политики хранения.

archive-push

```
pg_probackup archive-push -B каталог_копий --instance=имя_экземпляра
--wal-file-name=имя_файла_wal [--wal-file-path=путь_файлов_wal]
[--help] [--dry-run] [--no-sync] [--compress] [--no-ready-rename] [--overwrite]
[-j число_потоков] [--batch-size=размер_порции]
[--archive-timeout=тайм-аут]
[--compress-algorithm=алгоритм_сжатия]
[--compress-level=уровень_сжатия]
[параметры_удалённого_режима] [параметры_журнала]
```

Копирует файлы WAL в соответствующий подкаталог каталога копий, проверяя целевой экземпляр по *имени_экземпляра* и значению `system-identifier`. Если параметры экземпляра резервной копии и кластера не совпадают, операция копирования не выполняется, и выдаётся ошибка: `Refuse to push WAL segment segment_name into archive. Instance parameters mismatch`. (Отказано в помещении сегмента имя_сегмента в архив. Параметры экземпляра не совпадают.)

Если файлы, которые требуется копировать, уже имеются в каталоге копий, pg_probackup вычисляет и сравнивает их контрольные суммы. В случае совпадения контрольных сумм `archive-push` пропускает соответствующий файл и выдаёт код успешного завершения. Если же они не совпадают, операция `archive-push` завершается с ошибкой. Если вы хотите заменять существующие файлы WAL в случае несовпадения контрольных сумм, добавьте к команде `archive-push` ключ `--overwrite`.

Содержимое каждого файла копируется во временный файл с расширением `.part`. Если такой временный файл уже существует, pg_probackup ждёт, что он исчезнет в течение заданного параметром `archive_timeout` времени, а если этого не происходит, отбрасывает его. После переноса содержимого выполняется атомарная операция переименования. Тем самым гарантируется, что в случае ошибки команды `archive-push` непрерывное архивирование не остановится и что при параллельном архивировании WAL из разных источников в один архив повреждение архива исключено.

Postgres Pro запрашивает сегменты WAL по одному. Для ускорения архивирования вы можете воспользоваться параметром `--batch-size`, определяющим размер порции из нескольких копируемых сегментов WAL. Вместе с параметром `--batch-size` также можно применить указание `-j`, чтобы порции копировались в несколько потоков.

Сегменты WAL, копируемые в архив, по умолчанию (без указания флага `--no-sync`) гарантированно сбрасываются на диск.

Команду `archive-push` можно использовать в значении параметра archive_command Postgres Pro при настройке непрерывного архивирования WAL.

За подробностями обратитесь к подразделам Общие параметры, Параметры архивирования и Параметры сжатия.

archive-get

```
pg_probackup archive-get -B каталог_копий --instance=имя_экземпляра --wal-file-path=путь_файлов_wal --wal-file-name=имя_файла_wal
[-j число_потоков] [--batch-size=размер_порции]
[--prefetch-dir=каталог_предвыборки] [--no-validate-wal]
[--dry-run] [--help] [параметры_удалённого_режима] [параметры_журнала]
```

Копирует файлы WAL из соответствующего подкаталога каталога резервных копий в каталог журнала предзаписи кластера. Эта команда автоматически устанавливается программой pg_probackup в значении параметра `restore_command` при восстановлении архивных копий с применением архива WAL. Устанавливать её вручную не нужно.

Postgres Pro запрашивает сегменты WAL по одному. Для ускорения восстановления вы можете воспользоваться параметром `--batch-size`, определяющим размер порции из нескольких копируемых сегментов WAL. Вместе с параметром `--batch-size` также можно применить указание `-j`, чтобы порции сегментов копировались в несколько потоков.

За подробностями обратитесь к подразделам Общие параметры, Параметры архивирования и Параметры сжатия.

catchup

```
pg_probackup catchup -b режим_синхронизации
--source-pgdata=путь_к_копируемому_каталогу_данных
--destination-pgdata=путь_к_целевому_каталогу_данных
[--help] [-j | --threads=число_потоков] [--dry-run]
[--write-rate-limit=скорость_записи]
[--stream[--temp-slot[=true|false|on|off]] [-P | --perm-slot] [-S | --slot=имя_слота]
[--exclude-path=ПУТЬ]
[-T СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ]
[-X | --waldir=каталог_wal]
[параметры_подключения] [параметры_удалённого_режима]
[параметры_журнала]
```

Создаёт копию экземпляра Postgres Pro, не используя каталог резервных копий.

`-b` *`режим_синхронизации`*
`--backup-mode=`*`режим_синхронизации`*

Выбирает режим синхронизации. Поддерживаются следующие режимы: `FULL`, `DELTA` и `PTRACK`.

`--source-pgdata=`*`путь_к_копируемому_каталогу_данных`*

Задаёт путь к каталогу данных копируемого экземпляра. Каталог может находиться как локально, так и удалённо.

`--destination-pgdata=`*`путь_к_целевому_каталогу_данных`*

Задаёт путь к локальному целевому каталогу данных.

`-j` *`число_потоков`*
`--threads=`*`число_потоков`*

Задаёт число параллельных потоков для процесса `catchup`.

`--stream`

Копирует экземпляр в режиме доставки STREAM, при котором все необходимые файлы WAL передаются с исходного сервера по протоколу репликации. По умолчанию включён для `catchup`.

`--write-rate-limit=`*`скорость_записи`*

Задаёт скорость записи данных на диск (в мегабайтах/гигабайтах в секунду). По умолчанию указывается в мегабайтах в секунду. Например: `--write-rate-limit=1GBps` или `--write-rate-limit=100` (мегабайты в секунду). Значение по умолчанию: 0 (без ограничений).

Если параметр указан, по завершении команды `catchup` выводится следующая информация:

- `written` — объём записанных данных, в МБ.
- `total time` — время между первой и последней записью (в секундах). Обратите внимание, что это не общее время выполнения команды `catchup`.
- `sleep time` — время принудительной задержки, в секундах.
- `average rate` — фактическая средняя скорость записи, в мегабайтах в секунду.

Например:

```
INFO: Rate limit: written 14975.445 MB, total time 17.163 s, sleep time 2.370 s,
 average rate 872.560715 MBps
```

`-x=`*`префикс_пути`*
`--exclude-path=`*`префикс_пути`*

Определяет префикс для файлов, которые не будут копироваться при синхронизации экземпляров Postgres Pro. Такой префикс должен содержать путь относительно каталога данных экземпляра. Если в префиксе указан каталог, ни один файл в этом каталоге не будет копирован.

Предупреждение

Используйте этот параметр с осторожностью, поскольку исключение файлов может привести к неполной синхронизации.

`--temp-slot[=true|false|on|off]`

Создаёт *временный* слот физической репликации для передачи WAL с копируемого экземпляра Postgres Pro. Параметр `--temp-slot` включён по умолчанию. Это гарантирует, что все нужные сегменты WAL будут доступны, если в процессе копирования произойдёт переключение сегментов WAL. Этот параметр можно использовать только вместе с параметром `--stream` и нельзя использовать с параметром `--perm-slot`. По умолчанию имя временного слота — `pg_probackup_slot`. Чтобы его поменять, воспользуйтесь параметром `--slot`/`-S` и явно укажите `--temp-slot` или `--temp-slot=true|on`.

`-P`
`--perm-slot`

Создаёт *постоянный* слот физической репликации для передачи WAL с копируемого экземпляра Postgres Pro. Этот параметр можно использовать только вместе с параметром `--stream` и нельзя использовать с параметром `--temp-slot`. По умолчанию имя постоянного слота — `pg_probackup_perm_slot`, но его можно поменять, воспользовавшись параметром `--slot`/`-S`.

`-S` *`имя_слота`*
`--slot=`*`имя_слота`*

Задаёт слот репликации, к которому будет выполнено подключение для передачи WAL. Этот параметр можно указать только вместе с параметром `--stream`.

`-T СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ`
`--tablespace-mapping=СТАРЫЙ_КАТАЛОГ=НОВЫЙ_КАТАЛОГ`

Перемещает табличное пространство из каталога *СТАРЫЙ_КАТАЛОГ* в *НОВЫЙ_КАТАЛОГ* во время восстановления. И *СТАРЫЙ_КАТАЛОГ*, и *НОВЫЙ_КАТАЛОГ* должны задаваться абсолютными путями. Если путь содержит знак равно (=), экранируйте этот знак обратной косой чертой. Данный параметр может указываться неоднократно для перемещения нескольких табличных пространств.

`-X каталог_wal`
`--waldir=каталог_wal`

Задать каталог, в который будут записаны файлы WAL. По умолчанию файлы WAL будут записываться в подкаталог `pg_wal` целевого каталога, но с помощью этого параметра их можно поместить в любое место. Путь *каталог_wal* должен быть абсолютным; этот путь может не существовать, но если он существует, он должен быть пустым, чтобы команда `catchup` могла выполняться с параметром `--catchup-mode=FULL`.

За подробной информацией о параметрах команды обратитесь к подразделам Общие параметры, Параметры соединения и Параметры удалённого режима.

За подробностями обратитесь к подразделу Клонирование и синхронизация экземпляра Postgres Pro.

maintain

```
pg_probackup maintain --B каталог_копий --instance=имя_экземпляра
[-i ид_резервной_копии] --status-sync [--help] [--dry-run]
[--log-level-console=уровень_сообщений] [--log-level-file=уровень_сообщений]
[--log-format-console=формат_журнала] [--log-format-file=формат_журнала]
[--log-filename=файл_журнала]
```

Выполняет очистку каталога с резервными копиями указанного экземпляра после принудительного прерывания предыдущих команд pg_probackup.

`--status-sync`

Отслеживает устаревшие блокировки и зависшие резервные копии, а также обновляет состояния резервных копий.

За дополнительной информацией о параметрах команды обратитесь к подразделам Общие параметры и Параметры журнала.

Параметры

В этом подразделе описываются параметры командной строки для команд pg_probackup. Если какое-либо значение параметра может быть получено из переменной окружения, имя этой переменной указывается в верхнем регистре ниже параметра командной строки. Некоторые значения могут быть получены из файла конфигурации `pg_probackup.conf`, находящегося в каталоге копий.

За подробностями обратитесь к Подразделу «Настройка pg_probackup».

Если некоторый параметр задаётся несколькими способами, значение в командной строке имеет наивысший приоритет, а значение в `pg_probackup.conf` — наименьший.

Общие параметры

Ниже приведён список параметров общего характера.

`--dry-run`

Выполняет пробный запуск нужной команды, который не вносит никаких реальных изменений: не создаются, не удаляются и не перемещаются файлы на диске. Этот флаг также позволяет проверить правильность всех параметров команды и её готовность к запуску. С `--dry-run` пропускается потоковая трансляция WAL.

`-B` *`каталог`*
`--backup-path=`*`каталог`*
`BACKUP_PATH`

Задаёт абсолютный путь к каталогу копий. Каталог копий — это каталог, в котором хранятся все файлы резервных копий и метаинформация. Поскольку это расположение необходимо задавать почти для всех команд pg_probackup, имеет смысл указать его один раз в переменной окружения `BACKUP_PATH`. В этом случае каждый раз указывать этот путь в командной строке не нужно.

`-D` *`каталог`*
`--pgdata=`*`каталог`*
`PGDATA`

Задаёт абсолютный путь к каталогу данных кластера. Этот параметр является обязательным только для команды add-instance. Другие команды могут получать этот путь из переменной окружения `PGDATA` или из файла конфигурации `pg_probackup.conf`.

`-i` *`ид_резервной_копии`*
`--backup-id=`*`ид_резервной_копии`*

Задаёт уникальный идентификатор резервной копии.

`--instance` *`имя_экземпляра`*

Указывает имя экземпляра Postgres Pro.

`-j` *`число_потоков`*
`--threads=`*`число_потоков`*

Задаёт число параллельных потоков, запускаемых командами `backup`, `restore`, `merge`, `validate`, `checkdb` и `archive-push`.

`--progress`

Включает вывод прогресса выполнения операций.

`--help`

Выводит подробную информацию по параметрам, которые принимает эта команда.

Параметры точки восстановления

Если настроено непрерывное архивирование WAL, вы можете передать один из этих параметров с командой restore или validate, чтобы указать момент, до которого должен быть восстановлен или проверен кластер баз данных.

`--recovery-target=immediate|latest`

Определяет, когда остановить восстановление:

- Со значением `immediate` восстановление завершается сразу после достижения согласованного состояния выбранной копии, либо последней копии из имеющихся, если параметр `-i`/`--backup-id` опущен. Такое поведение по умолчанию применяется для копий типа STREAM.
- Со значением `latest` восстановление продолжается до тех пор, пока не будут применены все имеющиеся в архиве сегменты WAL. При указании этого значения для параметра `--recovery-target` такое же значение задаётся для параметра `--recovery-target-timeline`.

`--recovery-target-timeline=`*`линия_времени`*

Выбирает линию времени для восстановления:

- `current` — линия времени указанной резервной копии. Это значение по умолчанию.
- `latest` — линия времени последней доступной резервной копии.
- Числовое значение.

`--recovery-target-lsn=`*`lsn`*

Указывает последовательный номер в журнале предзаписи, до которого будет производиться восстановление.

`--recovery-target-name=`*`имя_цели_восстановления`*

Указывает именованную точку сохранения, вплоть до которой будет восстановлен кластер.

`--recovery-target-time=`*`время`*

Указывает точку времени, вплоть до которой будет производиться восстановление. Если часовой пояс не указывается, подразумевается местное время.

Например: `--recovery-target-time="2027-04-09 18:21:32+00"`

`--recovery-target-xid=`*`ид_транзакции`*

Указывает идентификатор транзакции, вплоть до которой будет производиться восстановление.

`--recovery-target-inclusive=`*`boolean`*

Указывает на необходимость остановки сразу после (`true`) либо до (`false`) достижения целевой точки. Этот параметр можно использовать только вместе с параметром `--recovery-target-time`, `--recovery-target-lsn` или `--recovery-target-xid`. Значение по умолчанию определяется параметром recovery_target_inclusive.

`--recovery-target-action=pause|promote|shutdown`

Задаёт действие (recovery_target_action), которое должен выполнить сервер по достижении цели восстановления.

По умолчанию: `pause`

Параметры сохранения

Эти параметры могут использоваться с командами backup и delete.

Подробнее о политике хранения рассказывается в подразделе Настройка политики хранения.

`--retention-redundancy=`*`избыточность`*

Указывает, сколько полных резервных копий должно сохраняться в каталоге данных. Должно быть неотрицательным целым числом. Ноль отключает сохранение.

По умолчанию: `0`

`--retention-window=`*`окно`*

Количество дней, в течение которого возможно восстановление. Должно быть неотрицательным целым числом. При нулевом значении окно восстановления отсутствует.

По умолчанию: `0`

`--wal-depth=`*`глубина_wal`*

Количество резервных копий на каждой линии времени, которое должно сохраняться для обеспечения возможности восстановления PITR. Должно быть неотрицательным целым числом. При нулевом значении этот параметр отключается.

По умолчанию: `0`

`--delete-wal`

Удаляет файлы WAL, которые не являются необходимыми для восстановления кластера из имеющихся резервных копий.

`--delete-expired`

Удаляет резервные копии, не удовлетворяющие политике сохранения, определённой в файле конфигурации `pg_probackup.conf`.

`--merge-expired`

Объединяет самую старую инкрементальную копию, удовлетворяющую требованиям политики хранения, с её родительскими копиями, срок хранения которых истёк.

`--dry-run`

Выводит текущее состояние всех имеющихся резервных копий, но никакие операции, например, удаление или объединение старых копий, при этом не производятся.

Параметры закрепления

Эти параметры могут использоваться с командами backup и set-backup.

За подробностями обратитесь к подразделу Закрепление резервных копий.

`--ttl=`*`время_жизни`*

Задаёт время, на которое закрепляется резервная копия. Значение должно быть неотрицательным целым. Нулевое значение отменяет установленное ранее закрепление резервной копии. Поддерживаются следующие единицы измерения: ms (миллисекунды), s (секунды), min (минуты), h (часы), d (дни). По умолчанию подразумеваются секунды.

Например: `--ttl=30d`

`--expire-time=`*`время`*

Определяет момент времени, до которого будет храниться резервная копия. Время должно задаваться в формате ISO-8601. Если часовой пояс не указывается, подразумевается местное время.

Например: `--expire-time="2027-04-09 18:21:32+00"`

Параметры ведения журнала

Эти параметры могут использоваться с любой командой.

`--no-color`

Отключает цветовое выделение сообщений уровней `warning` и `error` в консоли.

`--log-level-console=`*`уровень_сообщений`*

Управляет уровнем сообщений, которые будут выводиться в журнал консоли. Допустимые уровни: `verbose`, `log`, `info`, `warning`, `error` и `off`. Каждый уровень включает все последующие, и с каждым последующим уровнем объём сообщений уменьшается. Вариант `off` отключает вывод в журнал консоли.

По умолчанию: `info`

> **Примечание**
>
> Все выводимые в консоль сообщения журнала передаются через stderr, чтобы их можно было отделить от вывода команд show и show-config.

`--log-level-file=`*`уровень_сообщений`*

Управляет уровнем сообщений, которые будут выводиться в файл журнала. Допустимые уровни: `verbose`, `log`, `info`, `warning`, `error` и `off`. Каждый уровень включает все последующие, и с каждым последующим уровнем объём сообщений уменьшается. Вариант `off` отключает вывод в файл журнала.

По умолчанию: `off`

`--log-filename=`*`файл_журнала`*

Определяет имена для создаваемых файлов журналов. Имена файлов обрабатываются по шаблону `strftime`, так что вы можете использовать спецкоды с % для выбора имён файлов, зависящих от времени.

По умолчанию: `pg_probackup.log`

Например, если задать шаблон `pg_probackup-%u.log`, pg_probackup будет записывать журнал в отдельные файлы по дням недели, и символы `%u` в имени будут заменяться соответствующим десятичным номером: `pg_probackup-1.log` в понедельник, `pg_probackup-2.log` во вторник и т. д.

Этот параметр действует, если включена запись в журнал (параметром `--log-level-file`).

`--error-log-filename=`*`файл_журнала_событий`*

Определяет имена только для файлов журналов ошибок. Имена файлов обрабатываются по шаблону `strftime`, так что вы можете использовать спецкоды с % для выбора имён файлов, зависящих от времени.

По умолчанию: none

Например, если задать шаблон `error-pg_probackup-%u.log`, pg_probackup будет записывать журнал в отдельные файлы по дням недели, и символы `%u` в имени будут заменяться соответствующим десятичным номером: `error-pg_probackup-1.log` в понедельник, `error-pg_probackup-2.log` во вторник и т. д.

Этот параметр полезен для диагностики и решения возникающих проблем.

`--log-directory=`*`каталог_журнала`*

Определяет каталог, в котором будут создаваться файлы журналов. Вы должны задать в этом параметре абсолютный путь. Этот каталог создаётся только при необходимости, когда в журнал выводится первое сообщение.

По умолчанию: `$BACKUP_PATH/log/`

`--log-format-console=`*`формат_журнала`*

Определяет формат журнала консоли. Устанавливается только из командной строки. Обратите внимание, что этот параметр нельзя указать в файле конфигурации `pg_probackup.conf` посредством команды set-config и что команда backup также воспринимает указание этого параметра в конфигурационном файле как ошибку. Этот параметр может иметь следующие значения:

- Если `plain`, то журнал выводится на консоль в формате обычного текста.
- Если `json`, то журнал выводится на консоль в формате JSON.

По умолчанию: `plain`

`--log-format-file=`*`формат_журнала`*

Определяет используемый формат файлов журнала. Этот параметр может иметь следующие значения:

- Если `plain`, то файлы журнала записываются в формате обычного текста.
- Если `json`, то файлы журнала записываются в формате JSON.

По умолчанию: `plain`

`--log-rotation-size=`*`размер_журнала_для_ротации`*

Максимальный размер отдельного файла журнала. Если это значение достигается, файл журнала прокручивается при выполнении какой-либо команды pg_probackup, за исключением `help` и `version`. Нулевое значение отключает прокрутку в зависимости от размера. Поддерживаются следующие единицы: kB (по умолчанию), MB, GB, TB.

По умолчанию: `0`

`--log-rotation-age=`*`возраст_журнала_для_ротации`*

Максимальное время жизни отдельного файла журнала. Если это значение достигается, файл журнала прокручивается при выполнении какой-либо команды pg_probackup, за исключением `help` и `version`. Время создания последнего файла журнала сохраняется в `$BACKUP_PATH/log/log_rotation`. Нулевое значение отключает прокрутку по времени. Поддерживаемые единицы: ms (миллисекунды), s (секунды), min (минуты, по умолчанию), h (часы), d (дни).

По умолчанию: `0`

Параметры подключения

Эти параметры могут использоваться с командами backup, catchup и checkdb.

pg_probackup поддерживает все переменные окружения libpq.

`-d` *`имя_бд`*
`--pgdatabase=`*`имя_бд`*
`PGDATABASE`

Задаёт имя базы данных для подключения. Это подключение используется только для управления процессом резервного копирования, так что вы можете подключиться к любой существующей базе. Если этот параметр не задаётся в командной строке, переменной окружения `PGDATABASE` или в конфигурационном файле `pg_probackup.conf`, pg_probackup принимает в качестве имени базы значение переменной окружения `PGUSER` или имя текущего пользователя, если переменная `PGUSER` не задана.

`-h` *`сервер`*
`--pghost=`*`сервер`*
`PGHOST`

Указывает имя системы, в которой работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

По умолчанию: `localhost`

`-p` *`порт`*
`--pgport=`*`порт`*
`PGPORT`

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

По умолчанию: `5432`

`-U` *`имя_пользователя`*
`--pguser=`*`имя_пользователя`*
`PGUSER`

Имя пользователя для подключения.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл .pgpass или переменная окружения

`PGPASSWORD`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Запрашивать пароль. (Устаревший параметр.)

Параметры сжатия

Эти параметры могут использоваться с командами backup и archive-push.

`--compress-algorithm=`*`алгоритм_сжатия`*

Определяет алгоритм, который будет использоваться для сжатия файлов данных. Возможные значения: `zlib`, `lz4`, `zstd`, `pglz` и `none`. Любое значение, отличное от `none`, включает сжатие. При этом сжимаются и файлы данных, и файлы WAL. По умолчанию сжатие отключено. Для команды archive-push алгоритм сжатия `pglz` не поддерживается.

Примечание

Программа pg_probackup поддерживает алгоритмы сжатия, включённые в конкретную версию Postgres Pro. В частности:

- lz4 поддерживается для Postgres Pro версии 14 и выше.
- zstd поддерживается для Postgres Pro версии 15 и выше.

По умолчанию: `none`

`--compress-level=`*`уровень_сжатия`*

Определяет уровень сжатия. Этот параметр можно использовать вместе с параметром `--compress-algorithm`. Возможные значения зависят от указанного алгоритма сжатия:

- 0 — 9 для `zlib`
- 1 для `pglz`
- 0 — 12 для `lz4`
- -131072 — 22 для `zstd`

При значении 0 для всех алгоритмов сжатия устанавливается уровень сжатия по умолчанию, равный `1`:

Примечание

Алгоритм сжатия lz4 имеет только один уровень — 1. Так что, если указан алгоритм сжатия `lz4`, а уровень сжатия в `--compress-level` установлен выше 1, фактически используется алгоритм lz4hc, который работает значительно медленнее, но обеспечивает лучшее сжатие.

По умолчанию: `1`

`--compress`

Задаёт алгоритм сжатия по умолчанию и устанавливает `--compress-level=1`. Алгоритм по умолчанию выбирается среди поддерживаемых Postgres Pro в соответствии с приоритетами: `zstd` (самый высокий) -> `lz4` -> `zlib` -> `pglz`. Параметр `--compress` переопределяет параметры

`--compression-algorithm` и `--compress-level` и не может быть использован одновременно с ними.

Параметры архивации

Эти параметры могут использоваться с командой archive-push в значении параметра archive_command и с командой archive-get в значении restore_command.

Дополнительно вы можете задать параметры удалённого режима и ведения журнала.

`--wal-file-path=`*`путь_файлов_wal`*

Задаёт путь файла WAL в *`archive_command`* и *`restore_command`*. В качестве значения для данного параметра укажите `%p` или явно задайте путь к файлу вне каталога данных. Если этот параметр не задан, используется путь, заданный в файле `pg_probackup.conf`.

`--wal-file-name=`*`имя_файла_wal`*

Задаёт имя файла WAL в *`archive_command`* и *`restore_command`*. В качестве значения для данного параметра укажите `%f` для правильной его обработки. Если значением параметра `--wal-file-path` является путь вне каталога данных, следует явно указывать имя файла.

`--overwrite`

Разрешает перезапись файлов WAL в архиве. Этот ключ действует при выполнении команды archive-push, когда указанный подкаталог каталога резервных копий уже содержит данный файл WAL, и его нужно заменить новой копией. Без этого ключа `archive-push` сообщит, что сегмент WAL уже существует, и прервёт операцию. Если заменяемый файл не изменился, `archive-push` пропускает его, независимо от указания `--overwrite`.

`--batch-size=`*`размер_порции`*

Используется для ускорения процесса архивирования при выполнении `archive-push` или процесса восстановления при выполнении `archive-get`. Задаёт максимальное количество файлов WAL, которое может быть скопировано в архив одним процессом `archive-push` или из архива одним процессом `archive-get`.

`--archive-timeout=`*`время_ожидания`*

Задаёт интервал, по истечении которого существующие файлы `.part` будут считаться потерянными. По умолчанию pg_probackup ждёт исчезновения этих файлов 300 секунд. Этот параметр можно использовать только с командой archive-push.

`--no-ready-rename`

Предотвращает переименование файлов состояния в каталоге `archive_status`. Этот параметр полезен, только когда в *`archive_command`* задано несколько команд, и применять его можно только с командой archive-push.

`--no-sync`

Не сбрасывать копируемые файлы WAL на диск. Этот флаг позволяет несколько ускорить процесс архивации. Использование этого флага может привести к повреждению архива WAL в случае аварии операционной системы или аппаратного сбоя. Данный параметр можно указать только с командой archive-push.

`--prefetch-dir=`*`путь`*

Каталог, в котором будут храниться предзагружаемые сегменты WAL при использовании параметра `--batch-size`. Этот каталог должен находиться в той же файловой системе и ниже той же точки монтирования, что и `PGDATA/pg_wal`. По умолчанию эти сегменты размещаются в каталоге `PGDATA/pg_wal/pbk_prefetch`. Данный параметр может применяться только с командой archive-get.

`--no-validate-wal`

Отключает проверку предзагруженных файлов WAL перед использованием. Применяйте этот параметр, если вы хотите увеличить скорость восстановления. Данный параметр может применяться только с командой archive-get.

Параметры удалённого режима

В этом подразделе описываются параметры, связанные с удалённым выполнением операций pg_probackup по SSH. Эти параметры могут использоваться с командами add-instance, set-config, backup, catchup, restore, archive-push и archive-get.

Подробнее о настройке и использовании удалённого режима рассказывается в Подразделе «Настройка удалённого режима» и Подразделе «Использование pg_probackup в удалённом режиме».

`--remote-proto=`*`протокол`*

Задаёт протокол для удалённого выполнения операций. В настоящее время поддерживается только протокол SSH. Возможные значения:

- `ssh` — включает удалённый режим с использованием SSH. Этот вариант используется по умолчанию.
- `none` — явным образом отключает удалённый режим.

Этот параметр можно не задавать, если указывается `--remote-host`.

`--remote-host=`*`целевой_адрес`*

Задаёт имя или IP-адрес целевого удалённого сервера.

`--remote-port=`*`порт`*

Задаёт целевой порт на удалённом сервере.

По умолчанию: `22`

`--remote-user=`*`имя_пользователя`*

Задаёт имя пользователя на удалённом сервере для SSH-соединения. В отсутствие этого параметра используется имя текущего пользователя, устанавливающего SSH-соединения.

`--remote-path=`*`путь`*

Задаёт каталог, в котором pg_probackup установлен на удалённой системе.

`--ssh-options=`*`параметры_ssh`*

Задаёт строку параметров командной строки для SSH. Например, следующим образом можно установить свойства *`keep-alive`* для SSH-подключений, которые будет открывать pg_probackup: `--ssh-options="-o ServerAliveCountMax=5 -o ServerAliveInterval=60"`. Полный список всех параметров можно найти в *руководстве по ssh_config*.

Параметры удалённого архива WAL

В этом подразделе описываются параметры, позволяющие задать аргументы archive-get для использования удалённого режима в команде restore_command при восстановлении PITR или восстановлении копий типа ARCHIVE.

`--archive-host=`*`целевой_адрес`*

Задаёт значение аргумента `--remote-host` для команды `archive-get`.

`--archive-port=`*`порт`*

Задаёт значение аргумента `--remote-port` для команды `archive-get`.

По умолчанию: `22`

`--archive-user=`*`имя_пользователя`*

Задаёт значение аргумента `--remote-user` для команды `archive-get`. В отсутствие этого указания используется имя пользователя, запускающего кластер Postgres Pro.

По умолчанию: пользователь Postgres Pro

Параметры инкрементального восстановления

В этом подразделе описываются параметры, связанные с инкрементальным восстановлением кластера. Эти параметры могут передаваться с командой restore.

`-I` *`инкрементальный_режим`*
`--incremental-mode=`*`инкрементальный_режим`*

Выбирает инкрементальный режим. Поддерживаются следующие режимы:

- `CHECKSUM` — заменять только страницы с неподходящей контрольной суммой и LSN.
- `LSN` — заменять только те страницы, LSN которых больше точки расхождения.
- `NONE` — обычное восстановление.

Параметры частичного восстановления

В этом подразделе описываются параметры, связанные с частичным восстановлением кластера. Эти параметры могут передаваться с командой restore.

`--db-exclude=`*`имя_бд`*

Задаёт имя базы данных, которая должна быть исключена из числа восстанавливаемых. Все остальные базы данных в кластере будут восстанавливаться, включая `template0` и `template1`. Этот параметр можно задать несколько раз, таким образом исключив несколько баз данных.

`--db-include=`*`имя_бд`*

Задаёт имя базы данных, которая должна восстанавливаться. Все остальные базы данных восстанавливаться не будут, за исключением `template0` и `template1`. Этот параметр можно задать несколько раз, таким образом выбрав для восстановления несколько баз данных.

Параметры тестирования и отладки

В этом подразделе описываются параметры, полезные лишь при тестировании или разработке.

`PGPROBACKUP_TESTS_SKIP_HIDDEN`

Указывает pg_probackup игнорировать копии, помеченные как скрытые. Заметьте, что сама утилита pg_probackup никогда не помечает копии как скрытые. Добиться такого состояния копии можно, только вручную отредактировав файл `backup.control`. Задать этот параметр можно только в переменных окружения.

`--destroy-all-other-dbs`

По умолчанию pg_probackup завершает работу ошибкой при попытке выполнить частичное инкрементальное восстановление, поскольку при этом удаляются базы данных, не включённые в список восстановления. Этот флаг позволяет игнорировать ошибку и продолжать частичное инкрементальное восстановление (например, чтобы снимок тестовой БД находился в том же состоянии, что и снимок производственной БД). Этот параметр можно использовать с командой restore.

> **Важно**
>
> Никогда не используйте этот флаг в производственном кластере.

`--lock-lifetime`

Задаёт время жизни блокировок, устанавливаемых при копировании, в секундах. Значение по умолчанию — 300. Минимальное значение — 60.

`PGPROBACKUP_TESTS_SKIP_EMPTY_COMMIT`

Указывает pg_probackup пропускать пустые транзакции после pg_backup_stop.

Версионирование

При разработке pg_probackup используется *семантическое* версионирование.

Авторы

Postgres Professional, Москва, Россия.

Благодарности

Программа pg_probackup основана на pg_arman, которая изначально была написана в NTT, а затем её развивал и поддерживал Микаэль Пакье.

vacuumlo

vacuumlo — удалить потерянные большие объекты из базы данных Postgres Pro

Синтаксис

`vacuumlo` [*параметр*...] *dbname*...

Описание

Программа vacuumlo представляет собой простую утилиту, которая удаляет все «потерянные» большие объекты из базы данных Postgres Pro. Потерянным большим объектом (БО) считается такой БО, OID которого не фигурирует ни в каком столбце `oid` или `lo` в базе данных.

Если вы применяете эту утилиту, вас также может заинтересовать триггер `lo_manage` в модуле lo. Триггер `lo_manage` полезен тем, что стремится предотвратить образование потерянных БО.

Обработке подвергаются все базы данных, перечисленные в командной строке.

Параметры

vacuumlo принимает следующие аргументы командной строки:

`-l` *предел*
`--limit=`*предел*

Удалять в одной транзакции ограниченное количество больших объектов (максимальное количество задаёт *предел*, по умолчанию 1000). Так как сервер запрашивает блокировку для каждого удаляемого БО, удаление слишком большого количества БО в одной транзакции чревато превышением лимита max_locks_per_transaction. Если вы всё же хотите, чтобы все удаления происходили в одной транзакции, установите этот предел, равным нулю.

`-n`
`--dry-run`

Не удалять ничего, только показать, какие операции должны были выполняться.

`-v`
`--verbose`

Выводить подробные сообщения о прогрессе.

`-V`
`--version`

Вывести версию vacuumlo и завершиться.

`-?`
`--help`

Вывести справку об аргументах командной строки vacuumlo и завершиться.

vacuumlo также принимает в командной строке следующие аргументы, задающие параметры подключения:

`-h` *сервер*
`--host=`*сервер*

Адрес сервера баз данных.

`-p` *порт*
`--port=`*порт*

Порт сервера баз данных.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя, под которым производится подключение.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как vacuumlo запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, vacuumlo лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Переменные окружения

`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию.

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Переменная окружения `PG_COLOR` выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

Примечания

Программа vacuumlo работает следующим образом: сначала vacuumlo строит временную таблицу, содержащую все OID больших объектов в выбранной базе данных. Затем она сканирует все столбцы в базе данных, имеющие тип `oid` или `lo`, и удаляет соответствующие записи из временной таблицы. (Замечание: рассматриваются только типы именно с такими именами, а не, например, домены на их базе.) Оставшиеся записи во временной таблице указывают на потерянные БО, которые затем и удаляются.

Автор

Питер Маунт `<peter@retep.org.uk>`

I.2. Сторонние клиентские приложения

В этом разделе рассматриваются сторонние клиентские приложения, включённые в дистрибутив Postgres Pro Standard. Их можно запускать откуда угодно, независимо от того, где находится сервер баз данных. Также обратите внимание, что доступен драйвер odbc (за подробностями обратитесь к Таблице 16.3).

ldap2pg

ldap2pg — автоматизирует процесс создания, обновления и удаления ролей и пользователей Postgres Pro из корпоративного каталога

Синтаксис

`ldap2pg` [*параметр*...] [*имя_БД*]

Описание

Встроенный функционал Postgres Pro включает проверку пароля существующей роли через протокол LDAP. ldap2pg позволяет автоматизировать процесс создания, обновления и удаления ролей и пользователей Postgres Pro из корпоративного каталога.

С задачами по управлению ролями связаны задачи по управлению правами пользователей, поскольку каждой роли должны быть предоставлены надлежащие права по умолчанию. Postgres Pro также может предоставлять и отзывать права.

Утилита ldap2pg поставляется вместе с Postgres Pro Standard в отдельном пакете `ldap2pg` (подробные инструкции по установке приведены в Главе 16).

ldap2pg может работать с *Samba DC*, *OpenLDAP*, *FreeIPA*, Oracle Internet Directory и Microsoft Active Directory.

Для ldap2pg **требуется** файл конфигурации с именем `ldap2pg.yaml`.

Возможности

- Чтение настроек из файла конфигурации в формате YAML.
- Создание, изменение и удаление ролей Postgres Pro через поисковые запросы LDAP.
- Создание статических ролей в YAML-файле конфигурации для заполнения записей LDAP.
- Управление родительскими ролями (псевдоним *группы*).
- Предоставление и отзыв прав статически или через записи LDAP.
- Пробный режим, режим проверки.
- Регистрация поисковых запросов LDAP в журнале в качестве команд `ldapsearch(1)`.
- Регистрация в журнале **всех** SQL-операторов.

Требования

ldap2pg выпускается как отдельный бинарный файл без зависимостей.

Для запуска утилиты ldap2pg требуется доступ суперпользователя или роли без прав суперпользователя, но с правами `CREATEDB` и `CREATEROLE`.

Необязательно запускать ldap2pg на том же компьютере, что и синхронизированный кластер Postgres Pro Standard.

Двух МиБ ОЗУ и одного виртуального процессора достаточно, чтобы утилита ldap2pg за секунды могла синхронизировать несколько тысяч ролей. Время синхронизации зависит от экземпляра Postgres Pro Standard и скорости отклика каталога LDAP.

Справка по командной строке

Расширение ldap2pg предоставляет гибкие возможности конфигурации, оно совместимо с psql и утилитами OpenLDAP и соответствует принципам *Twelve-Factor App*. ldap2pg читает конфигурацию из разных источников в следующем порядке (первый источник из списка читается последним):

1. параметры командной строки;
2. переменные окружения;
3. файл конфигурации;
4. `ldaprc`, `ldap.conf` и т. д.

Синтаксис командной строки выглядит следующим образом:

```
ldap2pg [OPTIONS] [dbname]
```

. Можно использовать следующие параметры:

`--check`

Устанавливает режим проверки: осуществляется выход с кодом 1, если экземпляр Postgres Pro не синхронизирован.

`--color`

Включает цветной вывод.

`-c string`
`--config string`

Задаёт путь до YAML-файла конфигурации. Используйте - для stdin.

`--directory string`

Задаёт путь до каталога, который содержит файлы конфигурации.

`--help`

Выводит справку и завершает работу (значение по умолчанию: true).

`--ldappassword-file string`

Задаёт путь к файлу с паролем LDAP.

`--quiet count`

Уменьшает уровень детализации журнала.

`--real`

Режим действия. Применяет изменения к экземпляру Postgres Pro.

`-skip-privileges`

Выключает синхронизацию прав.

`--verbose count`

Увеличивает уровень детализации журнала.

`--version`

Выводит информацию о версии и завершает работу (значение по умолчанию: true).

В необязательном параметре `dbname` можно передать имя базы данных, строку conninfo или URI. За дополнительными сведениями обратитесь к странице man psql(1).

По умолчанию ldap2pg работает в пробном режиме. Для описания поисковых запросов LDAP и сопоставлений ldap2pg требуется файл конфигурации.

Параметры можно определять несколько раз. В случае конфликта используется последний аргумент.

Переменные окружения

Для расширения ldap2pg в интерфейсе командной строки нет параметра, позволяющего настроить соединение с Postgres Pro. Однако, ldap2pg поддерживает переменные окружения `PG*` , используемые библиотекой libpq.

За подробной информацией о переменных окружения, используемых библиотекой libpq, обратитесь к *psql(1)*.

По такому же принципу устанавливается соединение с сервером LDAP: ldap2pg поддерживает стандартные переменные окружения `LDAP*` и файлы `.ldaprc`. За подробной информацией о конфигурации обратитесь к странице man `ldap.conf(5)`. ldap2pg принимает две дополнительные переменные: `LDAPPASSWORD` и `LDAPPASSWORD_FILE`.

ldap2pg загружает файл `.env` из родительского каталога `lda2pg.yml`, если такой файл существует.

Для переменных окружения используйте логические значения `true` или `false`, например `LDAP2PG_SKIPPRIVILEGES=true`.

Подсказка

Проверьте соединение с Postgres Pro, используя страницу man psql(1), и с сервером LDAP, используя страницу man ldapwhoami(1). Таким образом можно будет удостовериться, что ldap2pg работает правильно, и в дальнейшем будет легче отладить настройки и конфигурацию.

Настройка регистрации событий

У расширения ldap2pg есть несколько уровней регистрации событий:

- `ERROR`: информация о возникшей ошибке. При возникновении ошибки происходит сбой в работе ldap2pg.
- `WARNING`: предупреждение ldap2pg о вариантах, о которых должен знать пользователь.
- `CHANGE`: только изменения, которые применяются к кластеру Postgres Pro>.
- `INFO` (по умолчанию): информация о том, что делает ldap2pg, в особенности, перед длинной задачей.
- `DEBUG`: всё, включая обычные SQL-запросы и поисковые запросы LDAP.

Флаги командной строки `--quiet` и `--verbose` соответственно уменьшают и увеличивают уровень детализации журнала.

Самый высокий уровень детализации можно выбрать с помощью переменной окружения `LDAP2PG_VERBOSITY`. Например:

```
$ LDAP2PG_VERBOSITY=DEBUG ldap2pg

12:23:45 INFO   Starting ldap2pg                                  version=v6.0-alpha5
 runtime=go1.21.0 commit=<none>

12:23:45 WARN   Running a prerelease! Use at your own risks!

12:23:45 DEBUG  Searching configuration file in standard locations.

12:23:45 DEBUG  Found configuration file.                         path=./ldap2pg.yml

$
```

Формат вывода ldap2pg зависит от того, используется ли терминал TTY. Если стандартный поток ошибок выводится на терминал TTY, записи цветные и их легко прочитать пользователю. Во всех остальных случаях записи выводятся в формате logfmt для машинной обработки. Получить вывод в удобном для чтения пользователем формате можно с помощью флага командной строки `--color`.

Справка по файлу `ldap2pg.yml`

Для ldap2pg требуется файл конфигурации в формате YAML, которому обычно дают имя `ldap2pg.yml` и который помещают в рабочий каталог. Через этот файл можно настроить все необходимые параметры: запросы для проверки Postgres Pro, поисковые запросы LDAP, права и карта синхронизации.

Предупреждение

Для ldap2pg *требуется* файл конфигурации, где будет описана карта синхронизации.

За основу можно взять содержание протестированного файла `ldap2pg.yml`:

```
#
#
#       П Р И М Е Р   К О Н Ф И Г У Р А Ц И И   L D A P 2 P G
#
#
# Это пример файла конфигурации ldap2pg.yml, включающего определение статических
# ролей и групп, предоставление прав и настройку поиска LDAP.
#
# Предполагается, что выполнены следующие требования:
#
# - Все пользователи LDAP должны быть помещены в группу `ldap_roles`.
# - Права для чтения данных должны быть предоставлены пользователям в группе `readers`.
# - Права для записи данных должны быть предоставлены пользователям в группе `writers`.
# - Права для выполнения DDL-запросов должны быть предоставлены пользователям в группе
 `owners`.
# - Должна существовать одна или несколько баз данных со схемой public и, возможно, ещё
 какой-то схемой.
# - Права не должны быть ограничены схемой. То есть если у пользователя есть права для
 записи данных в базу, то они
#   распространяются на все схемы.
#
# Содержание каталога LDAP описано в test/fixtures/openldap-data.ldif
#
# Используйте информацию в соответствии с вашими потребностями!
#
#
# Версия формата файла. Позволяет ldap2pg проверить, поддерживается ли файл.
#
version: 6

#
#       1.   П Р О В Е Р К А   P O S T G R E S
#
#  За подробностями обратитесь к Разделу «Проверка кластера Postgres Pro».
#
postgres:
  roles_blacklist_query: [nominal, postgres, pg_*]
  databases_query: [nominal]

#
#       2.   О П Р Е Д Е Л Е Н И Е   П Р А В
#
# За подробностями обратитесь к Разделу «Управление правами». Права, обрамлённые
# двойным нижним подчёркиванием, относятся к встроенным профилям прав. За подробностями
 обратитесь к соответствующей документации
# в Разделе «Встроенные права»
# .
#

privileges:
  # Определить группу `ro` только с правами для чтения данных
```

```
  ro:
  - __connect__
  - __select_on_tables__
  - __select_on_sequences__
  - __usage_on_schemas__
  - __usage_on_types__

  # Определить группу `rw` только с правами для записи данных
  rw:
  - __temporary__
  - __all_on_tables__
  - __all_on_sequences__

  # Определить группу `ddl` только с правами для выполнения DDL-команд.
  ddl:
  - __create_on_schemas__


#
#       3.   К А Р Т А   С И Н Х Р О Н И З А Ц И И
#
# Ниже приведён список правил для назначения ролей и предоставления прав. Каждое
 правило
# можно превратить в шаблон с помощью атрибутов из записей LDAP, которые возвращает
 поисковой
# запрос.
#
# Все роли, найденные в кластере, но созданные не в соответствии с правилами, будут
 удалены. Все
# права, найденные в кластере, но созданные не в соответствии с правилами, будут
 отозваны.
#

rules:
- description: "Setup static roles and grants."
  roles:
  - names:
    - readers
    options: NOLOGIN
  - name: writers
    # Предоставить права для чтения пользователям с правами для записи данных
    parent: readers
    options: NOLOGIN
  - name: owners
    # Предоставить права для чтения и записи данных пользователю с ролью владельца
    parent: writers
    options: NOLOGIN

  grant:
  - privilege: ro
    role: readers
    # Привязать к конкретной схеме
    schemas: nominal
  - privilege: rw
    role: writers
  - privilege: ddl
    role: owners
```

```
- description: "Search LDAP to create readers, writers and owners."
  ldapsearch:
    base: cn=users,dc=bridoulou,dc=fr
    filter: "
    (|
      (cn=owners)
      (cn=readers)
      (cn=writers)
    )
    "
  role:
    name: '{member.cn}'
    options: LOGIN
    parent: "{cn}"
```

Расположение файла

ldap2pg ищет файл конфигурации в следующем порядке:

1. `ldap2pg.yml` в текущем рабочем каталоге;
2. `~/.config/ldap2pg.yml`;
3. `/etc/ldap2pg.yml`;
4. `/etc/ldap2pg/ldap2pg.yml`.

При заданном `LDAP2PG_CONFIG` или `--config` ldap2pg пропускает поиск файла конфигурации в стандартных местах расположения. Передавая `-`, можно указать, что файл необходимо прочитать из стандартного ввода. Этот способ полезен, если нужно изменить конфигурацию ldap2pg динамически.

Структура файла

В файле `ldap2pg.yml` есть несколько разделов:

- `postgres`: настройка соединения с Postgres Pro> и запросов для проверки.
- `ldap`: конфигурация для клиента LDAP.
- `privileges`: определение профилей прав.
- `rules`: список поисковых запросов LDAP и связанные с ними сопоставления ролей и прав.

Если у вас нет чёткого понимания, с чего начать, возьмите за основу протестированный файл ldap2pg.yml с подробными комментариями.

О YAML

YAML — это расширенный JSON. Документ в формате JSON является корректным YAML-документом. YAML — довольно свободный формат, где отступы имеют значение. Для примера можно посмотреть *шпаргалку по формату YAML*.

В файле `ldap2pg.yaml` будут использоваться звездочки для глобальных шаблонов и фигурные скобки для внедрения атрибутов LDAP. Обратите внимание, что эти значения должны быть заключены в кавычки.

```
rules:

  - role: {cn}  # Неправильный YAML-словарь

  - role: "{cn}"  # Строка с внедрённым атрибутом LDAP
```

Раздел `postgres`

В разделе `postgres` определяются пользовательские SQL-запросы для проверок в Postgres Pro.

Раздел `postgres` содержит несколько параметров `*_query`. Они могут представлять собой строку, содержащую SQL-запрос, или YAML-список для получения статического списка значений без выполнения запроса в кластере Postgres Pro.

`databases_query`

SQL-запрос для получения списка имён баз данных в кластере. По умолчанию ldap2pg ищет те базы данных, с которыми может установить соединение и объекты которых может переназначить владельцу. ldap2pg циклично просматривает базы данных, чтобы переназначить объекты перед удалением роли. ldap2pg управляет правом доступа к каждой базе данных.

```
postgres:

  databases_query: "SELECT datname FROM pg_catalog.pg_databases;"

  # или

  databases_query: [mydb]
```

Примечание

Если параметр `_query` задан как YAML-список, запрос для проверки в кластер не отправляется, а ldap2pg использует статическое значение.

`fallback_owner`

Имя роли, которая получает права владения базой данных вместо удалённой из базы роли.

Перед удалением роли ldap2pg переназначает объекты и очищает список управления доступом (access control list, ACL). Сначала целевой пользователь ldap2pg переназначает владельца базы данных. Новый владелец — это так называемый *запасной владелец (fallback owner)*. Затем другие объекты базы данных переназначаются каждому владельцу.

`managed_roles_query`

SQL-запрос для получения списка имён управляемых ролей.

ldap2pg может производить удаление ролей или изменение прав только для управляемых ролей. Как правило, в результате выполнения такого запроса возвращается список дочерних ролей, входящих в определённую группу, например, роли из группы `ldap_roles`. По умолчанию ldap2pg управляет всеми ролями, к которым имеет доступ.

`public` — это специальная роль, встроенная в Postgres Pro. Если в результатах запроса `managed_roles_query` появляется роль `public`, ldap2pg будет управлять правами в схеме `public`. ldap2pg управляет правами в схеме `public` по умолчанию.

Ниже приведён пример того, как настроить ldap2pg для управления ролью `public`, родительской ролью `ldap_roles` и любой дочерней ролью из группы `ldap_roles`:

```
postgres:

  managed_roles_query: |

    VALUES

      ('public'),

      ('ldap_roles')

    UNION

    SELECT DISTINCT role.rolname

    FROM pg_roles AS role
```

```
    JOIN pg_auth_members AS ms ON ms.member = role.oid

    JOIN pg_roles AS parent

      ON parent.rolname = 'ldap_roles' AND parent.oid = ms.roleid

    ORDER BY 1;
```

`roles_blacklist_query`

SQL-запрос, который возвращает имя роли и глобальный шаблон для добавления роли в черный список. ldap2pg не будет рассматривать эти роли. Значение по умолчанию: `[postgres, pg_*]`. ldap2pg добавляет в черный список своего пользователя.

```
postgres:
  roles_blacklist_query:
  - postgres
  - "pg_*"
  - "rds_*"
```

Предупреждение

Обратите внимание, что '*foo' — элемент в формате YAML. Шаблон, который *вводится* с помощью '*', необходимо заключать в кавычки.

`schemas_query`

SQL-запрос, который возвращает имена управляемых схем базы данных. ldap2pg выполняет этот запрос для баз данных, список которых был получен в результате запроса `databases_query`, только есть ldap2pg управляет правами в них. ldap2pg циклично просматривает все объекты в этих схемах, когда проверяет предоставленные права в кластере.

```
  postgres:
    schemas_query: |
      SELECT nspname FROM pg_catalog.pg_namespace
```

Раздел `ldap`

Раздел `ldap` необходим для настройки поведения клиента LDAP. Настройте соединение, используя файл `ldap.conf` и переменные окружения `LDAP*`.

`known_rdns`

Список атрибутов, которые являются частью уникального имени (distinguished name, DN).

ldap2pg пропускает поиск для атрибутов из этого списка. Например, `{member.cn}` не приведёт к вложенному поиску среди всех членов (member), если атрибут `cn` включён в список `known_rdns`. Значения по умолчанию: `[n, l, st, o, ou, c, street, dc, uid]`. Расширьте список, чтобы ускорить синхронизацию. Удалите значение, если оно не является частью уникального имени.

```
ldap:
  known_rdns: [cn, uid]
```

Раздел `privileges`

В разделе верхнего уровня `privileges` определяются права в Postgres Pro. Он представляет собой сопоставление, определяющее профили прав, на которые в дальнейшем даётся ссылка в правиле для предоставления права в карте синхронизации. Профиль прав — список ссылок на типы прав из ACL Postgres Pro или других профилей. Профиль прав может включать другой профиль рекурсивно. За подробностями обратитесь к разделу Управление правами.

```
privileges:
  reading:
  - on: GLOBAL DEFAULT
    type: SELECT
    object: TABLES
  writing:
  - reading
  - on: GLOBAL
    type: SELECT
    object: TABLES
```

Профиль прав, имя который начинается с _, неактивен, если он не включён в активный профиль.

`object`

Определяет целевой объект в ACL, где права предоставляются на уровне объектов.

Полезен только для ACL `GLOBAL DEFAULT` и `SCHEMA DEFAULT`, где объекты представляют собой классы целевых объектов, такие как `TABLES`, `SEQUENCES` и т. д. Правило предоставления прав определяет целевую схему в случае ACL `SCHEMA DEFAULT`.

```
privileges:
  reading:
  - type: SELECT
    on: GLOBAL DEFAULT
    object: TABLES
```

`type`

Тип права в соответствии с описанием в Разделе 5.8. Например, право `SELECT`, `REFERENCES`, `USAGE` и т. д.

Значение может быть задано одной строкой или списком строк. Форма множественного числа `types` допустима. Когда определяется несколько типов, для каждого типа определяется новое право, используя одинаковые атрибуты, такие как `on`.

```
privileges:
  reading:
  - type: USAGE
    on: SCHEMAS
```

`on`

Целевой ACL для типа права, например, TABLES, SEQUENCES, SCHEMAS и т. д. Обратите внимание на особые случаи `ALL TABLES`, `ALL SEQUENCES` и т. д. За подробностями обратитесь к разделу Управление правами.

```
privileges:
  reading:
  - type: SELECT
    on: ALL TABLES
```

Раздел `rules`

В разделе верхнего уровня `rules` определяются правила синхронизации. Он содержит список в формате YAML. Это единственный обязательный параметр в файле `ldap2pg.yaml`. Каждый элемент `rules` называется *сопоставлением*. Сопоставление — YAML-словарь с подразделом для любой `роли` или `права`. Сопоставление также может включать поле `description` и раздел `ldapsearch`.

```
rules:
- description: "Define DBA roles"
  ldapsearch:
```

```
    base: ...
  roles:
  - name: "{cn}"
    options: LOGIN SUPERUSER
```

Подраздел `ldapsearch` необязательный. Определить роли и предоставить права можно без запросов к каталогу.

`description`

Свободная строка, которая используется для регистрации событий в журнале. В этот параметр нельзя внедрить параметр mustache.

`ldapsearch`

Директива определяет параметры поисковых запросов LDAP. Она названа вслед за утилитой командной строки ldapsearch, входящей в дистрибутив OpenLDAP. Поведение директивы в целом должно совпадать с поведением утилиты.

Примечание

В тексте документации запрос LDAP (LDAP query) называется поисковым запросом LDAP, а термин «запрос» относится к SQL.

В директивах `ldapsearch` в правила назначения ролей (правила `role`) и правила представления прав (правила `grant`) можно и нужно внедрять атрибуты LDAP в фигурных скобках. За подробностями обратитесь к разделу Запрос к каталогу через LDAP.

`base`, `scope` and `filter`

Эти параметры имеют тот же смысл, определение и значение по умолчанию, что и соответствующие параметры утилиты командной строки ldapsearch.

```
rules:
- ldapsearch:
    base: ou=people,dc=acme,dc=tld
    scope: sub
    filter: >
      (&f
         (member=*)
         (cn=group_*)
      )
```

`joins`

Позволяет настроить вложенный поиск LDAP. Раздел `joins` — словарь с именем атрибута в качестве ключа и параметрами поискового запроса LDAP в качестве значения. В этом случае параметры поискового запроса LDAP такие же, как и для изначального полноценного поиска LDAP. Вложенный поиск в рамках другого вложенного поиска не поддерживается.

```
rules:
- ldapsearch:
    joins:
      member:
        filter: ...
        scope: ...
  role:
  - name: "{member.sAMAccountName}"
```

Основа вложенного поиска — значение ссылающегося атрибута, например, каждое значение `member`. Настроить атрибут `base` для вложенного поиска невозможно. Также ldap2pg извлекает атрибуты для вложенных поисков из правил `role` и `grant`. В рамках полноценного поиска доступен только один уровень вложенного поиска. Запустить вложенный поиск внутри другого вложенного поиска невозможно.

За подробностями обратитесь к разделу Запрос к каталогу через LDAP.

Примечание

Выполнить вложенный поиск среди всех записей, полученных в результате полноценного поиска, иногда очень сложно. Запрос можно уточнить с помощью фильтров поиска, таких как `memberOf`. За подробностями обратитесь к вашему администратору каталога LDAP и документации.

`role`

Задаёт правило для описания одной или нескольких ролей, которые требуются в целевом кластере Postgres Pro. Описание включает имя (name), атрибуты (options), конфигурацию (config), членство (membership) роли и комментарий (comment). Форма множественного числа `roles` допустима. Значение может включать как одно правило, так и список правил.

```
rules:
- role:
    name: dba
    options: SUPERUSER LOGIN
- roles:
  - name: group0
    options: NOLOGIN
  - name: group1
    options: NOLOGIN
```

`comment`

Определяет SQL-комментарий к роли. Значение по умолчанию: `Managed by ldap2pg`. В параметр можно внедрять атрибуты LDAP.

При внедрении атрибутов LDAP нужно обращать внимание на количество создаваемых комбинаций. Если шаблон генерирует один комментарий, ldap2pg скопирует комментарий для каждой роли, созданной по правилу для назначения ролей. Если шаблон генерирует несколько комментариев, то ldap2pg связывает имя роли и комментарий. Если сгенерированных комментариев больше или меньше, чем ролей, возникнет сбой в работе ldap2pg.

Пример определения статического комментария, общего для всех созданных ролей:

```
rules:
- roles:
    names:
    - alice
    - bob
    comment: "Static roles from YAML."
```

Пример генерации единого комментария из уникального имени записи LDAP, который копируется для всех созданных ролей:

```
rules:
- ldapsearch:
    ...
  role:
    name: "{cn}"
    comment: "Generated from LDAP entry {dn}."
```

Пример генерации уникального комментария для каждой созданной роли:

```
rules:
- ldapsearch:
    ...
  role:
    name: "{member.cn}"
    comment: "Generated from LDAP entry {member}."
```

Подсказка

Если роль определяется несколько раз, родительские роли объединяются. Другие поля остаются такими же, как в первом определении роли.

`name`

Имя роли, которая требуется в кластере. Значение может включать одну строку или список строк. Форма множественного числа `names` допустима. В параметр можно внедрять атрибуты LDAP в фигурных скобках. Если определяется несколько имён, для каждого имени определяется отдельная роль, используя одинаковые атрибуты, такие как `options` и `parents`. Параметр `comment` подчиняется отдельным правилам, за подробностями обратитесь к пункту `comment`.

```
rules:
- roles:
    name: "my-role-name"
```

При внедрении атрибута LDAP в имя каждое значение атрибута LDAP каждой записи LDAP будет определять новую роль. Если в формате определены несколько атрибутов LDAP, будет сгенерированы все комбинации атрибутов.

ldap2pg заключает имя роли в двойные кавычки в целевом кластере Postgres Pro. Капитализация сохраняется, пробелы допускаются (даже если содержательно они неуместны).

К этому параметру ldap2pg применяет ограничение `roles_blacklist_query`.

`options`

Определяет атрибуты роли Postgres Pro. Параметр может представлять собой строку, похожую на SQL-строку, или YAML-словарь. Допустимые варианты: `BYPASSRLS`, `CONNECTION LIMIT`, `LOGIN`, `CREATEDB`, `CREATEROLE`, `INHERIT`, `REPLICATION` и `SUPERUSER`. Доступные значения зависят от версии целевого кластера Postgres Pro и прав пользователя ldap2pg.

```
- roles:
  - name: my-dba
    options: LOGIN SUPERUSER
  - name: my-group
    options:
      LOGIN: no
      INHERIT: yes
```

`config`

Определяет конфигурационные параметры Postgres Pro, которые будут заданы для роли. Должен быть YAML-словарем. Доступные конфигурационные параметры зависят от версии целевого кластера Postgres Pro. Чтобы задать некоторые параметры, требуются права суперпользователя. Отсутствие прав для установки значений конфигурационного параметра приведёт к сбою в работе ldap2pg.

```
- roles:
```

```
  - name: my-db-writer
    config:
      log_statement: mod
      log_min_duration_sample: 100
```

Если задать параметру `config` значение `null` (значение по умолчанию), функционал для роли будет отключён. Если в параметре `config` словарь, ldap2pg удалит параметры, заданные в кластере, но не определённые в YAML-файле конфигурации ldap2pg. Чтобы сбросить все параметры, задайте в параметре `config` пустой словарь, как в примере ниже.

```
- roles:
  - name: reset-my-configuration
    config: {}
```

Обратите внимание, что атрибуты LDAP не раскрываются в значениях конфигурационных параметров.

`parent`

Имя родительской роли. Допускается список имён. Форма множественного числа `parents` также допустима. Родительская роль назначается с помощью команды `GRANT ROLE parent TO role;`. В параметр `parent` можно внедрять атрибуты LDAP в фигурных скобках. К этому параметру ldap2pg применяет ограничение `roles_blacklist_query`. Родительской ссылкой могут быть локальные роли, которыми не управляет ldap2pg.

```
rules:
- role:
    name: myrole
    parent: myparent
```

`before_create`

SQL-фрагмент, который необходимо выполнить перед созданием роли. В `before_create` можно внедрять атрибуты LDAP в фигурных скобках. Экранирование атрибута с помощью `.identifier()` или `.string()` — ответственность пользователя.

```
rules:
- ldapsearch: ...
  role:
    name: "{cn}"
    before_create: "INSERT INTO log VALUES ({cn.string()})"
```

`after_create`

SQL-фрагмент, который необходимо выполнить после создания роли. В `after_create` можно внедрять атрибуты LDAP в фигурных скобках. Экранирование атрибута с помощью `.identifier()` или `.string()` — ответственность пользователя.

```
rules:
- ldapsearch: ...
  role:
    name: "{sAMAccountName}"
    after_create: "CREATE SCHEMA {sAMAccountName.identifier()} AUTHORIZATION
 {sAMAccountName.identifier()}"
```

`grant`

Отвечает за предоставление прав роли с соответствующими параметрами. Может быть сопоставлением или списком сопоставлений. Форма множественного числа `grants` также допустима.

```
rules:
- grant:
```

```
    privilege: reader
    databases: __all__
    schema: public
    role: myrole
```

`database`

Ограничивает предоставление прав одной или несколькими базами данных. Может быть списком имён. Форма множественного числа `databases` допустима. Специальное значение `__all__` позволяет распространить настройку на все управляемые базы данных, список которых был получен с помощью `databases_query`. Значение по умолчанию: `__all__`. Предоставленные права, найденные в других базах данных, будут отозваны. В параметр можно внедрять атрибуты LDAP в фигурных скобках.

Этот параметр игнорируется, если речь идёт о правах, которые действуют на уровне экземпляра (например, права для настройки `LANGUAGE`).

`privilege`

Имя права из списка прав, определённого в разделе `privileges` YAML-файла конфигурации. Может быть списком имён. Форма множественного числа `privileges` допустима. Параметр обязательный, значение по умолчанию отсутствует. В параметр можно внедрять атрибуты LDAP в фигурных скобках.

`role`

Имя целевой роли предоставляемого права (*роль с предоставленным правом (granted role)* или *правообладатель (grantee)*). Должно быть указано в списке `managed_roles_query`. Может быть списком имён. Форма множественного числа `roles` доступна. В параметр можно внедрять атрибуты LDAP в фигурных скобках. К этому параметру ldap2pg применяет ограничение `roles_blacklist_query`.

`schema`

Имя схемы из списка схем, полученного с помощью `schemas_query`. Специальное значение `__all__` включает *все управляемые схемы баз данных*. Может быть списком имён. Форма множественного числа `schemas` допустима. В параметр можно внедрять атрибуты LDAP в фигурных скобках.

Этот параметр игнорируется для прав доступа к `DATABASE` или для других прав, которые действуют на уровне экземпляра или базы данных.

`owner`

Имя роли, для которой необходимо настроить права по умолчанию. Специальное значение `__auto__` сводит настройку к управляемым ролям с правом `CREATE` в целевой схеме. Может быть списком имён. Форма множественного числа `owners` допустима. В параметр можно внедрять атрибуты LDAP в фигурных скобках.

Раздел `acls`

Определяет списки Postgres Pro ACL. ACL — это набор запросов, которые позволяют отображать список предоставленных прав в кластере и управлять правами, предоставляя и отзывая их для элементов списка. Действие ACL ограничено экземпляром или базой данных. Параметр добавляется в профили прав для ссылки на ACL. Написать пользовательский ACL — непростая задача. Для её выполнения необходимо понимание PostgreSQL и ldap2pg.

`acls`

Раздел верхнего уровня `acls` — сопоставление, определяющее списки ACL. Все поля являются обязательными.

```
acls:
```

```
  PROCEDURE:
    scope: database
    inspect: |
      WITH ...
    grant: GRANT ...
    revoke: REVOKE ...
```

`scope`

Область действия ACL. Параметр может принимать значение `instance` или `database`.

`inspect`

SQL-запрос, который выдаёт список предоставленных прав в кластере. Сигнатура запроса зависит от области действия ACL. За подробностями обратитесь к описанию пользовательского ACL.

`grant`

SQL-запрос для предоставления права. В запрос можно добавить шаблон в угловых скобках. Некоторые параметры внедряются как ключевые слова, то есть как в обычных SQL-запросах. Некоторые заключают в кавычки как идентификаторы.

Доступные параметры:

- `<acl>` имя ACL. Обычный SQL-запрос.
- `<database>` имя базы данных, к которой предоставляют доступ. Идентификатор в кавычках.
- `<grantee>` имя роли, которой предоставляют права. Идентификатор в кавычках.
- `<object>` имя объекта, к которому нужно предоставить доступ. Идентификатор в кавычках.
- `<owner>` имя роли, которой необходимо предоставить права. Идентификатор в кавычках.
- `<privilege>` тип права. Обычный SQL-запрос.
- `<schema>` имя схемы, к которой нужно предоставить доступ. Идентификатор в кавычках.

`revoke`

SQL-запрос, чтобы отозвать право. Как и в запрос `grant`, в этот запрос можно добавить шаблон в угловых скобках. Принимает те же параметры, что и запрос для предоставления прав. Добавление других параметров между GRANT и REVOKE даёт непредсказуемый результат.

Проверка кластера Postgres Pro

Работа ldap2pg подчиняется принципам явного создания/неявного удаления и явного назначения/неявного отзыва. По этой причине проверять кластер на предмет того, что необходимо удалить/отозвать, очень важно для успешной синхронизации.

ldap2pg проверяет базы данных, схемы, роли, владельцев и предоставленные права через SQL-запросы. В разделе `postgres` YAML-файла конфигурации можно настроить такие запросы при помощи параметров, которые заканчиваются `_query`. За подробностями обратитесь к Подразделу «Раздел `postgres`».

Какие базы данных необходимо синхронизировать?

Запрос `databases_query` возвращает плоский список баз данных для управления. В результате выполнения запроса `databases_query` в список должна попасть база данных по умолчанию, определённая в `PGDATABASE`. При удалении ролей ldap2pg циклично просматривает список баз данных, чтобы переназначить объекты и отозвать права, предоставленные ролям, которые будут удалены. Список баз данных также позволяет сузить рамки проверки предоставленных прав. ldap2pg будет отзывать предоставленные права только в базах данных из списка. За подробностями обратитесь к Подразделу «Раздел `postgres`».

```
postgres:
  databases_query: |
    SELECT datname
    FROM pg_catalog.pg_database
    WHERE datallowconn IS TRUE;
```

Синхронизация подмножества ролей

По умолчанию ldap2pg управляет всеми ролями Postgres Pro, к которым имеет доступ, кроме ролей, добавленных в чёрный список по умолчанию. Чтобы ldap2pg синхронизировало только подмножество ролей, необходимо настроить выборку внутри `postgres:managed_roles_query`. Пример запроса ниже позволяет исключить суперпользователей из объектов для синхронизации.

```
postgres:
  managed_roles_query: |
    SELECT 'public'
    UNION
    SELECT rolname
    FROM pg_catalog.pg_roles
    WHERE rolsuper IS FALSE
    ORDER BY 1;
```

ldap2pg будет выполнять операции по удалению, отзыву и предоставлению прав только для тех ролей, которые будут получены в результате этого запроса.

Часто подобный запрос ограничивает выборку одной группой, например, `ldap_roles`. Таким образом ldap2pg работает только с подмножеством ролей в кластере.

Роли `public` нет в системном каталоге. Поэтому чтобы ldap2pg управляло правами `public`, необходимо явно включить роль `public` в список управляемых ролей. Это настройки по умолчанию. Но даже если роль `public` управляемая, ldap2pg не будет удалять или менять её, если её нет в каталоге.

Чтобы полностью исключить некоторые роли из выборки, можно использовать `roles_blacklist_query`.

```
postgres:
  roles_blacklist_query: [postgres, pg_*]  # Это значение по умолчанию.
```

Примечание

Элементы, которые вводятся '*', *необходимо* заключать в кавычки. В противном случае произойдёт ошибка при обработке YAML, например, «found undefined alias (найдены неопределённые псевдонимы)».

Проверка схем

Для управления правами, которые действуют на уровне схем, ldap2pg необходимо знать, какими схемами оно может управлять в рамках каждой базы данных. Для этого используется запрос `schemas_query`.

Настройка прав по умолчанию для роли владельца

Чтобы настроить права по умолчанию, ссылаясь на право, используйте ключевое слово `default`:

```
privileges:
  reading:
  - default: global
    type: SELECT
```

```
    on: TABLES
```

Затем предоставьте право, используя `правило предоставления прав`:

```
rules:
- grant:
  - privilege: reading
    role: readers
    schema: public
    owner: ownerrole
```

В качестве владельца можно использовать роль `__auto__`. Для каждой схемы ldap2pg настроит все управляемые роли, которые имеют право `CREATE` в схеме.

```
rules:
- grant:
  - privilege: reading
    role: readers
    schema: public
    owner __auto__
```

ldap2pg настраивает права по умолчанию в последнюю очередь после всех действующих прав. Таким образом право `CREATE` в схеме предоставляется до того, как ldap2pg проверит роли создателей в схемах.

Статические запросы

Все запросы можно заменить **статическим списком** в формате YAML. Этот список будет использоваться, как будто он получен от Postgres Pro. Это очень удобно, если нужно заморозить какое-то значение, например, указание на базы данных или схемы.

```
postgres:
  databases_query: [postgres]
  schemas_query: [public]
```

Управление ролями

ldap2pg выполняет синхронизацию ролей Postgres Pro в три этапа:

1. Циклично просматривает `rules` и генерирует список с необходимыми ролями из правил `role`.
2. Проверяет в Postgres Pro существующие роли, их атрибуты и членство.
3. Сравнивает два набора ролей и применяет необходимые изменения к кластеру Postgres Pro с помощью команд `CREATE`, `DROP` и `ALTER`.

Каждая запись `role` в `rules` является правилом, по которому роль не создаётся или создаётся одна или несколько ролей с соответствующими параметрами. `role` — это своеобразный шаблон. Правила `role` позволяют избежать дублирования членства и атрибутов роли с помощью заданного списка имён.

В одном файле можно использовать и статические, и динамические правила.

Запуск без прав суперпользователя

Расширение ldap2pg предназначено для работы от имени не суперпользователя. Пользователь, который будет выполнять синхронизацию, должен иметь атрибут `CREATEROLE`, чтобы управлять другими ролями не суперпользователя. Атрибут `CREATEDB` позволяют такому пользователю управлять владельцами баз данных.

Чтобы пользователь ldap2pg мог правильно управлять группами, параметру `createrole_self_grant` должно быть задано значение `inherit,set`.

```
CREATE ROLE ldap2pg LOGIN CREATEDB CREATEROLE;
ALTER ROLE ldap2pg SET createrole_self_grant TO 'inherit,set;
```

Запуск без прав суперпользователя до версии PostgreSQL 16 реализован некорректно. В таких случаях в целях безопасности рекомендуется запускать ldap2pg с правами суперпользователя.

Игнорирование ролей

ldap2pg полностью игнорирует роли, которые соответствуют одному из глобальных шаблонов, определённых в `roles_blacklist_query`:

```
postgres:
  # Это значение по умолчанию.
  roles_blacklist_query: [postgres, pg_*]
```

Черный список ролей также влияет на предоставление прав. ldap2pg не будет выполнять команды `grant` и `revoke` для роли, которая соответствует одному из элементов черного списка.

ldap2pg помещает в чёрный список своего пользователя.

Членство

ldap2pg управляет родительскими ролями. К ним ldap2pg применяет ограничение `roles_blacklist_query`. Однако ldap2pg может предоставлять права неуправляемым родительским ролям. Таким образом можно создавать группу вручную и управлять её членами с помощью ldap2pg.

Запрос из каталога через LDAP

ldap2pg читает поисковые запросы LDAP через записи в `ldapsearch` в порядке, определённом в `rules`.

Поисковой запрос LDAP **не** является обязательным. ldap2pg может создавать роли, определённые статически из YAML-файла конфигурации. Каждый поисковой запрос LDAP выполняется только один раз. Цикличное выполнение или устранение дубликатов для поисковых запросов LDAP невозможно.

> **Подсказка**
>
> ldap2pg регистрирует поисковые запросы LDAP в журнале как команды ldapsearch. Чтобы увидеть записи, включите подробные сообщения.
>
> Чтобы поправить неправильный поисковой запрос, скопируйте команду в вашу оболочку и обновите параметры. Если всё работает, отправьте верные параметры обратно в YAML.

Конфигурация доступа к каталогу

ldap2pg читает конфигурацию каталога из файла `ldaprc` и через переменные окружения `LDAP*`. Доступные параметры LDAP:

- `BASE`
- `BINDDN`
- `PASSWORD`
- `REFERRALS`
- `SASL_AUTHCID`
- `SASL_AUTHZID`
- `SASL_MECH`
- `TIMEOUT`
- `TLS_REQCERT`
- `NETWORK_TIMEOUT`
- `URI`

За подробностями о значении и формате каждого параметра обращайтесь к странице man ldap.conf(5).

Внедрение атрибутов LDAP

В ряд параметров можно внедрить атрибуты LDAP в фигурных скобках. Для этого заключите имя атрибута в фигурные скобки. Например, `{cn}` или `{sAMAccountName}`. ldap2pg расширяется до всех значений атрибута для каждой записи, найденной в результате поиска.

Если в параметре несколько атрибутов LDAP, ldap2pg получает все комбинации атрибутов для каждой записи.

Рассмотрим следующие записи LDAP:

```
dn: uid=dimitri,cn=Users,dc=bridoulou,dc=fr
objectClass: inetOrgPerson
uid: dimitri
sn: Dimitri
cn: dimitri
mail: dimitri@bridoulou.fr
company: external
dn: cn=domitille,cn=Users,dc=bridoulou,dc=fr
objectClass: inetOrgPerson
objectClass: organizationalPerson
objectClass: person
objectClass: top
cn: domitille
sn: Domitille
company: acme
company: external
```

Формат `{company}_{cn}` с приведёнными выше записями LDAP сгенерирует следующие строки:

- `acme_domitille`
- `external_domitille`
- `external_dimitri`

Псевдоатрибут `dn` всегда доступен и ссылается на уникальное имя (DN) исходной записи LDAP.

Доступ к относительному уникальному имени (RDN) и вложенный поиск

Если тип атрибута — DN, можно сослаться на относительное уникальное имя (RDN) через точку. Например, `<attribute>.<rdn>`. Если у относительного уникального имени есть несколько значений, возвращается только первое. Другие значения получить невозможно.

Например, если в записи LDAP есть атрибут `member` со значением `cn=toto,cn=Users,dc=bridoulou,dc=fr`, формат `{member.cn}` сгенерирует `toto`. Формат `{member.dc}` сгенерирует `ldap`. Получить значения `acme` и `fr` не получится.

К известным RDN относятся `cn`, `l`, `st`, `o`, `ou`, `c`, `street`, `dc` и `uid`. Другие атрибуты запускают вложенный поиск. Формат `{member.sAMAccountName}` ограничит базу поиска LDAP атрибутом `sAMAccountName` и запустит вложенный поиск среди всех значений `member`.

Регистр атрибутов LDAP

При внедрении атрибута LDAP в фигурных скобках можно задать регистр с помощью методов `.lower()` или `.upper()`.

```
- ldapsearch: ...
  role: "{cn.lower()}"
```

Управление правами

Управлять правами довольно сложно. ldap2pg пытается сделать этот процесс легче и безопаснее.

Основы

Основной принцип работы ldap2pg таков: вместо того, чтобы отзывать все права и роли и переназначать их, ldap2pg идёт по пути «проверить и изменить». Процесс аналогичен тому, как происходит синхронизация ролей, включая следующие три этапа:

1. Циклично просмотреть `rules` и сгенерировать набор необходимых назначений.
2. Проверить в кластере Postgres Pro назначенные права.
3. Сравнить два набора и обновить значения в кластере Postgres Pro с помощью команд `grant` и `revoke`.

ldap2pg оперирует следующими основными объектами при управлении правами:

- `Privilege`: действие, которое разрешается выполнять с объектом, например, `CONNECT ON DATABASE`.
- `grant`: право, предоставленное роли в отношении объекта.
- `ACL`: список назначений (grant).
- `profile`: список прав.
- `rule`: шаблон для предоставления необходимого права.

В файле `ldap2pg.yml` содержатся профили прав и правила назначения прав. ldap2pg синхронизирует ACL по одному, базу данных за базой данных. В последнюю очередь ldap2pg синхронизирует права по умолчанию.

По умолчанию ldap2pg не управляет правами. Чтобы включить управление правами, сначала необходимо определить хотя бы один активный профиль в разделе `privileges`. Проще всего переиспользовать в активном пользовательском профиле встроенные профили прав, входящие в дистрибутив ldap2pg.

Определение профилей прав

Профиль прав — список ссылок на типы прав, связанные с ACL, или другие профили.ldap2pg устанавливает несколько предопределённых ACL, таких как `DATABASE`, `LANGUAGE` и т. д. Тип права может иметь значение `USAGE`, `CONNECT` и т.д, в соответствии с описанием в Разделе 5.8. За подробностями о формате профиля прав обратитесь к документации, посвященной YAML-разделу `privileges`.

ldap2pg загружает ACL, на который даётся ссылка, через проверку в кластере Postgres Pro с помощью детально прописанных запросов. ldap2pg проверяет только ACL, на которые даётся ссылка хотя бы в одном профиле. Найденные в кластере назначения прав отзываются, если иное явно не прописано в правиле `grant`.

> **Предупреждение**
>
> Если назначение не подпадает под правило, права отзываются!
>
> После проверки ACL ldap2pg *отзывает* все назначенные права, найденные в экземпляре Postgres Pro и не требуемые ни в одном правиле `grant` в разделе `rules`.

Расширенная проверка экземпляра

При управлении ролями ldap2pg проводит более глубокую проверку экземпляра Postgres Pro. После синхронизации ролей и перед синхронизацией прав ldap2pg проверяет схемы. После синхронизации прав и перед синхронизацией прав по умолчанию ldap2pg проверяет владельца объектов. Владелец объекта — это роль, у которой есть право `CREATE` в схеме.

Назначение профиля прав

Проверка миллионов прав отнимает у экземпляра Postgres Pro много ресурсов. Известно, что процесс отзыва прав в Postgres Pro медленный. Лучше всего назначать права группе ролей, чтобы пользователи наследовали права. ldap2pg позволяет определять статические группы в YAML и наследовать их при создании роли из каталога.

Используйте `grant rule`, чтобы назначить профиль прав одной или нескольким ролям. При назначении прав необходимо определить правообладателя. Назначение можно ограничить одной или несколькими базами данных, а также одной или несколькими схемами. Если профиль прав включает права по умолчанию, можно определить владельцев, которым будут назначены права по умолчанию.

По умолчанию назначение применяется ко всем управляемым базам данных, список которых возвращается `databases_query`, а также ко всем схемам каждой базы данных, список которых возвращается `schemas_query`.

Пример

В примере ниже определён профиль с тремя правами. В `rules` определяются три группы ролей, и им назначается соответствующий профиль прав:

```
privileges:
  reading:
  - __connect__
  - __usage_on_schemas__
  - __select_on_tables__
  writing:
  - reading  # включает права чтения
  - __insert_on_tables__
  - __update_on_tables__
  owning:
  - writing
  - __create_on_schemas__
  - __truncate_on_tables__
rules:
- role:
  - names:
    - readers
    - writers
    - owners
    options: NOLOGIN
- grant:
  - privilege: reading
    role: readers
  - privilege: writing
    role: writers
  - privilege: owning
    role: owners
```

Ещё один способ включить права чтения данных в права записи — сделать так, чтобы группа ролей с правами записи (writers) наследовала группу ролей с правами чтения (readers).

Управление правами public

В Postgres Pro есть псевдороль с именем `public`. Это универсальная роль, которая включает *всех пользователей*. Все роли в Postgres Pro неявно наследуют от роли `public`. При назначении права роли `public` это право также назначается всем существующим ролям и ролям, которые будут созданы в дальнейшем.

У Postgres Pro также есть схема `public`. Схема `public` — это существующая схема, которая доступна во всех базах данных.

В Postgres Pro есть несколько встроенных прав для роли `public` и особенно для схемы `public`. Например, у `public` есть право `CONNECT` ко всем базам данных по умолчанию. Это означает, что для конфигурации доступа к базам данных можно использовать только файл `pg_hba.conf`, что требует прав администратора для доступа к кластеру и вызова функции `pg_reload_conf()`.

По умолчанию ldap2pg включает роль `public` в список управляемых ролей. Предопределённый ACL знает, как проверять встроенные права, назначенные роли `public`. Если роль `public` нужно сохранить, исключите роль `public` из `managed_roles_query`.

Управление правами по умолчанию

Если права `SELECT` предоставляются роли на все таблицы в схеме, права не будут распространяться на новые таблицы, созданные после предоставления прав. Вместо того, чтобы после создания каждого нового объекта выполнять ldap2pg заново, в Postgres Pro предусмотрено определение прав по умолчанию для объектов, которые будут созданы в дальнейшем.

Postgres Pro привязывает права по умолчанию к роли создателя (creator). Когда пользователь с этой ролью создаёт объект, Postgres Pro предоставляет соответствующие права по умолчанию новому объекту. Например, благодаря команде `ALTER DEFAULT PRIVILEGES FOR ROLE bob GRANT SELECT ON TABLES TO alice;` в каждой новой таблице, создаваемой пользователем с ролью bob у пользователя с ролью alice будет право `SELECT`:

Если ldap2pg создаёт или удаляет роли создателя, права по умолчанию для этих ролей в ldap2pg должны быть правильно настроены. При управлении правами через ldap2pg обязательно настраивать права по умолчанию для роли создателя.

ldap2pg проверяет создателей в Postgres Pro, по схемам, не в каталоге LDAP. Создатель — это роль с атрибутом `LOGIN` и правом `CREATE` в схеме. Вручную для любой управляемой роли можно назначить целевого владельца предоставляемых прав.

В ldap2pg нельзя настроить права на схемы с атрибутом `__all__`. Вместо этого используйте список ACL `GLOBAL DEFAULT`. Если предоставление и отзыв прав по умолчанию необходимо ограничить схемой, используйте список ACL `SCHEMA DEFAULT`.

В примере ниже для роли alice настраиваются права по умолчанию, благодаря которым у роли bob будет право SELECT для всех таблиц, которые alice будет создавать в дальнейшем.

```
privileges:
  reading:
  - type: SELECT
    on: SCHEMA DEFAULT
    object: TABLES
  owning:
  - type: CREATE
    on: SCHEMAS

rules:
- roles:
    names:
    - alice
    - bob
    options: LOGIN
- grant:
    privilege: owning
    role: alice
- grant:
    privilege: reading
    role: bob
```

В Postgres Pro есть жёстко заданные глобальные права по умолчанию. Если у роли не настроены глобальные права по умолчанию, Postgres Pro подразумевает некоторые права. По умолчанию Postgres Pro предоставляет права только роли владельца (owner). Их можно посмотреть при изменении прав по умолчанию. Postgres Pro скопирует жёстко заданные значения вместе с предоставленными вами правами.

Если явно повторно не предоставить жёстко заданные права в файле `ldap2pg.yml`, ldap2pg отзовет их. Обратите внимание, что владельцу таблицы не нужно предоставлять право SELECT для принадлежащих ему таблиц. Поэтому в жёстко заданных правах по умолчанию нет необходимости. Так что можно разрешить ldap2pg убрать эти ненужные права по умолчанию.

Встроенные права

ldap2pg предоставляет несколько встроенных списков ACL и предопределённых профилей прав для множественного использования. Эти права предоставляются **не гарантированно**. Их настройку в ваших базах данных необходимо проверять точно так же, как и при работе со своим кодом.

Использование предопределённых профилей прав

Профиль прав — список ссылок на тип права в ACL. В ldap2pg ACL — набор запросов для проверки, предоставления и отзыва прав на классы объектов. Запрос на проверку расширяет тип `aclitem` Postgres Pro для отображения всех предоставленных прав из системного каталога. Профиль прав может включать другой профиль.

Имя встроенного профиля обрамляется двойным нижним подчёркиванием (__). ldap2pg отключает профиль прав, который начинается с _. Чтобы включить такие профили, необходимо добавить встроенный профиль прав в другой профиль. Если два профиля ссылаются на одно и то же право, ldap2pg проверит его один раз.

```
privileges:
  ro:
  - __connect__
  - __usage_on_schemas__
  - __select_on_tables__

  rw:
  - ro
  - __insert__

  ddl:
  - rw
  - __all_on_schemas__

rules:
- grant:
    privilege: ddl
    database: mydb
    role: admins
```

Имя встроенного профиля подчиняется следующим нестрогим правилам:

- `..._on_all_tables__` ссылается на список ACL `ALL TABLES IN SCHEMA`. Аналогично для последовательностей и функций.
- `__default_...__` ссылается на права по умолчанию, которые действуют на глобальном уровне и на уровне схемы.
- `__..._on_tables__` объединяет `__..._on_all_tables__` и `__default_..._on_tables__`.
- Группа, чьё имя начинается с `__all_on_...__`, *приравнивается* к разрешению `ALL PRIVILEGES` в SQL. Однако каждое право будет предоставляться отдельно.
- В имени права, относящегося к одному типу объектов, нет суффикса `_on_<type>`. Например, `__delete_on_tables__` — псевдоним `__delete__`.

В этом разделе не описан стандарт SQL и значение каждого права SQL. За подробностями о правах SQL обратитесь к документации команд `GRANT` и `ALTER DEFAULT PRIVILEGES`.

Справка по ACL

Ниже приведён список встроенных списков ACL.

Для действующих прав:

`DATABASE`

Права в базе данных, такие как `CONNECT`, `CREATE` и т. д.

`SCHEMA`

Управление правами `USAGE` и `CREATE` в схеме.

`LANGUAGE`

Управление правом `USAGE` для процедурных языков.

`ALL FUNCTIONS IN SCHEMA`

Управление правом `EXECUTE` для всех функций в рамках схемы.

`ALL SEQUENCES IN SCHEMA`

То же самое, но для последовательностей.

`ALL TABLES IN SCHEMA`

То же самое, но для таблиц и представлений.

`GLOBAL DEFAULT`

Управление правами по умолчанию в базе данных.

`SCHEMA DEFAULT`

Управление правами по умолчанию в рамках схемы.

Список ACL `ALL ... IN SCHEMA` проверяет, предоставлено ли право только подмножеству объектов. Это называется *частичное* предоставление права (partial grant). Частично предоставленное право или отзывается, если оно не нужно, или предоставляется заново, если ожидается.

Дать ссылку на эти списки ACL можно, используя параметр privileges: on в YAML. Пример:

```
privileges:
  myprofile:
  - type: SELECT
    on: ALL TABLES IN SCHEMA
```

Права по умолчанию ссылаются на тип права и класс объектов. ldap2pg проверяет права по умолчанию для следующих классов объектов:

- `SEQUENCES`
- `FUNCTIONS`
- `TABLES`

В профиле прав необходимо дать ссылку на класс объектов с помощью параметра object в YAML.

Настроить пользовательский ACL пока невозможно.

Справка по профилям

Профиль `__all_on_functions__`

`__execute_on_functions__`

Профиль `__all_on_schemas__`

- `__create_on_schemas__`

- __usage_on_schemas__

__all_on_sequences__ **профиль**

- __select_on_sequences__
- __update_on_sequences__
- __usage_on_sequences__

Профиль __all_on_tables__

- __delete_on_tables__
- __insert_on_tables__
- __select_on_tables__
- __truncate_on_tables__
- __update_on_tables__
- __references_on_tables__
- __trigger_on_tables__

Профиль __delete_on_tables__

- __default_delete_on_tables__
- __delete_on_all_tables__

Профиль __execute_on_functions__

- __default_execute_on_functions__
- __execute_on_all_functions__

Профиль __insert_on_tables__

- __default_insert_on_tables__
- __insert_on_all_tables__

Профиль __references_on_tables__

- __default_references_on_tables__
- __references_on_all_tables__

Профиль __select_on_sequences__

- __default_select_on_sequences__
- __select_on_all_sequences__

Профиль __select_on_tables__

- __default_select_on_tables__
- __select_on_all_tables__

Профиль __trigger_on_tables__

- __default_trigger_on_tables__
- __trigger_on_all_tables__

Профиль __truncate_on_tables__

- __default_truncate_on_tables__
- __truncate_on_all_tables__

Профиль __update_on_sequences__

- __default_update_on_sequences__
- __update_on_all_sequences__

Профиль __update_on_tables__

- __default_update_on_tables__
- __update_on_all_tables__

Профиль __usage_on_sequences__

- __default_usage_on_sequences__

- __usage_on_all_sequences__

Справка по правам

Ниже приведён список предопределённых прав:

Имя	**Право**
`__connect__`	`CONNECT ON DATABASE`
`__create_on_schemas__`	`CREATE ON SCHEMA`
`__delete_on_all_tables__`	`DELETE ON ALL TABLES IN SCHEMA`
`__execute_on_all_functions__`	`EXECUTE ON ALL FUNCTIONS IN SCHEMA`
`__insert_on_all_tables__`	`INSERT ON ALL TABLES IN SCHEMA`
`__references_on_all_tables__`	`REFERENCES ON ALL TABLES IN SCHEMA`
`__select_on_all_sequences__`	`SELECT ON ALL SEQUENCES IN SCHEMA`
`__select_on_all_tables__`	`SELECT ON ALL TABLES IN SCHEMA`
	`TEMPORARY ON DATABASE`
`__trigger_on_all_tables__`	`TRIGGER ON ALL TABLES IN SCHEMA`
`__truncate_on_all_tables__`	`TRUNCATE ON ALL TABLES IN SCHEMA`
`__update_on_all_sequences__`	`UPDATE ON ALL SEQUENCES IN SCHEMA`
`__update_on_all_tables__`	`UPDATE ON ALL TABLES IN SCHEMA`
`__usage_on_all_sequences__`	`USAGE ON ALL SEQUENCES IN SCHEMA`
`__usage_on_schemas__`	`USAGE ON SCHEMA`

Справка по правам по умолчанию

Ниже приведён список предопределённых прав по умолчанию. В профиле прав по умолчанию даётся ссылка на права по умолчанию, которые действуют на глобальном уровне и на уровне схемы.

Имя	**Право**
`__default_delete_on_tables__`	`DELETE ON TABLES`
`__default_execute_on_functions__`	`EXECUTE ON FUNCTIONS`
`__default_insert_on_tables__`	`INSERT ON TABLES`
`__default_references_on_tables__`	`REFERENCES ON TABLES`
`__default_select_on_sequences__`	`SELECT ON SEQUENCES`
`__default_select_on_tables__`	`SELECT ON TABLES`
`__default_trigger_on_tables__`	`TRIGGER ON TABLES`
`__default_truncate_on_tables__`	`TRUNCATE ON TABLES`
`__default_update_on_sequences__`	`UPDATE ON SEQUENCES`

Имя	Право
`__default_update_on_tables__`	`UPDATE ON TABLES`
`__default_usage_on_sequences__`	`USAGE ON SEQUENCES`

Подсказки по использованию

В этом разделе даются некоторые рецепты для различных пользовательских сценариев работы с ldap2pg.

Конфигурирование `pg_hba.conf` через LDAP

ldap2pg **НЕ** настраивает Postgres Pro за пользователя. Для конфигурирования Postgres Pro внимательно прочитайте Раздел 19.10. Перед написанием `ldap2pg.yaml` стоит надлежащим образом настроить Postgres Pro. Ниже приведёны шаги для настройки Postgres Pro через LDAP в рекомендуемом порядке:

1. Напишите поисковой запрос LDAP и проверьте его с помощью `ldapsearch(1)`. Это также позволяет проверить соединение с вашим каталогом LDAP.
2. В кластере Postgres Pro **вручную** создайте единственную роль, поместив пароль от неё в каталог LDAP.
3. Внесите изменения в файл `pg_hba.conf`, следуя Разделу 19.10, чтобы успешно авторизоваться с паролем из LDAP в созданной роли.

После настройки аутентификации LDAP в кластере Postgres Pro можно переходить к автоматизации процесса создания ролей из каталога LDAP с помощью ldap2pg:

1. Напишите простой файл `ldap2pg.yaml` с одним поисковым запросом LDAP, чтобы настроить параметры соединения ldap2pg с Postgres Pro и сервером LDAP. По умолчанию ldap2pg работает в пробном режиме, поэтому можно без опасений циклично запускать ldap2pg, пока расширение не будет настроено корректно.
2. Затем дополните файл `ldap2pg.yaml` в соответствии с вашими потребностями, следуя Разделу «Справка по командной строке». Запустите ldap2pg в режиме действия и проверьте, что ldap2pg поддерживает созданную вами единственную тестовую роль и что через неё можно установить соединение с кластером.
3. Наконец определите, когда и как будет запускаться синхронизация: регулярно с помощью файла crontab/через задачу Ansible/вручную/ по-другому. Убедитесь, что ldap2pg запускается регулярно, целенаправленно и с уведомлениями.

Поиск в каталоге LDAP

Первый шаг — выполнить поиск в вашем сервере LDAP с помощью `ldapsearch(1)`, утилиты командной строки из дистрибутива OpenLDAP. Пример:

```
$ ldapsearch -H ldaps://ldap.ldap2pg.docker -U testsasl -W
Enter LDAP Password:
SASL/DIGEST-MD5 authentication started
SASL username: testsasl
SASL SSF: 128
SASL data security layer installed.
# extended LDIF
#
# LDAPv3
...
# search result
search: 4
result: 0 Success
```

```
# numResponses: 16
# numEntries: 15
$
```

Сохраните настройки в файле `ldaprc`:

```
LDAPURI ldaps://ldap.ldap2pg.docker
LDAPSASL_AUTHCID testsasl
```

И в окружении: `LDAPPASSWORD=secret`

Затем обновите `ldapsearch`, чтобы правильно сопоставить записи ролей на сервере LDAP:

```
$ ldapsearch -H ldaps://ldap.ldap2pg.docker -U testsasl -W -b
 cn=dba,ou=groups,dc=ldap,dc=ldap2pg,dc=docker '' member
...
# dba, groups, ldap.ldap2pg.docker
dn: cn=dba,ou=groups,dc=ldap,dc=ldap2pg,dc=docker
member: cn=Alan,ou=people,dc=ldap,dc=ldap2pg,dc=docker
member: cn=albert,ou=people,dc=ldap,dc=ldap2pg,dc=docker
member: cn=ALICE,ou=people,dc=ldap,dc=ldap2pg,dc=docker

# search result
search: 4
result: 0 Success

...
$
```

Теперь перенесите запрос в `ldap2pg.yaml` и установите сопоставление ролей, чтобы создавать роли из каждого значения каждой записи, которую возвращает поисковой запрос LDAP:

```
- ldapsearch:
    base: cn=dba,ou=groups,dc=ldap,dc=ldap2pg,dc=docker
  role:
    name: '{member.cn}'
    options: LOGIN SUPERUSER
```

Выполните проверку:

```
$ ldap2pg
...
Querying LDAP cn=dba,ou=groups,dc=ldap,dc=ldap2pg,dc=docker...
Would create alan.
Would create albert.
Would update options of alice.
...
Comparison complete.
$
```

За подробностями о том, как управлять созданием ролей из записи LDAP, обратитесь к разделу Конфигурация. Когда результат сравнения будет удовлетворительным, перейдите в режим действия с помощью `--real`.

Использование отказоустойчивости LDAP

Встроенный функционал ldap2pg включает поддержку отказоустойчивости LDAP точно так же, как расширение поддерживает любого клиента OpenLDAP. Чтобы указать все серверы, перечислите их URI через пробел.

```
$ LDAPURI="ldaps://ldap1 ldaps://ldap2" ldap2pg
```

За подробностями обратитесь к ldap.conf(5).

Запуск не от имени суперпользователя

Поскольку у ролей в Postgres Pro> есть атрибут `CREATEROLE`, управлять ролями можно без прав суперпользователя. С точки зрения безопасности лучше управлять ролями без прав суперпользователя.

> **Предупреждение**
>
> До версии Postgres Pro 15 пользователь с правом `CREATEROLE` почти приравнивался к суперпользователю. Такой пользователь мог предоставить себе самому практически любое право. Поэтому ldap2pg поддерживает запуск не от имени суперпользователя только с версиями Postgres Pro 16 и выше.

ldap2pg поддерживает такой сценарий. Однако стоит внимательно относиться к ограничениям. Назовем роль не суперпользователя, которая даёт возможность создавать другие роли, `creator`.

- Невозможно управлять такими атрибутами ролей, как `SUPERUSER`, `BYPASSRLS` и `REPLICATION`. Поэтому обнаружить ложных суперпользователей не получится.
- Убедитесь, что `creator` может отзывать все права, предоставляемые управляемым пользователям.

Удаление всех ролей

Если в какой-то момент потребуется удалить все роли в кластере Postgres Pro, использовать расширение ldap2pg будет эффективно. Для удаления ролей явно определите пустые правила (`rules`).

```
$ echo '{version: 6, rules: []}' | ldap2pg --config -
...
Empty synchronization map. All roles will be dropped!
...
```

В этом примере применяется чёрный список по умолчанию. ldap2pg никогда не удаляет свою роль, которую использует для установки соединения.

ldap2pg как Docker-контейнер

Предполагается, что это не первая встреча пользователя с Docker, поэтому в этом разделе настройки описаны не будут.

Чтобы запустить контейнер, используйте следующую команду:

```
$ docker run --rm dalibo/ldap2pg --help
```

В образе Docker для ldap2pg используются те же конфигурационные параметры, которые описаны в разделах Интерфейс командной строки и `ldap2pg.yml`. Можно подключить файл конфигурации `ldap2pg.yml`.

```
$ docker run --rm -v ${PWD}/ldap2pg.yml:/workspace/ldap2pg.yml dalibo/ldap2pg
```

Можно также экспортировать некоторые переменные окружения с помощью параметра `-e`:

```
$ docker run --rm -v ${PWD}/ldap2pg.yml:/workspace/ldap2pg.yml -e
 PGDSN=postgres://postgres@localhost:5432/ -e LDAPURI=ldaps://localhost -e
 LDAPBINDDN=cn=you,dc=entreprise,dc=fr -e LDAPPASSWORD=pasglop dalibo/ldap2pg
```

Убедитесь, что контейнер может разрешить имя компьютера, на который даётся указание. При использовании внутренних разрешений имени не забудьте добавить к вашей команде параметр `–dns=`, который указывает на ваш внутренний DNS-сервер. За подробностями обратитесь *сюда*.

Пользовательский ACL

В ldap2pg встроены списки ACL для общих объектов, таких как `DATABASE`, `SCHEMA`, `TABLE`, `FUNCTION` и т. д. В Postgres Pro есть много других объектов, таких как `FOREIGN DATA WRAPPER`, `FOREIGN SERVER`, `FOREIGN TABLE`, `TYPE` и т. д. В такой ситуации может возникнуть потребность в управлении пользовательским ACL. Для этого можно написать пользовательский ACL.

Примечание

Чтобы написать пользовательский ACL, требуются навыки продвинутого пользователя. Необходимо разбираться в списках ACL Postgres Pro и конфигурации ldap2pg. Перед началом прочитайте разделы Управление правами и Справка по файлу `ldap2pg.yml`, убедитесь, что вы хорошо понимаете документацию по ACL Postgres Pro и что права успешно синхронизированы с ldap2pg.

Определите пользовательский ACL в YAML-файле конфигурации.

Вариант использования

Предположим, в вашей базе данных есть пользовательский тип `enum` `myenum`.

```
CREATE TYPE public.myenum AS ENUM ('toto', 'titi', 'tata');
```

Нам нужно получить доступ к управлению правами для этого объекта, в конечном итоге для других типов, через пользовательский ACL.

Определение имени

Назовите ваш ACL по ключевому слову в операторе `GRANT` или `REVOKE`. Из `GRANT USAGE ON TYPE mytype TO myrole` определите имя для `TYPE` вашего ACL.

```
acls:
  TYPE:
    ...
```

Область действия

Postgres Pro определяет типы пользователей в рамках схемы. Поскольку схема будет чётко задана, ограничим действие ACL базой данных.

```
acls:
  TYPE:
    scope: database
```

Предоставление и отзыв прав

Написать операторы `GRANT` и `REVOKE` легче всего. За подробностями о формате запроса обратитесь к описанию предоставления прав. Используем поле `object` в параметре grant, чтобы поместить туда имя типа. Схема public жёстко задана, о чём было сказано выше.

```
acls:
  TYPE:
    scope: database
    grant: GRANT <privilege> ON TYPE public.<object> TO <grantee>;
    revoke: REVOKE <privilege> ON TYPE public.<object> FROM <grantee>;
```

Проверка

Запрос на проверку — это самая сложная часть. Нужно уметь работать со встроенным системным типом Postgres Pro `aclitem` и встроенными системными функциями `aclexplode` и `acldefault`. Сигнатура запроса на проверку зависит от области действия ACL.

Для области действия `instance`:

- `type`: строка, описывающая тип права как ключевое слово SQL.
- `object`: строка, описывающая имя объекта как SQL-идентификатор.

- `grantee`: строка, описывающая имя роли как SQL-идентификатор.

Для области действия `database`:

- `type`: строка, описывающая тип права как ключевое слово SQL.
- `object`: строка, описывающая имя объекта как SQL-идентификатор.
- `grantee`: строка, описывающая имя роли как SQL-идентификатор.
- `partial`: логическое значение, указывающее, является ли предоставление права частичным.

Параметр `partial` указывает ldap2pg, что нужно снова предоставить права `ALL ... IN SCHEMA`. Поскольку наш ACL обрабатывает объекты по одному, значение `partial` всегда будет `false`.

ldap2pg отправляет один параметр в запросе на проверку — актуальный список типов прав, управляемых конфигурацией. Список представляет собой массив текста. Ожидается, что в результате запроса права, не включённые в список, не попадут в выборку.

Для списка ACL `TYPE` проверим права для доступа к системному каталогу `pg_type`.

```
acls:
  TYPE:
    scope: database
    grant: GRANT <privilege> ON <acl> public.<object> TO <grantee>;
    revoke: REVOKE <privilege> ON <acl> public.<object> FROM <grantee>;
    inspect: |
      WITH grants AS (
        SELECT typname,
               (aclexplode(COALESCE(typacl, acldefault('T', typowner)))).privilege_type
 AS priv,
               (aclexplode(COALESCE(typacl, acldefault('T',
 typowner)))).grantee::regrole::text AS grantee
          FROM pg_catalog.pg_type
        WHERE typnamespace::regnamespace = 'public'::regnamespace
          AND typtype <> 'b'  -- exclude base type.
      )
      SELECT grants.priv AS "privilege",
             grants.typname AS "object",
             CASE grants.grantee WHEN '-' THEN 'public' ELSE grants.grantee END AS
 grantee,
             FALSE AS partial
        FROM grants
      WHERE "priv" = ANY ($1)
      ORDER BY 2, 3, 1
      ;
```

Видно, что это не простой запрос. Он поддерживается в версии Postgres Pro 17.

Использование

Теперь можно переходить к использованию вашего пользовательского ACL в профиле. Ссылайтесь на объект в профиле, а не в правиле предоставлении права.

```
privileges:
  custom:
  - type: USAGE
    on: TYPE
    object: myenum

rules:
- roles:
    names:
    - alice
```

```
  grant:
    privilege: custom
    role: alice
```

В профиле необходимо дать ссылку на все типы вручную. При запуске ldap2pg изменения должны быть внесены в вашу базу данных:

```
$ ldap2pg
...
16:52:02 CHANGE Would Revoke privileges.                         grant="USAGE ON TYPE
 myenum TO public" database=db0
16:52:02 CHANGE Would Grant privileges.                          grant="USAGE ON TYPE
 myenum TO alice" database=db0
16:52:02 INFO   Comparison complete.                             searches=0 roles=1
 queries=5 grants=1
16:52:02 INFO   Use --real option to apply changes.
16:52:02 INFO   Done.                                            elapsed=44.345229ms
 mempeak=1.6MiB ldap=0s inspect=28.992071ms sync=0s
```

На этом всё.

Отладка

Если возникла проблема, изолируйте её. Ограничьте конфигурацию вашим пользовательским ACL. Синхронизируйте единственную роль, предоставьте ей права. Не используйте поисковые запросы LDAP, используйте только статические правила. Синхронизируйте единственную базу данных. Включите вывод отладочных сообщений с помощью параметра `--verbose`.

ldap2pg просматривает по очереди базы данных, затем списки ACL. Для каждого ACL выводится следующее сообщение:

В первой строке с информацией о вашем ACL присутствует атрибут записей `acl=TYPE`.

```
    17:13:35 DEBUG  Inspecting grants.                                 acl=TYPE
 scope=database database=db0
```

Затем идут сообщения для проверки: запрос и аргументы. Для каждого предоставленного права появляется сообщение `Found grant (найденное предоставленное право)`..

```
17:13:35 DEBUG  Executing SQL query:
WITH grants AS (
...
WHERE "priv" = ANY ($1)
ORDER BY 2, 3, 1
;
 arg=[USAGE]
17:13:35 DEBUG  Found grant in Postgres instance.                grant="USAGE ON TYPE
 myenum TO public" database=db0
```

После этого ldap2pg расширяет предоставленные права, сгенерированные правилом. Для каждого сгенерированного предоставленного права выводится сообщение `Wants grant (требуется предоставить право )`..

```
17:13:35 DEBUG  Wants grant.                                     grant="USAGE ON TYPE
 myenum TO alice" database=db0
```

В конце ldap2pg выводит изменения, которые будут применены.

```
17:13:35 CHANGE Would Revoke privileges.                         grant="USAGE ON TYPE
 myenum TO public" database=db0
17:13:35 DEBUG  Would Execute SQL query:
```

```
REVOKE USAGE ON TYPE public."myenum" FROM "public";
17:13:35 CHANGE Would Grant privileges.                          grant="USAGE ON TYPE
 myenum TO alice" database=db0
17:13:35 DEBUG  Would Execute SQL query:
GRANT USAGE ON TYPE public."myenum" TO "alice";
```

По завершении операции ldap2pg выводит сообщении с заключением, даже если не требуется никаких изменений.

```
17:13:35 DEBUG  Privileges synchronized.                         acl=TYPE database=db0
```

Авторы

- *Этьен БЕРСАК (Étienne BERSAC)*
- *Пьер-Луи ГОНОН (Pierre-Louis GONON)*
- *Дамьен Казельс (Damien Cazeils)*
- *Гарольд ле КЛЕМАН де СЕН-МАРК (Harold le CLÉMENT de SAINT-MARCQ)*
- *Рэндольф Вурхис (Randolph Voorhies)*

pgbadger

pgbadger — инструмент для быстрого анализа журналов Postgres Pro с созданием подробных отчётов и графиков

Синтаксис

`pgbadger` [*параметр_подключения*...] [*параметр*...] [*файл_журнала*...]

Описание

pgbadger — это анализатор журналов Postgres Pro/PostgreSQL, который быстро строит подробные отчёты, обрабатывая файлы журналов сервера. pgbadger поставляется вместе с Postgres Pro как отдельный скрипт на языке Perl.

В качестве *файла_журнала* можно передать отдельный файл, список файлов, или команду, выдающую список файлов. Если передать «-», содержимое журнала будет считываться из стандартного ввода.

Анализатор запросов pgbadger может обрабатывать огромные файлы журналов и сжатые файлы. Он может автоматически определять формат файла журнала (syslog, stderr, csvlog или jsonlog), если файл достаточно большой. Поддерживаемые форматы сжатия: gzip, bzip2, lz4, xz, zip и zstd. Для формата xz у вас должна быть установлена версия xz выше 5.05, которая поддерживает параметр `--robot`. Чтобы pgbadger определял размер несжатого файла для формата lz4, файл должен быть сжат с использованием параметра `--content-size`.

pgbadger поддерживает любой формат префиксов строк журнала, который можно задать в параметре log_line_prefix в файле конфигурации `postgresql.conf`, при условии, что указаны как минимум `%t` и `%p`.

Также могут быть проанализированы файлы журналов pgbouncer.

Для ускорения разбора журнала можно использовать режимы многопоточной обработки: выделять несколько ядер для нескольких файлов журнала или несколько ядер для одного файла. Эти режимы также можно сочетать.

pgbadger также может анализировать удалённые файлы журналов, получаемые через SSH-соединение без пароля. Этот режим может использоваться со сжатыми файлами и даже поддерживает многопоточность с выделением нескольких ядер на файл.

Примеры отчётов можно найти на странице *https://pgbadger.darold.net/#reports*.

Ограничения

pgbadger в настоящее время имеет следующие ограничения:

- Многопоточная обработка не поддерживается при чтении сжатых файлов журналов и CSV-файлов, а также в Windows.
- Файлы журналов формата CSV не могут быть обработаны удалённо.
- Журналы csvlog не могут быть переданы из стандартного ввода.

Подготовка и настройка

pgbadger поставляется вместе с Postgres Pro Standard в виде отдельного пакета `pgbadger` (подробные инструкции по установке приведены в Главе 16). Установив pgbadger, настройте его, как описано далее.

Подготовка для обработки определённых форматов журнала (необязательно)

Если вы планируете анализировать файлы журналов CSV, установите Perl-модуль `Text::CSV_XS`.

Подготовка для экспорта статистики (необязательно)

Если вы хотите экспортировать статистику в формате JSON, установите Perl-модуль `JSON::XS`:

Чтобы установить этот дополнительный модуль:

- В системах на базе Debian запустите:

  ```
  sudo apt-get install libjson-xs-perl
  ```

- В RPM-системе запустите:

  ```
  sudo yum install perl-JSON-XS
  ```

Подготовка для обработки сжатых файлов журнала (необязательно)

По умолчанию pgbadger автоматически определяет формат сжатого файла журнала по расширению файла и соответствующим образом использует утилиты распаковки:

- zcat для `gz`
- bzcat для `bz2`
- lz4cat для `lz4`
- zstdcat для `zst`
- unzip или xz для `zip` или `xz`

Если необходимая утилита находится вне каталогов, указанных в `PATH`, задайте путь к ней в параметре командной строки `--zcat`. Например:

```
--zcat="/usr/local/bin/gunzip -c" или --zcat="/usr/local/bin/bzip2 -dc"
--zcat="C:\tools\unzip -p"
```

> **Примечание**
>
> Благодаря автоопределению формата сжатых файлов вы можете использовать разные форматы журналов gz, bz2, lz4, xz, zip и zstd. Однако если вы зададите своё значение `--zcat`, обрабатывать сжатые файлы разных форматов не получится.

Настройка сервера Postgres Pro

Установите определённые параметры конфигурации в файле `postgresql.conf`:

- Настройте журналирование SQL-запросов.

  Чтобы включить журналирование SQL-запросов и получать в статистике запросов собственно текст этих запросов, установите

  ```
  log_min_duration_statement = 0
  ```

  На нагруженном сервере вы можете задать ненулевое значение, чтобы в журнал вносились только запросы с большей продолжительностью.

  Если вам нужна только информация о продолжительности и количестве запросов, а не подробные сведения о них, установите для log_min_duration_statement значение -1, при котором запись операторов по длительности отключается, и включите log_duration.

  При включении `log_min_duration_statement` добавятся отчёты о самых медленных запросах и запросах, которые заняли больше всего времени. Обратите внимание: если вы установите для log_statement значение `all`, параметр `log_min_duration_statement` не будет работать.

Предупреждение

Не устанавливайте для `log_min_duration_statement` неположительные значения одновременно с включением `log_duration` и `log_statement`, так как в результате нарушится подсчёт запросов и значительно увеличится размер журнала. Всегда предпочтительнее устанавливать `log_min_duration_statement`.

- Установите префикс строк журнала в log_line_prefix.

  Он должен включать как минимум спецпоследовательность времени (`%t`, `%m` или `%n`) и спецпоследовательность, идентифицирующую процесс (`%p` или `%c`). Например, для журналов stderr параметр должен содержать как минимум

  ```
  log_line_prefix = '%t [%p]: '
  ```

  В префикс строк журнала также можно добавить имя пользователя, базы данных, приложения и IP-адрес клиента. Например

  для журналов stderr:

  ```
  log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '
  ```

  или

  ```
  log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
  ```

  и для журналов syslog:

  ```
  log_line_prefix = 'user=%u,db=%d,app=%a,client=%h '
  ```

  или

  ```
  log_line_prefix = 'db=%d,user=%u,app=%a,client=%h '
  ```

- Чтобы получить больше информации из файлов журналов, задайте следующие параметры конфигурации:

  ```
  log_checkpoints = on
  log_connections = on
  log_disconnections = on
  log_lock_waits = on
  log_temp_files = 0
  log_autovacuum_min_duration = 0
  log_error_verbosity = default
  ```

  Чтобы этот набор параметров работал, не включайте log_statement, так как pgbadger не сможет разобрать получившийся журнал.

- Установите язык, на котором сервер будет выводить сообщения; сообщения должны быть на английском языке с поддержкой локали или без неё:

  ```
  lc_messages='C'
  ```

  или

  ```
  lc_messages='en_US.UTF-8'
  ```

  Локали других языков, например `ru_RU.utf8`, не поддерживаются.

Использование

Ниже приведены простые примеры, иллюстрирующие различные аспекты использования pgbadger.

```
pgbadger /var/lib/pgpro/std-17/data/log/postgresql-2022-01-14_000000.log
```

```
pgbadger /var/lib/pgpro/std-17/data/log/postgres.log.2.gz /var/lib/pgpro/std-17/data/
log/postgres.log.1.gz /var/lib/pgpro/std-17/data/log/postgresql-2022-01-14_000000.log
pgbadger /var/lib/pgpro/std-17/data/log/postgresql/postgresql-2022-01-*
pgbadger --exclude-query="^(COPY|COMMIT)" /var/lib/pgpro/std-17/data/log/
postgresql-2022-01-14_000000.log
pgbadger -b "2022-01-25 10:56:11" -e "2022-01-25 10:59:11" /var/lib/pgpro/std-17/data/
log/postgresql-2022-01-25-0000.log
cat /var/lib/pgpro/std-17/data/log/postgresql-2022-01-14_000000.log | pgbadger -
# Префикс строк журнала, выводимого в stderr
pgbadger --prefix '%t [%p]: user=%u,db=%d,client=%h' /var/lib/pgpro/std-17/data/log/
postgresql-2022-08-21*
pgbadger --prefix '%m %u@%d %p %r %a : ' /var/lib/pgpro/std-17/data/log/
postgresql-2022-08-21-0000.log
# Префикс строк журнала, выводимого в syslog
pgbadger --prefix 'user=%u,db=%d,client=%h,appname=%a' /var/lib/pgpro/std-17/data/log/
postgresql-2022-08-21*
# Использовать 8 ядер для ускорения обработки 10-гигабайтного файла
pgbadger -j 8 /var/lib/pgpro/std-17/data/log/postgresql-2022-08-21-0000.log
# Задание cron, которое еженедельно формирует отчёт об ошибке
30 23 * * 1 /usr/bin/pgbadger -q -w /var/lib/pgpro/std-17/data/log/
postgresql-2022-01*.log -o /var/www/pg_reports/pg_errors.html
```

Указание удалённых файлов журнала

Удалённые файлы журналов для анализа задаются посредством URI. Поддерживаемые протоколы: HTTP[S] и [S]FTP. Для загрузки файлов будет использоваться команда `curl`, и разбираться они будут сразу во время загрузки. Также поддерживается протокол SSH, в этом случае для загрузки файлов применяется команда `ssh`, как и при использовании параметра `--remote-host`.

Для указания удалённых файлов журналов вы можете использовать такие URI:

```
pgbadger http://172.12.110.1//var/lib/pgpro/std-17/data/log/
postgresql-2022-01-14_000000.log
pgbadger ftp://username@172.12.110.14/postgresql-2022-01-14_000000.log
pgbadger ssh://username@172.12.110.14:2222//var/lib/pgpro/std-17/data/log/
postgresql-2022-01-14_000000.log*
```

Также возможно обрабатывать одновременно и локальные, и удалённые файлы журналов Postgres Pro (в том числе, журнал pgbouncer):

```
pgbadger /var/lib/pgpro/std-17/data/log/postgresql-2022-01-14_000000.log ssh://
username@172.12.110.14/pgbouncer.log
```

Параллельная обработка

Чтобы включить параллельную обработку, укажите параметр `-j N`, где *N* — количество используемых ядер.

Параллельная обработка в pgbadger выполняется по следующему алгоритму:

```
Для каждого файла журнала
   размер порции = int(размер файла / N)
   определить начальное/конечное смещение этих порций
   породить N процессов и искать начальное смещение каждого фрагмента
     завершать каждый процесс разбора по достижении конечного смещения его фрагмента
     записать статистику, собранную процессом, во временный бинарный файл
   дождаться завершения всех дочерних процессов
Затем все сгенерированные бинарные временные файлы считываются и загружаются в
память для формирования выходного html.
```

Когда применяется этот метод, pgbadger может усечь или пропустить в начале/конце порций максимум *N* запросов на файл журнала, что вряд ли имеет значение, если в вашем файле журнала миллионы запросов. Вероятность того, что потеряется именно интересующий вас запрос, близка к нулю, так что потерю такого объёма можно считать приемлемой. В большинстве случаев запрос учитывается дважды, но в усечённом виде.

Если у вас много небольших файлов журналов и много ядер, для ускорения лучше выделить одно ядро на один файл журнала. Чтобы включить такое поведение, укажите параметр `-J` *`N`*. Таким образом, вы можете быть уверены, что не потеряете ни одного запроса в отчётах. Например, если обрабатывать 200 файлов по 10 Мбайт каждый, параметр `-J` начинает показывать свою эффективность с 8 ядрами.

Ниже представлены результаты теста, полученные при обработке на сервере с 8 ядрами одного файла размером 9,5 гигабайт.

```
 Параметр | 1 ядро   |  2 ядра |  4 ядра | 8 ядер
----------+----------+---------+---------+--------
   -j     | 1h41m18 |  50m25  |  25m39  | 15m58
   -J     | 1h41m18 |  54m28  |  41m16  | 34m45
```

С 200 файлами журналов по 10 мегабайт каждый, что в сумме составляет 2 гигабайта, результаты немного отличаются:

```
 Параметр |  1 ядро | 2 ядра | 4 ядра | 8 ядер
----------+---------+--------+--------+--------
   -j     |  20m15  |  9m56  |  5m20  | 4m20
   -J     |  20m15  |  9m49  |  5m00  | 2m40
```

Поэтому рекомендуется использовать `-j`, кроме случаев, когда нужно обработать сотни небольших файлов и имеется минимум 8 ядер.

Важно

Во время параллельной обработки pgbadger создаёт множество временных файлов с именами `tmp_pgbadgerXXXX.bin` в каталоге `/tmp` и удаляет их в конце.

Создание инкрементальных отчётов

Следующий пример задания cron демонстрирует создание отдельных еженедельных отчётов, в предположении, что осуществляется еженедельная ротация файлов журналов:

```
0 4 * * 1 /usr/bin/pgbadger -q `find /var/lib/pgpro/std-17/data/log/ -mtime -7 -name
 "postgresql.log*"` -o /var/www/pg_reports/pg_errors-`date +\%F`.html -l /var/reports/
pgbadger_incremental_file.dat
```

Вы можете включить режим автоматического создания инкрементальных отчётов pgbadger, воспользовавшись параметром `-I`/`--incremental`. В этом режиме pgbadger создаёт один отчёт в день и один накопительный отчёт в неделю. Результаты обработки журналов сначала сохраняются в двоичном формате, в каталоге, задаваемом параметром `-O`/`--outdir`, а затем строятся ежедневные и еженедельные HTML-отчёты с индексным файлом. В индексном файле отображаются раскрывающиеся меню по неделям со ссылкой на недельный отчёт и ссылками на отчёты по дням недели. Например, вы можете запустить pgbadger с ежедневной ротацией файла следующим образом:

```
0 4 * * * /usr/bin/pgbadger -I -q /var/lib/pgpro/std-17/data/log/postgresql/
postgresql.log.1 -O /var/www/pg_reports/
```

Вы получите все ежедневные и еженедельные отчёты. В этом режиме pgbadger автоматически создаёт инкрементальный файл в выходном каталоге, поэтому задавать параметр `-l` не нужно, если вы не хотите изменить путь к этому файлу. Это означает, что вы можете запускать pgbadger в этом режиме каждый день с файлом журнала, который заменяется каждую неделю, и при этом записи журнала не будут учитываться дважды. Для экономии места на диске вы можете использовать параметр командной строки `-X`/`--extra-files`, чтобы pgbadger записывал ресурсы CSS и JavaScript в отдельные файлы в выходном каталоге. Тогда эти ресурсы будут подключены с использованием тегов script и link.

В инкрементальном режиме вы также можете указать, сколько недель должно сохраняться в отчёте, воспользовавшись параметром `-O`/`--retention`:

```
/usr/bin/pgbadger --retention 2 -I -q /var/lib/pgpro/std-17/data/log/postgresql/
postgresql.log.1 -O /var/www/pg_reports/
```

Если запуск pg_dump запланирован на 23:00 и 13:00 каждый день, вы можете исключить эти периоды из отчёта следующим образом:

```
pgbadger --exclude-time "2013-09-.* (23|13):.*" postgresql.log
```

Это поможет избежать появления операторов `COPY`, которые выдаёт pg_dump, в верху списка самых медленных запросов. Вы также можете использовать `--exclude-appname "pg_dump"`, чтобы решить эту проблему более простым способом.

Перестроение отчётов

Чтобы обновить все HTML-отчёты после исправления или расширения функциональности отчётов pgbadger, вы можете перестроить инкрементальные отчёты, если двоичный файл данных сохранился. Для этого выполните:

```
rm /path/to/reports/*.js
rm /path/to/reports/*.css
pgbadger -X -I -O /var/www/pg_reports/ --rebuild
```

При этом также будут обновлены все файлы ресурсов (JavaScript и CSS). Добавьте параметр `-E`/`--explode`, если он использовался при создании отчётов.

Создание месячных отчётов

По умолчанию в инкрементальном режиме pgbadger создаются только ежедневные и еженедельные отчёты. Чтобы получить месячный накопительный отчёт, нужно отдельно запустить pgbadger с соответствующим параметром. Например, чтобы создать отчёт за август 2021 года, запустите:

```
pgbadger -X --month-report 2021-08 /var/www/pg_reports/
```

В результате в представление календаря с инкрементальными отчётами будет добавлена связанная с названием месяца ссылка на этот месячный отчёт. Построение отчёта за месяц можно запускать каждый день, и этот отчёт будут перестраиваться заново. По умолчанию такие отчёты не строятся, так как они могут формироваться слишком долго. Как и при перестроении отчётов, если отчёты создавались с параметром `-E`/`--explode` (включающим разбиение отчёта по базам данных), его необходимо указать и при построении месячного отчёта:

```
pgbadger -E -X --month-report 2021-08 /var/www/pg_reports/
```

Выбор формата файла отчёта

Формат файла отчёта pgbadger определяется расширением файла, переданного параметру `-o`/`--outfile`.

Для создания настраиваемых инкрементальных и накопительных отчётов используйте двоичный формат (`-o *.bin`).

Например, чтобы ежечасно обновлять отчёт pgbadger из ежедневного файла журнала, вы можете каждый час запускать следующие команды:

```
# Создать файлы инкрементальных данных в двоичном формате
pgbadger --last-parsed .pgbadger_last_state_file -o sunday/hourX.bin /var/lib/pgpro/
std-17/data/log/postgresql-Sun.log
# Создать новый HTML-отчёт из сгенерированного двоичного файла
pgbadger sunday/*.bin
```

В качестве другого примера предположим, что у вас каждый час создаётся новый файл журнала. Чтобы отчёты перестраивались при каждой ротации журнала, выполните:

```
pgbadger -o day1/hour01.bin /var/lib/pgpro/std-17/data/log/postgresql-2022-01-23_10.log
pgbadger -o day1/hour02.bin /var/lib/pgpro/std-17/data/log/postgresql-2022-01-23_11.log
pgbadger -o day1/hour03.bin /var/lib/pgpro/std-17/data/log/postgresql-2022-01-23_12.log
...
```

А чтобы обновлять HTML-отчёт, например, каждый раз после создания нового двоичного файла, просто запустите:

```
pgbadger -o day1_report.html day1/*.bin
```

Настройте команды в соответствии с вашими потребностями.

Применяйте формат JSON (`-o *.json`) для обмена данными с другими языками и для облегчения использования вывода pgbadger другими средствами мониторинга, такими как Cacti или Graphite.

Вы можете использовать и другие форматы вывода, соответствующие вашим потребностям. Например, следующая команда создаст XML-файл сеансов *Tsung* только для запросов `SELECT`:

```
  pgbadger -S -o sessions.tsung --prefix '%t [%p]: user=%u,db=%d ' /var/lib/pgpro/
std-17/data/log/postgresql-2022-01-14_000000.log
```

Коды выхода

pgbadger возвращает следующие коды выхода:

- `0`: успешное завершение работы
- `1`: работа прервана из-за ошибки
- `2`: работа была прервана, например с помощью `Ctrl+C`
- `3`: файл PID уже существует или не может быть создан
- `4`: в командной строке не был указан файл журнала

Параметры

В этом разделе описываются параметры командной строки pgbadger.

`-a` *`минуты`*
`--average` *`минуты`*

Указывает длительность интервала (в минутах) для построения графика усреднённых значений для запросов и подключений.

По умолчанию: 5.

`-A` *`минуты`*
`--histo-average` *`минуты`*

Указывает длительность интервала (в минутах) для построения гистограмм для запросов.

По умолчанию: 60.

`-b` *`дата_время`*
`--begin` *`дата_время`*

Дата и время или только время начала диапазона анализируемых данных в журналах.

`-c` *`сервер`*
`--dbclient` *`сервер`*

Обрабатывать только записи, относящиеся к указанному клиентскому компьютеру.

`-C`
`--nocomment`

Удалять из запросов комментарии /* ... */.

`-d` *`имя`*
`--dbname` *`имя`*

Обрабатывать только записи, относящиеся к указанной базе данных.

`-D`
`--dns-resolv`

Заменять IP-адреса клиентов их DNS-именами.

Предупреждение

Это может значительно замедлить работу pgbadger.

`-e` *`дата_время`*
`--end` *`дата_время`*

Дата и время или только время конца диапазона анализируемых данных в журналах.

`-E`
`--explode`

Строить отдельные отчёты для разных баз данных. Глобальная информация, не связанная с какой-либо базой данных, добавляется в отчёт по базе данных `postgres`.

`-f` *`тип_журнала`*
`--format` *`тип_журнала`*

Указывает тип журнала.

Возможные значения: `syslog`, `syslog2`, `stderr`, `jsonlog`, `csv`, `pgbouncer`, `logplex`, `rds` и `redshift`. Следует использовать, когда pgbadger не может определить формат журнала.

`-G`
`--nograph`

Отключает графики в выводе HTML.

`-h`
`--help`

Вывести подробную информацию о параметрах pbadger и завершиться.

`-H` *`путь`*
`--html-outdir` *`путь`*

Указывает путь к каталогу, в который будет записываться HTML-отчёт в инкрементальном режиме. Обратите внимание, что двоичные файлы остаются в каталоге, указанном в `-O`/`--outdir`.

`-i` *`имя`*
`--ident` *`имя`*

Имя программы, по которому можно выделить сообщения Postgres Pro в журналах syslog.

По умолчанию: `postgres`.

`-I`
`--incremental`

Использовать инкрементальный режим, в котором отчёты будут генерироваться по дням в отдельном каталоге, указанном параметром `-O`/`--outdir`.

`-j` *`число`*
`--jobs` *`число`*

Задаёт количество одновременно выполняемых заданий обработки журнала. Журналы csvlog всегда обрабатываются в однопоточном режиме.

По умолчанию: 1.

`-J` *`число`*
`--Jobs` *`число`*

Указывает количество файлов журнала, которые могут разбираться параллельно.

По умолчанию: 1.

`-l` *`имя_файла`*
`--last-parsed` *`имя_файла`*

Указывает файл для сохранения последней разобранной строки и её даты в целях обеспечения инкрементальной обработки журналов. Полезно использовать для просмотра ошибок с момента последнего запуска или для получения одного отчёта в день с еженедельной ротацией журнала.

`-L` *`имя_файла`*
`--logfile-list` *`имя_файла`*

Указывает файл, содержащий список файлов журналов для анализа.

`-m` *`размер`*
`--maxlength` *`размер`*

Задаёт максимальную длину запроса в отчётах (в символах). Более длинные запросы будут усечены.

По умолчанию: 100000.

`-M`
`--no-multiline`

Отключает составление многострочных операторов. Таким образом отчёты становятся чище, особенно в случае ошибок, когда выводится много лишнего.

`-N` *`имя`*
`--appname` *`имя`*

Обрабатывать только записи журнала, относящиеся к указанному приложению.

`-o` *`имя_файла`*
`--outfile` *`имя_файла`*

Указывает имя файла для вывода и определяет формат файла отчёта: `out.html`, `out.txt`, `out.bin` или `out.json`. Может использоваться несколько раз для получения разных форматов. Для вывода `json` убедитесь, что установлен Perl-модуль `JSON::XS`. Если в качестве имени файла указать «-», отчёт будет записан в стандартный вывод.

По умолчанию: `out.html`, `out.txt`, `out.bin`, `out.json` или `out.tsung`,

для соответствующего формата вывода.

`-O` *`путь`*
`--outdir` *`путь`*

Указывает каталог, в который будут записаны выходные файлы.

`-p` *`строка`*
`--prefix` *`строка`*

Указывает значение настраиваемого параметра log_line_prefix, заданное в файле `postgresql.conf`. Полезен в случаях, когда вы используете префикс строк, отличный от стандартных префиксов `log_line_prefix`, например, когда префикс включает дополнительные переменные, такие как IP-адрес клиента или имя приложения. Строка должна содержать спецпоследовательности для указания времени (%t, %m или %n) и процессов (%p или %c).

`-P`
`--no-prettify`

Отключает улучшение визуального представления кода SQL-запросов.

`-q`
`--quiet`

Полностью отключает вывод в stdout, не выводится даже индикатор выполнения.

`-Q`
`--query-numbering`

Добавляет нумерацию запросов в вывод, когда используется ключ `--dump-all-queries` или `--normalized-only`.

`-r` *`адрес`*
`--remote-host` *`адрес`*

Указывает компьютер, на котором выполнится команда `cat` для удалённого файла; её вывод будет разобран локально.

`-R` *`число`*
`--retention` *`число`*

Задаёт интервал (в неделях), в течение которого отчёты должны сохраняться в каталоге вывода в инкрементальном режиме. Каталоги предшествующих недель и дней удаляются автоматически.

По умолчанию: 0 (отчёты не удаляются).

`-s` *`число`*
`--sample` *`число`*

Задаёт количество сохраняемых выборок запросов.

По умолчанию: 3.

`-S`
`--select-only`

Обрабатывать только запросы SELECT.

`-t` *`число`*
`--top` *`число`*

Указывает количество запросов для хранения/отображения.

По умолчанию: 20.

`-T` *`строка`*
`--title` *`строка`*

Указывает заголовок страницы отчёта в формате HTML.

`-u` *`имя_пользователя`*
`--dbuser` *`имя_пользователя`*

Обрабатывать только записи журнала, относящиеся к указанному пользователю.

`-U` *`имя_пользователя`*
`--exclude-user` *`имя_пользователя`*

Указывает имя пользователя для исключения относящихся к нему записей из отчёта.

`-v`
`--verbose`

Включает подробный режим или режим отладки.

По умолчанию: выкл.

`-V`
`--version`

Вывести версию pgbadger и завершиться.

`-w`
`--watch-mode`

Обрабатывать только ошибки, по аналогии с *Logwatch*.

`-W`
`--wide-char`

Кодировать HTML-вывод запросов в UTF-8, чтобы избежать сообщений Perl «Wide character in print» (Широкий символ в print).

`-x` *`формат`*
`--extension` *`формат`*

Задаёт формат вывода. Возможные значения: `text`, `html`, `bin` или `json`.

По умолчанию: `html`.

`-X`
`--extra-files`

В инкрементальном режиме записывать ресурсы CSS и JavaScript в каталог вывода как отдельные файлы.

`-z` *`команда`*
`--zcat` *`команда`*

Задаёт полную команду для запуска программы zcat. Используйте, если zcat, bzcat или unzip находится не в каталоге, включённом в `PATH`.

`-Z` *`+/-XX`*
`--timezone` *`+/-XX`*

Задаёт смещение часового пояса от GMT. Используйте для настройки даты/времени в графиках JavaScript. Значение может быть целым числом (например, 2) или числом с плавающей точкой (например, 2.5).

`--pie-limit` *`число`*

Определяет число, данные круговой диаграммы меньше которого будут заменены суммой.

`--exclude-query` *`регулярное_выражение`*

Задаёт регулярное выражение, при совпадении с которым запросы будут исключены из отчёта. Например: «^(VACUUM|COMMIT)». Можно использовать несколько раз.

`--exclude-file` *имя_файла*

Указывает путь к файлу, содержащему все регулярные выражения для исключения соответствующих им запросов из отчёта, по одному выражению в строке.

`--include-query` *регулярное_выражение*

Задаёт регулярное выражение для включения соответствующих ему запросов в отчёт. Можно использовать несколько раз. Например: «(tbl1|tbl2)».

`--include-file` *имя_файла*

Указывает путь к файлу, содержащему все регулярные выражения для включения соответствующих им запросов в отчёт, по одному выражению в строке.

`--disable-error`

Отключает формирование отчёта об ошибках.

`--disable-hourly`

Отключает формирование почасового отчёта.

`--disable-type`

Отключает формирование отчёта о запросах по типам, базам данных и пользователям.

`--disable-query`

Отключает формирование отчётов о запросах, например: самых медленных и наиболее частых, по пользователям, базам данных и т. д.

`--disable-session`

Отключает формирование отчёта о сеансах.

`--disable-connection`

Отключает формирование отчёта о соединениях.

`--disable-lock`

Отключает формирование отчёта о блокировках.

`--disable-temporary`

Отключает формирование отчёта о временных файлах.

`--disable-checkpoint`

Отключает формирование отчётов о контрольных точках/точках перезапуска.

`--disable-autovacuum`

Отключает формирование отчёта об автоочистке.

`--charset` *имя*

Указывает набор символов HTML, который будет использоваться.

По умолчанию: utf-8.

`--csv-separator` *char*

Указывает разделитель полей CSV.

По умолчанию: «,».

`--exclude-time` *регулярное_выражение*

Указывает регулярное выражения для исключения из отчёта записей с соответствующей ему меткой времени. Например: «2013-04-12 .*». Можно использовать несколько раз.

`--include-time` *регулярное_выражение*

Указывает регулярное выражение для включения в отчёт записей с соответствующей ему меткой времени. Например: «2013-04-12 .*». Можно использовать несколько раз.

`--exclude-db` *имя*

Указывает имя базы данных для исключения связанных с ней записей журнала из отчёта. Например: «outdated_db». Можно использовать несколько раз.

`--exclude-appname` *имя*

Задаёт имя приложения для исключения связанных с ним записей журнала из отчёта. Например: «pg_dump». Можно использовать многократно.

`--exclude-line` *регулярное_выражение*

Указывает регулярное выражение для исключения из отчёта записей журнала при совпадении со всей строкой. Можно использовать несколько раз.

`--exclude-client` *имя*

Указывает IP-адрес/имя клиента для исключения связанных с ним записей журнала из отчёта. Можно использовать несколько раз.

`--anonymize`

Анонимизировать все константы в запросах. Позволяет скрыть конфиденциальные данные.

`--noreport`

Отключает формирование отчётов в инкрементальном режиме.

`--log-duration`

Связывать записи журнала, сформированные в результате действия параметров `log_duration = on` и `log_statement = all`.

`--enable-checksum`

Добавить сумму MD5 под каждым запросом.

`--journalctl` *команда*

Указывает команду для генерирования информации, аналогичной той, что содержится в файле журнала Postgres Pro. Обычно это: `journalctl -u postgrespro-std-17`.

`--pid-dir` *путь*

Указывает путь к файлу с PID.

По умолчанию: `/tmp`.

`--pid-file` *имя_файла*

Указывает имя файла с PID для управления параллельным выполнением pgbadger.

По умолчанию: `pgbadger.pid`.

`--rebuild`

Перестроить все HTML-отчёты в каталогах инкрементального вывода, содержащих файлы двоичных данных.

`--pgbouncer-only`

Показывать в заголовке только связанные с pgbouncer меню.

`--start-monday`

Начинать календарные недели с понедельника (в инкрементальном режиме). По умолчанию они начинаются в воскресенье.

`--iso-week-number`

Начинать календарные недели в инкрементальном режиме с понедельника (по умолчанию недели начинаются с воскресенья), с нумерацией недель согласно ISO 8601: от 01 до 53, где первая неделя — это первая неделя года, в которой не менее 4 дней.

`--normalized-only`

Только выгрузить все нормализованные запросы в файл `out.txt`.

`--log-timezone` *`+/-XX`*

Задаёт смещение часового пояса для пересчёта значений даты/времени, прочитанных из файла журнала, перед анализом. Ненулевое смещение усложняет поиск соответствующих записей в журнале по дате/времени. Значение может быть целым числом (например, 2) или числом с плавающей точкой (например, 2.5).

`--prettify-json`

Улучшать визуальное представление JSON.

`--month-report` *`ГГГГ-ММ`*

Указывает месяц (ГГГГ-ММ), за который нужно построить сводный HTML-отчёт. Для построения отчёта требуются каталоги инкрементального вывода, содержащие все необходимые двоичные данные.

`--day-report` *`ГГГГ-ММ-ДД`*

Указывает день (ГГГГ-ММ-ДД) для создания HTML-отчёта. Для построения отчёта требуются каталоги инкрементального вывода, содержащие все необходимые двоичные данные.

`--noexplain`

Отключает обработку строк журнала, сформированных механизмом auto_explain.

`--command` *`команда`*

Задаёт команду, от которой будет получаться содержимое журнала через stdin. pgbadger откроет канал для получения её вывода и будет анализировать выводимые ею записи.

`--no-week`

Не строить еженедельные запросы в инкрементальном режиме. Это полезно, если такие отчёты строятся слишком долго.

`--explain-url` *`URL`*

Позволяет переопределить URL визуализатора вывода EXPLAIN ANALYZE.

По умолчанию: *`http://explain.depesz.com/`*

`--tempdir` *`путь`*

Указывает каталог для временных файлов.

По умолчанию: `File::Spec->tmpdir() || '/tmp'`.

`--no-process-info`

Отключает смену названия процесса pgbadger на другое, помогающее идентифицировать этот процесс. Полезно в системах, в которых не поддерживается изменение названия процесса.

`--dump-all-queries`

Выгрузить все запросы, найденные в файле журнала, в текстовый файл, подставляя значения привязываемых параметров в соответствующие позиции в запросах.

`--keep-comments`

Сохранять комментарии в нормализованных запросах. Это позволяет различать одинаковые нормализованные запросы.

`--no-progressbar`

Отключает вывод индикатора выполнения.

`--dump-raw-csv`

Разобрать журнал и выгрузить информацию в формате CSV без дальнейшей обработки и создания отчёта.

`--include-pid` *`PID`*

Сообщать только о событиях, связанных с PID сеанса (%p). Можно использовать несколько раз.

`--include-session` *`ID`*

Сообщать только о событиях, связанных с ID сеанса (%c). Можно использовать несколько раз.

`--histogram-query` *`VAL`*

Использовать пользовательские входящие для гистограммы времени запроса. Входящие по умолчанию в миллисекундах: 0, 1, 5, 10, 25, 50, 100, 500, 1000, 10000.

`--histogram-session` *`VAL`*

Использовать пользовательские входящие для гистограммы времени сеанса. Входящие по умолчанию в миллисекундах: 0, 500, 1000, 30000, 60000, 600000, 1800000, 3600000, 28800000.

Параметры подключения к удалённому журналу

pgbadger может анализировать удалённые файлы журналов, получаемые через SSH-соединение без пароля. IP-адрес или имя целевого компьютера задаётся в параметре `-r`/`--remote-host`. Другие свойства SSH-подключения определяются следующими параметрами:

`--ssh-program` *`ssh`*

Указывает путь к используемому SSH-клиенту.

По умолчанию: ssh.

`--ssh-port` *`порт`*

Указывает порт SSH для подключения.

По умолчанию: 22.

`--ssh-user` *`имя_пользователя`*

Указывает имя пользователя для подключения.

По умолчанию: имя пользователя, запускающего pgbadger.

`--ssh-identity` *`имя_файла`*

Указывает путь к файлу идентификации.

`--ssh-timeout` *секунды*

Задаёт тайм-аут в секундах на случай сбоя SSH-соединения.

По умолчанию: 10.

`--ssh-option` *параметры*

Задаёт список параметров для SSH-соединения. Следующие параметры используются всегда:

`-o ConnectTimeout=$ssh_timeout -o PreferredAuthentications=hostbased,publickey`

`-o PreferredAuthentications=hostbased,publickey`

Автор

Жиль Даролд `<gilles@darold.net>`

pg_repack

pg_repack — утилита и расширение Postgres Pro Standard для реорганизации таблиц

Синтаксис

`pg_repack` [*параметр*...] [*имя_бд*]

Описание

Модуль *pg_repack* — это расширение Postgres Pro Standard, которое позволяет ликвидировать раздувание таблиц и индексов и может дополнительно восстанавливать физический порядок кластеризованных индексов. В отличие от CLUSTER и `VACUUM FULL`, оно выполняет эти операции «на ходу», обходясь без исключительных блокировок таблиц в ходе их обработки. К тому же pg_repack действует эффективно, демонстрируя производительность, сравнимую с непосредственным использованием `CLUSTER`.

pg_repack — это альтернативная ветвь развития проекта *https://github.com/reorg/pg_reorg*.

Вы можете выбрать один из следующих способов реорганизации данных:

- Неблокирующая кластеризация (`CLUSTER`) (с упорядочиванием по кластеризующему индексу)
- Упорядочивание по указанным столбцам
- Неблокирующая полная очистка (`VACUUM FULL`) (только упаковка строк)
- Перестроение или перемещение только индексов таблицы

> **Примечание**
>
> Использовать эту утилиту могут только суперпользователи.

> **Примечание**
>
> В целевой таблице должен быть первичный ключ (PRIMARY KEY) или как минимум уникальный индекс по столбцу NOT NULL.

Установка

Для систем Linux pg_repack поставляется с Postgres Pro Standard в виде отдельного пакета `pg-repack-std-17`, который требует, чтобы был установлен пакет `postgrespro-std-17-server` со всеми зависимостями. Список предоставляемых пакетов и подробное описание установки можно найти в Главе 16. В системах Windows pg_repack устанавливается автоматически в составе Postgres Pro.

Установив pg_repack, загрузите расширение `pg_repack` в базу данных, которую вы хотите обрабатывать, следующим образом:

```
$ psql -c "CREATE EXTENSION pg_repack" -d ваша_база
```

Впоследствии вы можете удалить `pg_repack` из инсталляции Postgres Pro, выполнив `DROP EXTENSION pg_repack`.

Чтобы обновить ранее установленную версию pg_repack, просто удалите из базы данных старую версию, как описано выше, и установите новую.

Параметры

Параметры реорганизации

`-a`
`--all`

Попытаться перепаковать все базы данных в кластере. Базы данных, в которых расширение `pg_repack` не установлено, будут пропущены.

`-t` *`таблица`*
`--table=`*`таблица`*

Реорганизовать только указанную таблицу (таблицы). Реорганизовать несколько таблиц можно, добавив несколько ключей `-t`. По умолчанию реорганизуются все подходящие таблицы в целевых базах данных.

`-I` *`таблица`*
`--parent-table=`*`таблица`*

Реорганизовать как указанную таблицу (таблицы), так и дочерние. Реорганизовать несколько иерархий таблиц можно, добавив несколько ключей `-I`.

`-c` *`схема`*
`--schema=`*`схема`*

Перепаковать таблицы только в указанной схеме (схемах). Перепаковать несколько схем можно, добавив несколько ключей `-c`. Можно использовать в сочетании с `--tablespace` для перемещения таблиц в другое табличное пространство.

`-o` *`столбец`*`[, ...]`
`--order-by=`*`столбец`*`[, ...]`

Выполнить неблокирующую кластеризацию (`CLUSTER`), упорядочивая данные по указанным столбцам.

`-n`
`--no-order`

Выполнить неблокирующую полную очистку (`VACUUM FULL`). Начиная с версии 1.2, это производится по умолчанию для некластеризованных таблиц.

`-N`
`--dry-run`

Только показать, какие таблицы будут перепакованы, и завершиться.

`-j` *`num_jobs`*
`--jobs=`*`num_jobs`*

Установить заданное количество дополнительных соединений к Postgres Pro и использовать эти дополнительные подключения для перестраивания индексов таблиц в параллельном режиме. Параллельное перестроение индексов поддерживается только при реорганизации таблиц полностью, без ключей `--index` или `--only-indexes`. Если ваш сервер имеет несколько процессорных ядер и быструю дисковую подсистему, параллельный режим может быть полезен для ускорения pg_repack.

`-s` *`табличное_пространство`*
`--tablespace=`*`табличное_пространство`*

Перенести перепакованные таблицы в заданное табличное пространство: по сути это неблокирующая версия команды `ALTER TABLE ... SET TABLESPACE`. Индексы таблиц остаются в исходном табличном пространстве, если только дополнительно не указан ключ `--moveidx`.

`-S`
`--moveidx`

Также перенести индексы перепакованных таблиц в табличное пространство, заданное ключом `--tablespace`.

`-i` *индекс*
`--index=`*индекс*

Перепаковать только указанный индекс. Перепаковать несколько индексов можно, добавив несколько ключей `-i`. Можно использовать в сочетании с `--tablespace` для перемещения индексов в другое табличное пространство.

`-x`
`--only-indexes`

Перепаковать только индексы таблиц, заданных ключами `--table`.

`-T` *сек*
`--wait-timeout=`*сек*

Расширению pg_repack необходимо получить исключительную блокировку в конце реорганизации. Этот параметр определяет, сколько секунд pg_repack будет ждать получения этой блокировки. Если за это время блокировка не будет получена и параметр `--no-kill-backend` не указан, pg_repack принудительно отменит конфликтующие запросы. Если же вы используете Postgres Pro или PostgreSQL версии 8.4 и новее, pg_repack прибегнет к вызову `pg_terminate_backend()`, чтобы отключить все оставшиеся фоновые процессы, после того, как это время истечёт дважды. Значение по умолчанию: 60 сек.

`-D`
`--no-kill-backend`

Пропускать перепаковку таблицы вместо отмены конфликтующих запросов, если блокировка не может быть получена в течение времени, указанного в `--wait-timeout`. Значение по умолчанию — `false`.

`-Z`
`--no-analyze`

Не выполнять `ANALYZE` после полной реорганизации таблицы. В отсутствие этого ключа после реорганизации `ANALYZE` выполняется.

`-k`
`--no-superuser-check`

Пропускать проверку суперпользователя клиентом. Этот параметр полезен при использовании pg_repack на платформах, которые поддерживают запуск расширения не от имени суперпользователя.

`-C`
`--exclude-extension`

Пропускать таблицы, которые принадлежат указанному расширению (расширениям). Время планирования некоторых расширений может сильно зависеть от таких таблиц.

`--switch-threshold`

Переключать таблицы, когда в таблице журнала остаётся заданное число кортежей. Этот параметр можно использовать, чтобы иметь возможность нагнать таблицы с большим объёмом записи.

Параметры подключения

Параметры подключения к серверам. Параметр `--all` нельзя использовать вместе с `--dbname`, `--table` или `--parent-table`.

`-a`
`--all`

Реорганизовать все базы данных.

`[-d]` *`имя_бд`*
`[--dbname=]`*`имя_бд`*

Указывает имя базы данных для реорганизации. Если оно не указано, и параметр `-a` (или `--all`) не используется, то имя базы берётся из переменной окружения `PGDATABASE`. Если и эта переменная не задана, выбирается имя подключающегося пользователя.

`-h` *`сервер`*
`--host=`*`сервер`*

Указывает имя компьютера, на котором работает сервер. Если значение начинается с косой черты, оно определяет каталог Unix-сокета.

`-p` *`порт`*
`--port=`*`порт`*

Указывает TCP-порт или расширение файла локального Unix-сокета, через который сервер принимает подключения.

`-U` *`имя_пользователя`*
`--username=`*`имя_пользователя`*

Имя пользователя для подключения.

`-w`
`--no-password`

Не выдавать запрос на ввод пароля. Если сервер требует аутентификацию по паролю и пароль не доступен с помощью других средств, таких как файл `.pgpass`, попытка соединения не удастся. Этот параметр может быть полезен в пакетных заданиях и скриптах, где нет пользователя, который вводит пароль.

`-W`
`--password`

Принудительно запрашивать пароль перед подключением к базе данных.

Это несущественный параметр, так как pg_repack запрашивает пароль автоматически, если сервер проверяет подлинность по паролю. Однако чтобы понять это, pg_repack лишний раз подключается к серверу. Поэтому иногда имеет смысл ввести `-W`, чтобы исключить эту ненужную попытку подключения.

Общие параметры

`-e`
`--echo`

Выводить команды, которые pg_repack формирует и передаёт на сервер.

`-E` *`уровень`*
`--elevel=`*`уровень`*

Устанавливает уровень выводимых сообщений: `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` и `PANIC`. Уровень по умолчанию: `INFO`.

`--help`

Вывести справку об аргументах командной строки pg_repack и завершиться.

`-V`
`--version`

Вывести версию pg_repack и завершиться.

Переменные окружения

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Параметры подключения по умолчанию

Эта утилита, как и большинство других утилит Postgres Pro, также использует переменные среды, поддерживаемые libpq (см. Раздел 32.15).

Примеры

Выполнение неблокирующей кластеризации (`CLUSTER`) всех кластеризованных таблиц и полной очистки (`VACUUM FULL`) всех некластеризованных таблиц в базе данных `test`:

```
$ pg_repack test
```

Выполнение неблокирующей полной очистки (`VACUUM FULL`) в таблицах `foo` и `bar` в базе данных `test` (кластерные индексы, если они есть, игнорируются):

```
$ pg_repack --no-order --table foo --table bar test
```

Перемещение всех индексов таблицы `foo` в табличное пространство `tbs`:

```
$ pg_repack -d test --table foo --only-indexes --tablespace tbs
```

Перемещение указанного индекса в табличное пространство `tbs`:

```
$ pg_repack -d test --index idx --tablespace tbs
```

Диагностика

В случае ошибок pg_repack выдаёт соответствующие сообщения. В следующем списке поясняются причины ряда ошибок.

В случае неисправимых ошибок вам потребуется провести очистку вручную. Для этого просто удалите `pg_repack` из базы данных, а затем снова установите это расширение.

Для Postgres Pro или PostgreSQL версии 9.1 и новее выполните:

```
DROP EXTENSION pg_repack CASCADE
```

в базе данных, где произошла ошибка. Затем выполните:

```
CREATE EXTENSION pg_repack
```

Для предыдущих версий загрузите скрипт `$SHAREDIR/contrib/uninstall_pg_repack.sql` в базу данных, где произошла ошибка, а затем повторно загрузите `$SHAREDIR/contrib/pg_repack.sql`.

```
INFO: database "db" skipped:
pg_repack VER is not installed in the database:
pg_repack is not installed in the database when the --all option is specified.
```

Создайте расширение `pg_repack` в базе данных.

```
ERROR: pg_repack VER is not installed in the database:
```

```
pg_repack is not installed in the database specified by --dbname
```

Создайте расширение `pg_repack` в базе данных.

```
ERROR: program 'pg_repack V1' does not match database library 'pg_repack V2':
There is a mismatch between the pg_repack binary and the database library
(.so or .dll).
```

Обнаруженное несоответствие версий может быть связано с нахождением некорректной программы в переменной `PATH` или обращением к неправильной базе данных. Проверьте корректность каталога программы и базы данных; если всё верно, возможно, вам придётся переустановить `pg_repack`.

```
ERROR: extension 'pg_repack V1' required, found 'pg_repack V2':
The SQL extension found in the database does not match the version required by the
 pg_repack program.
```

Столкнувшись с такой ошибкой, удалите расширение из базы данных и пересоздайте его, как описано в Разделе «Установка».

```
ERROR: relation "table" must have a primary key or not-null unique keys:
The target table doesn't have a PRIMARY KEY or any UNIQUE constraints defined.
```

Создайте в указанной таблице ограничение PRIMARY KEY или UNIQUE.

```
ERROR: query failed: ERROR: column "col" does not exist:
The target table doesn't have columns specified by --order-by option.
```

Укажите существующие столбцы.

```
WARNING: the table "tbl" already has a trigger called z_repack_trigger:
The trigger was probably installed during a previous attempt
to run pg_repack on the table which was interrupted
and for some reason failed to clean up the temporary objects.
```

Вы можете ликвидировать все временные объекты, удалив и создав расширение заново: это описано подробно в Разделе «Установка».

```
WARNING: trigger "trg" conflicting on table "tbl":
The target table has a trigger whose name follows z_repack_trigger
in alphabetical order.
```

Триггер `z_repack_trigger` должен быть последним триггером в наборе BEFORE. Переименуйте свой триггер, чтобы он шёл в алфавитном порядке перед `pg_repack`; это можно сделать командой:

```
ALTER TRIGGER zzz_my_trigger
 ON таблица RENAME TO yyy_my_trigger;
```

```
ERROR: Another pg_repack command may be running on the table. Please try again later.
```

Когда две команды `pg_repack` выполняются одновременно, возможна взаимоблокировка. В таком случае стоит попробовать выполнять эти команды с некоторой задержкой.

```
WARNING: Cannot create index "schema"."index_xxxxx", already exists
DETAIL: An invalid index may have been left behind by a previous
pg_repack on the table which was interrupted. Please use DROP INDEX
"schema"."index_xxxxx" to remove this index and try again.
```

Похоже, что в базе данных остался временный индекс, созданный программой pg_repack, но он не удаляется автоматически во избежание недоразумений. Если этот индекс действительно был создан старым заданием pg_repack и по какой-то причине не удалился, вы можете просто выполнить `DROP INDEX` и повторить команду `repack` ещё раз.

Ограничения

pg_repack имеет следующие ограничения.

Временные таблицы

pg_repack не может реорганизовывать временные таблицы.

Индексы GiST

pg_repack не может кластеризовать таблицу согласно индексу `GiST`.

Команды DDL

Пока работает pg_repack, нельзя выполнять команды DDL с целевыми таблицами, *за исключением* `VACUUM` и `ANALYZE`. Для реализации этого ограничения pg_repack устанавливает блокировку `ACCESS SHARE` в целевой таблице на время реорганизации таблицы.

Если вы используете версию расширения 1.1.8 или старее, воздержитесь от выполнения каких-либо команд DDL с целевыми таблицами в процессе работы pg_repack. Во многих случаях pg_repack выдаст ошибку и отменит свои изменения корректно, но с этими старыми версиями в некоторых случаях было возможно разрушение данных.

См. также

clusterdb, vacuumdb

I.3. Дополнительные серверные приложения Postgres Pro

В этом разделе рассматриваются дополнительные серверные приложения Postgres Pro Standard. Они обычно запускаются в той же системе, где работает сервер баз данных. Обратитесь также к Справке: «Серверные приложения Postgres Pro» за информацией о серверных приложениях, относящихся к ядру дистрибутива Postgres Pro Standard.

mamonsu

mamonsu — агент мониторинга для сбора метрик операционной системы и Postgres Pro

Синтаксис

`mamonsu agent` [*действие_агента*]

`mamonsu bootstrap -M` *пользователь_mamonsu* [`-c` | `--config`] [`-x` | `--create-extensions`] [*параметры_подключения*]

`mamonsu export` {`template` | `config`} *имя_файла* [*параметры_экспорта*]

`mamonsu report` [*параметры_отчёта*] [*параметры_подключения*]

`mamonsu tune` [*параметры_оптимизации*] [*параметры_подключения*]

`mamonsu upload` [*параметры_передачи*]

`mamonsu zabbix dashboard upload` *имя_шаблона*

`mamonsu zabbix` {`template` | `host` | `hostgroup`} *действие_сервера*

`mamonsu zabbix item` {`error` | `lastvalue` | `lastclock`} *ид_узла*

`mamonsu zabbix version`

`mamonsu` [`-c` | `--config`] [`-d` | `--daemonize`]

`mamonsu --version`

`mamonsu --help`

Описание

Приложение mamonsu представляет собой агент мониторинга, собирающий метрики Postgres Pro и системы, которые можно визуализировать на сервере Zabbix. В отличие от стандартного Zabbix Agent, который тоже можно настроить для сбора метрик Postgres Pro, mamonsu использует одно подключение к базе данных, что позволяет минимизировать влияние на целевую систему.

Агент mamonsu включает следующие компоненты:

- Руководящий процесс, наблюдающий за активностью системы и базы данных.
- Модули, определяющие, какие метрики Postgres Pro и системы будут собираться.
- Файлы конфигурации и шаблона, в которых определяется, какие модули будут использоваться и как будут представляться собранные данные.

mamonsu является активным агентом, что означает, что он передаёт серверу Zabbix данные сразу, по мере их сбора. В mamonsu имеется набор готовых метрик для мониторинга Postgres Pro и системы, но вы можете разработать и дополнительные модули для сбора других интересующих вас метрик.

В mamonsu также реализован интерфейс командной строки для управления некоторыми параметрами сервера Zabbix, для получения общего представления о конфигурации целевой системы, а также для оптимизации параметров Postgres Pro и системы «на лету». Список всех команд mamonsu приведён в Разделе «Справка по командной строке».

Установка и подготовка

> **Важно**
>
> Перед началом установки рекомендуется внимательно ознакомиться с Замечаниями о совместимости.

Для использования mamonsu вы должны создать учётную запись Zabbix и настроить сервер Zabbix, как описано в *документации Zabbix*. Естественно, если вы планируете наблюдать за метриками Postgres Pro, у вас должен работать экземпляр Postgres Pro. Если вам нужны инструкции по установке и настройке кластера Postgres Pro, обратитесь к Главе 16 и Разделу 17.2.

> **Примечание**
>
> Тогда как метрики Postgres Pro mamonsu может собирать с удалённого кластера, метрики системы собираются только локально. Поэтому если вы решите собирать метрики удалённого экземпляра Postgres Pro, во избежание путаницы обязательно отключите системные метрики, иначе оба набора будут представлены в Zabbix как метрики одного узла.

Приложение mamonsu поставляется вместе с Postgres Pro Standard в виде отдельного предварительно собранного пакета `mamonsu` (подробные инструкции по установке приведены в Главе 16). Завершив установку mamonsu, выполните следующие действия, чтобы начать сбор метрик:

1. **По желанию произведите специальную подготовку mamonsu**

   Если вы опустите этот этап, метрики можно будет собирать только с правами суперпользователя, что не рекомендуется.

   На этом этапе команда `bootstrap` задаёт пользователя mamonsu, создаёт схему `mamonsu` в базе данных, заданной явно или по умолчанию, а также расширения и объекты, необходимые для мониторинга.

   Команда `bootstrap` не имеет обязательных параметров. При запуске без параметров она берёт необходимые значения из файла конфигурации (по умолчанию — `/etc/mamonsu/agent.conf`). Ниже приведён простой пример подготовительной процедуры:

   1. В разделе `[postgres]` файла `agent.conf` укажите параметры подключения для кластера Postgres Pro, за которым вы собираетесь наблюдать. За дополнительными сведениями обратитесь к разделу Параметры подключения.
   2. Выполните следующую команду для подготовки mamonsu:

      ```
      mamonsu bootstrap
      ```

      Эта команда создаст пользователя `mamonsu` с паролем `mamonsu`, а также функции мониторинга в базе данных, указанной в `agent.conf`, и предоставит право выполнять их пользователю `mamonsu`.

   В следующем примере параметры `bootstrap` передаются явно.

   1. Создайте непривилегированного пользователя базы данных для mamonsu. Например:

      ```
      CREATE USER mamonsu_user WITH PASSWORD 'mamonsu_password';
      GRANT pg_monitor TO mamonsu_user;
      ```

   2. Создайте базу данных, которая будет применяться для установления подключения к Postgres Pro. Например:

      ```
      CREATE DATABASE mamonsu_database OWNER mamonsu_user;
      ```

   3. Выполните следующую команду для подготовки mamonsu:

```
mamonsu bootstrap -M mamonsu_user -x -c /etc/mamonsu/agent.conf -d
 mamonsu_database -U postgres --host myhost --port 5432
```

Для получения подробной информации о параметрах `-M`, `-x` и `-c` см. `bootstrap`. За дополнительными сведениями о других параметрах обратитесь к разделу Параметры подключения. Эта команда создаст функции мониторинга в *`базе_mamonsu`* и предоставит право выполнять их *`пользователю_mamonsu`*. Кроме того, будет создано расширение pg_buffercache, позволяющее собирать метрики общего кеша буферов. При этом задаётся другой путь к файлу конфигурации.

В результате суперпользователь не требуется для дальнейших подключений. Обратите внимание, что в этой базе данных создаётся несколько таблиц в схеме `mamonsu`. Не удаляйте эти таблицы, так как они необходимы для работы mamonsu.

2. **Настройте mamonsu**

Отредактируйте конфигурационный файл `agent.conf`, по умолчанию размещённый в каталоге `/etc/mamonsu/`.

- Настройте параметры, связанные с Zabbix. Значение `address` должно указывать на работающий сервер Zabbix, а в поле `client` должно задаваться имя узла Zabbix. Узлы, доступные для вашей учётной записи Zabbix, вы можете увидеть в его веб-интерфейсе на странице **Configuration** (Настройка) > **Hosts** (Узлы сети).

```
[zabbix]
; включён по умолчанию
enabled = True
client = имя_узла_zabbix
address = сервер_zabbix
```

- По умолчанию mamonsu будет собирать и метрики Postgres Pro, и системные метрики. При необходимости вы можете запретить сбор метрик первого или второго типа, установив в файле `agent.conf` для параметра `enabled` значение `False` в разделе `[postgres]` или `[system]`, соответственно.

```
[system]
; включён по умолчанию
enabled = True
```

Примечание

Тогда как метрики Postgres Pro mamonsu может собирать с удалённого кластера, метрики системы собираются только локально. Поэтому если вы решите собирать метрики удалённого экземпляра Postgres Pro, во избежание путаницы обязательно отключите системные метрики, иначе оба набора будут представлены в Zabbix как метрики одного узла.

- Если вы будете собирать метрики Postgres Pro, необходимо задать параметры подключения к кластеру Postgres Pro, за которым вы намерены наблюдать. Для этого задайте в полях `user`, `password` и `database` соответственно имя пользователя (*`mamonsu_user`*), пароль (*`mamonsu_password`*) и базу данных (*`mamonsu_database`*), указанные при подготовке. Если вы пропустили подготовительную процедуру, задайте имя и пароль суперпользователя, а также целевую базу.

```
[postgres]
; включён по умолчанию
enabled = True
user = пользователь_mamonsu
database = база_mamonsu
```

```
password = пароль_mamonsu
port = 5432
```

Это базовые параметры, позволяющие начать использование mamonsu. Кроме них можно настроить и другие параметры mamonsu, описанные в Разделе «Параметры конфигурации».

3. **Настройте представление метрик на сервере Zabbix**

   1. Создайте шаблон, определяющий, как будут представляться собранные метрики на сервере Zabbix:

      ```
      mamonsu export template template.xml
      ```

      mamonsu формирует файл шаблона `template.xml` в текущем каталоге. По умолчанию отображаемым в Zabbix именем шаблона будет `PostgresPro-OS`, где *`OS`* — название вашей операционной системы. Чтобы получить шаблон с другим отображаемым именем, в показанную выше команду можно добавить аргумент `--template-name`.

   2. При желании укажите учётные данные в Zabbix в следующих переменных окружения в той системе, где настраивается мониторинг:

      - Задайте в переменных `ZABBIX_USER` и `ZABBIX_PASSWD` имя и пароль вашей учётной записи Zabbix, соответственно.
      - Задайте в переменной `ZABBIX_URL` адрес `http://zabbix/`

      Если этого не сделать, вам нужно будет указывать следующие параметры при каждом вызове любой подкоманды `mamonsu zabbix`:

      ```
      --url=http://zabbix/ --user=имя_пользователя_zabbix --
      password=пароль_пользователя_zabbix
      ```

   3. Загрузите шаблон `template.xml` на сервер Zabbix.

      ```
      mamonsu zabbix template export template.xml
      ```

      Шаблон также можно загрузить через веб-интерфейс Zabbix: воспользуйтесь своей учётной записью в Zabbix и перейдите на страницу **Templates** (Шаблоны) > **Import** (Импорт).

   4. Привяжите полученный шаблон к узлу, за которым будете наблюдать.

      Для этого в веб-интерфейсе Zabbix выберите нужный узел, перейдите на страницу **Templates** (Шаблоны) > **Add** (Добавить), выберите шаблон и затем щёлкните **Update** (Обновить).

      ### Подсказка

      Если вы хотите связать шаблон с создаваемым узлом в Zabbix, вы можете сделать это в командной строке, используя команды `mamonsu zabbix`. Подробнее это описывается в Подразделе «Управление параметрами сервера Zabbix в командной строке».

Завершив настройку, запустите mamonsu. Например, в системах Linux вы можете запустить mamonsu в виде службы так:

```
service mamonsu start
```

mamonsu считывает все параметры из файла конфигурации mamonsu и начинает мониторинг системы.

Справка по командной строке

agent

Синтаксис:

```
mamonsu agent { metric-list | metric-get имя_метрики | version }
```

Предоставлять информацию о собираемых метриках из командной строки. Вы можете задать одно из следующих действий:

`metric-list`

Вывести список метрик, которые собирает mamonsu. Вывод этой команды включает ключ метрики, её последнее значение и время получения этого значения.

`metric-get` *`имя_метрики`*

Получить последнее значение указанной метрики. Список доступных метрик можно получить с помощью команды `metric-list`.

`version`

Вывести версию mamonsu.

bootstrap

Синтаксис:

```
mamonsu bootstrap -M имя пользователя
        [-c[ файл конфигурации] | --config[=файл конфигурации]]
        [-x | --create-extensions] [параметры подключения]
```

Подготовить mamonsu к работе. Данная команда может принимать следующие параметры:

`-M`

Задаёт имя пользователя, который будет владеть всеми процессами mamonsu. Указанный пользователь не должен быть привилегированным.

`-c` *`файл_конфигурации`*
`--config=`*`файл_конфигурации`*

Задаёт файл `agent.conf`, в котором раздел `[postgres]` содержит имя базы данных, которое будет использоваться, если параметр `--dbname` не указан среди *`параметров подключения`*.

Значение по умолчанию: `/etc/mamonsu/agent.conf`

`-x`
`--create-extensions`

Создаёт вспомогательные расширения. Будет создано расширение pg_buffercache, которое больше не создаётся по умолчанию. Для Postgres Pro версии 12 или выше команда `bootstrap -x` создаст функции для работы с pgpro_stats и предоставит права на их выполнение пользователю `mamonsu`.

`параметры_подключения`

Необязательные параметры подключения, задаваемые в командной строке.

export

Синтаксис:

```
mamonsu export config имя_файла.conf  [--add-plugins=каталог_модулей]
mamonsu export template имя_файла.xml [--add-plugins=каталог_модулей]
                                     [--template-name=имя_шаблона]
                                     [--application=имя_приложения]
                                     [--old-zabbix]
```

Сгенерировать файл конфигурации или файл шаблона для набора метрик. Для настройки набора метрик можно передать дополнительные параметры:

`--add-plugins=`*`каталог_модулей`*

Собирать метрики, определённые в дополнительных модулях, расположенных в заданном *`каталоге_модулей`*. Если вы будете использовать какие-либо дополнительные модули, этот параметр нужно передавать при формировании файла конфигурации и шаблона.

`--template-name=`*`имя_шаблона`*

Задаёт название шаблона, которое будет отображаться на сервере Zabbix.

Значение по умолчанию: `PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы

`--application=`*`имя_приложения`*

Задаёт идентификатор, с которым собранные метрики будут отображаться на сервере Zabbix.

Значение по умолчанию: `App-PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы

`--old-zabbix`

Экспортировать шаблон для сервера Zabbix версии 4.2 или старее. По умолчанию шаблон генерируется в формате, совместимом с Zabbix 4.4 и новее.

export zabbix-parameters

Синтаксис:

```
mamonsu export zabbix-parameters имя_файла.conf [--add-plugins=каталог_модулей]
                                      [--plugin-type={pg | sys | all}]
                                      [--pg-version=версия]
                                      [--config=файл_конфигурации]
```

Экспортировать конфигурацию метрик для использования со стандартным *агентом Zabbix*. Для настройки сбора метрик можно передать дополнительные параметры:

`--add-plugins=`*`каталог_модулей`*

Собирать метрики, определённые в дополнительных модулях, расположенных в заданном *`каталоге_модулей`*. Если вы будете использовать какие-либо дополнительные модули, этот параметр нужно передавать при формировании файла конфигурации и шаблона.

`--plugin-type={pg | sys | all}`

Типы метрик, которые должны собираться:

- `pg` — метрики Postgres Pro.
- `sys` — системные метрики.
- `all` — все метрики.

По умолчанию: `all`

`--pg-version=`*`версия`*

Задаёт основную версию сервера, для которого будет настраиваться набор метрик. Программа `mamonsu` может собирать метрики во всех поддерживаемых версиях Postgres Pro, а также в PostgreSQL, начиная с версии 9.5.

Значение по умолчанию: `10`

`--config=`*`файл_конфигурации`*

Задаёт путь к файлу `agent.conf`, который будет источником определений метрик.

Значение по умолчанию: `/etc/mamonsu/agent.conf`

export zabbix-template

Синтаксис:

```
mamonsu export zabbix-template имя_файла.conf [--add-plugins=каталог_модулей]
                                 [--plugin-type={pg | sys | all}]
                                 [--template-name=имя_шаблона]
                                 [--application=имя_приложения]
                                 [--config=файл_конфигурации]
                                 [--old-zabbix]
```

Экспортировать шаблон для использования со стандартным агентом Zabbix. Для настройки конфигурации можно передать дополнительные параметры:

`--add-plugins=`*`каталог_модулей`*

Собирать метрики, определённые в дополнительных модулях, расположенных в заданном *`каталоге_модулей`*. Если вы будете использовать какие-либо дополнительные модули, этот параметр нужно передавать при формировании файла конфигурации и шаблона.

`--plugin-type={pg | sys | all}`

Типы метрик, которые должны собираться:

- `pg` — метрики Postgres Pro.
- `sys` — системные метрики.
- `all` — все метрики.

По умолчанию: `all`

`--template-name=`*`имя_шаблона`*

Задаёт название шаблона, которое будет отображаться на сервере Zabbix.

Значение по умолчанию: `PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы

`--application=`*`имя_приложения`*

Задаёт идентификатор, с которым собранные метрики будут отображаться на сервере Zabbix.

Значение по умолчанию: `App-PostgresPro-`*`OS`*, где *`OS`* — название вашей операционной системы

`--config=`*`файл_конфигурации`*

Задаёт путь к файлу `agent.conf`, который будет источником определений метрик.

Значение по умолчанию: `/etc/mamonsu/agent.conf`

`--old-zabbix`

Экспортировать шаблон для сервера Zabbix версии 4.2 или старее. По умолчанию шаблон генерируется в формате, совместимом с Zabbix 4.4 и новее.

report

Синтаксис:

```
mamonsu report [--run-system=логическое_значение] [--run-postgres=логическое_значение]
               [--print-report=логическое_значение] [--report-path=файл_отчёта]
               [--disable-sudo] [параметры_подключения]
```

Выдать подробный отчёт об оборудовании, операционной системе, использовании памяти и других характеристиках объекта мониторинга, а также конфигурацию Postgres Pro.

Отчёт настраивается с помощью следующих необязательных параметров:

`--run-system=`*`логическое_значение`*

Включить в генерируемый отчёт информацию о системе.

По умолчанию: `True` (вкл.)

`--run-postgres=`*`логическое_значение`*

Включить в генерируемый отчёт информацию о Postgres Pro.

По умолчанию: `True` (вкл.)

`--print-report=`*`логическое_значение`*

Вывести отчёт в stdout.

По умолчанию: `True` (вкл.)

`--report-path=`*`файл_отчёта`*

Вывести отчёт в указанный файл.

Путь по умолчанию: `/tmp/report.txt`

`--disable-sudo`

Собирать при формировании отчёта только те данные, для получения которых не нужны права суперпользователя. Этот режим поддерживается только в системах Linux.

`параметры_подключения`

Необязательные параметры подключения, задаваемые в командной строке.

tune

Синтаксис:

```
mamonsu tune [--dry-run] [--disable-sudo] [--log-level {INFO|DEBUG|WARN}]
             [--dont-tune-pgbadger] [--dont-reload-postgresql]
             [параметры_подключения]
```

Оптимизировать конфигурацию Postgres Pro и системы, исходя из собранной статистики. Настроить оптимизацию позволяют следующие параметры:

`--dry-run`

Вывести параметры конфигурации системы и Postgres Pro, которые будут настроены, но не изменять их фактически.

`--disable-sudo`

Оптимизировать только те параметры, для изменения которых не нужны права суперпользователя. Этот режим поддерживается только в системах Linux.

`--dont-tune-pgbadger`

Не оптимизировать параметры pgbadger.

`--log-level { INFO | DEBUG | WARN}`

Изменить уровень детализации сообщений.

Значение по умолчанию: `INFO`

`--dont-reload-postgresql`

Не допустить выполнения приложением mamonsu функции `pg_reload_conf()`. В этом случае изменения некоторых параметров, которые вступают в силу только после перезагрузки конфигурации Postgres Pro, не начнут действовать немедленно.

параметры_подключения

Необязательные параметры подключения, задаваемые в командной строке.

upload

Синтаксис:

```
mamonsu upload [--zabbix-file=файл_метрик]
               [--zabbix-address=адрес_zabbix] [--zabbix-port=порт]
               [--zabbix-client=имя_узла_zabbix] [--zabbix-log-level={INFO|DEBUG|WARN}]
```

Передать данные метрик, ранее сохранённые в файле, на сервер Zabbix для визуализации. Более подробно получение файла метрик рассматривается в Подразделе «Параметры вывода».

Эта команда может принимать следующие параметры:

`--zabbix-address=`*адрес_zabbix*

Адрес сервера Zabbix.

Значение по умолчанию: `localhost`

`--zabbix-port=`*порт*

Порт сервера Zabbix.

Значение по умолчанию: 10051

`--zabbix-file=`*файл_метрик*

Имя текстового файла с метриками, которые можно визуализировать, например `localhost.log`.

`--zabbix-client=`*имя_узла_zabbix*

Имя узла в Zabbix.

Значение по умолчанию: `localhost`

`--zabbix-log-level={INFO|DEBUG|WARN}`

Изменить уровень детализации сообщений.

Значение по умолчанию: `INFO`

zabbix dashboard upload

Синтаксис:

```
mamonsu dashboard upload имя_шаблона
```

Загрузить панель Zabbix с метриками mamonsu в шаблон на сервере Zabbix версии 6.0 или выше.

zabbix item

Синтаксис:

```
mamonsu zabbix item {error | lastvalue | lastclock } имя_узла
```

Просмотреть заданное свойство последних метрик, полученных сервером Zabbix для указанного узла.

zabbix version

Синтаксис:

```
mamonsu zabbix version
```

Получить версию сервера Zabbix, к которой подключён агент mamonsu.

zabbix host

Синтаксис:

```
mamonsu zabbix host list
mamonsu zabbix host show имя_узла
mamonsu zabbix host id имя_узла
mamonsu zabbix host delete ид_узла
mamonsu zabbix host create имя_узла ид_группы_узлов ид_шаблона адрес_mamonsu
mamonsu zabbix host info {templates | hostgroups | graphs | items} ид_узла
```

Позволяет управлять узлами Zabbix, выполняя одно из действий, описанных в Подразделе «Управление сервером Zabbix».

zabbix hostgroup

Синтаксис:

```
mamonsu zabbix hostgroup list
mamonsu zabbix hostgroup show имя_группы_узлов
mamonsu zabbix hostgroup id имя_группы_узлов
mamonsu zabbix hostgroup delete ид_группы_узлов
mamonsu zabbix hostgroup create имя_группы_узлов
```

Позволяет управлять группами узлов Zabbix, выполняя одно из действий, описанных в Подразделе «Управление сервером Zabbix».

zabbix template

Синтаксис:

```
mamonsu zabbix template list
mamonsu zabbix template show имя_шаблона
mamonsu zabbix template id имя_шаблона
mamonsu zabbix template delete ид_шаблона
mamonsu zabbix template export файл
```

Позволяет управлять шаблонами Zabbix, выполняя одно из действий, описанных в Подразделе «Управление сервером Zabbix».

[--config] [--daemonize]

Синтаксис:

```
mamonsu [-c файл конфигурации | --config=файл конфигурации]
        [-d | --daemonize]
```

Запустить в качестве агента мониторинга на переднем плане или в фоновом режиме, используя параметры, загруженные из файла конфигурации. Если указанный файл конфигурации или стандартный файл конфигурации не существует, завершиться с ошибкой. Полезно для отладки. Принимаются следующие параметры:

`-c` *`файл_конфигурации`*
`--config=`*`файл_конфигурации`*

Задаёт файл `agent.conf` для загрузки параметров конфигурации.

Значение по умолчанию: `/etc/mamonsu/agent.conf`

`-d`
`--daemonize`

Запустить в фоновом режиме. Без этого указания mamonsu будет работать на переднем плане.

--version

Синтаксис:

```
mamonsu --version
```

Вывести версию `mamonsu`.

--help

Синтаксис:

```
mamonsu --help
```

Вывести справку по параметрам командной строки `mamonsu`.

Параметры подключения

При вызове вышеуказанных команд *`параметры_подключения`* в командной строке задают параметры подключения к целевому кластеру Postgres Pro. *`параметры_подключения`* могут содержать аргументы `--host`, `--port`, `--dbname` (`-d`), `--username` (`-U`), и `--password` (`-W`). В аргументе `--dbname` (`-d`) должна указываться `mamonsu_database`, созданная для наблюдения. Заметьте, что в аргументе `--username` (`-U`) необходимо передать имя суперпользователя в данном кластере.

Если вы не зададите *`параметры_подключения`*, mamonsu будет использовать соответствующие параметры из файла конфигурации.

Управление сервером Zabbix

Используя mamonsu, вы можете управлять некоторой функциональностью Zabbix из командной строки. А именно, вы можете создавать или удалять узлы и группы узлов Zabbix, а также генерировать, импортировать и удалять шаблоны Zabbix с помощью описанных ниже команд. В них *`имя_объекта`* должно соответствовать типу объекта Zabbix в команде: template (шаблон), host (узел) или hostgroup (группа узлов).

`list`

Вывести список доступных шаблонов, узлов или групп узлов.

`show` *`имя_объекта`*

Вывести информацию об указанном шаблоне, узле или группе узлов.

`id` *`имя_объекта`*

Показать идентификатор заданного объекта, назначаемый автоматически сервером Zabbix.

`delete` *`ид_объекта`*

Удалить объект с заданным идентификатором.

`create` *`имя_группы_узлов`*
`create` *`имя_узла ид_группы_узлов ид_шаблона адрес_mamonsu`*

Создать новый узел или группу узлов.

`export` *`имя_шаблона`*

Сформировать шаблон Zabbix.

`info {templates | hostgroups | graphs | items}` *`ид_узла`*

Вывести подробную информацию о шаблонах, группах узлов, графиках и метриках, связанных с узлом с заданным идентификатором.

Параметры конфигурации

Файл конфигурации `agent.conf` по умолчанию располагается в каталоге `/etc/mamonsu`. Он содержит несколько групп параметров, определяющих, какие метрики собирать и как выводить собранные данные:

- параметры подключения
- параметры вывода
- параметры модулей

Все параметры должны задаваться в формате *`параметр`* `=` *`значение`*.

Параметры подключения агента

[postgres]

В разделе `[postgres]` настраивается сбор метрик Postgres Pro и могут содержаться следующие параметры:

`enabled`

Включает/отключает сбор метрик Postgres Pro при значении, равном `True` или `False`, соответственно.

По умолчанию: `True` (вкл.)

`user`

Пользователь, от имени которого будут запускаться процессы наблюдения за кластером. Это должен быть пользователь mamonsu, заданный при подготовке, либо суперпользователь, если команда `bootstrap` не запускалась.

Имя по умолчанию: `postgres`

`password`

Пароль указанного пользователя.

`database`

База данных, через подключение к которой будут собираться метрики.

Имя по умолчанию: `postgres`

`host`

Адрес сервера базы данных.

Значение по умолчанию: `localhost`

`port`

Целевой порт на сервере.

Значение по умолчанию: 5432

`application_name`

Имя приложения, идентифицирующее процесс mamonsu, выполняющийся в кластере Postgres Pro.

Значение по умолчанию: `mamonsu`

`query_timeout`

Значение statement_timeout для сеанса mamonsu, в секундах. Если запрос метрики Postgres Pro не успевает выполниться за указанное время, он прерывается.

Значение по умолчанию: 10

Также вместо параметров конфигурации можно использовать следующие переменные среды: `PGUSER`, `PGPASSWORD`, `PGHOST`, `PGDATABASE`, `PGPORT` и `PGAPPNAME`. За подробным описанием этих переменных обратитесь к Разделу 32.15 для их значения.

[system]

В разделе `[system]` настраивается сбор системных метрик и могут содержаться следующие параметры:

`enabled`

Включает/отключает сбор системных метрик при значении, равном `True` или `False`, соответственно.

По умолчанию: `True` (вкл.)

[zabbix]

В разделе `[zabbix]` настраиваются свойства подключения к серверу Zabbix и могут содержаться следующие параметры:

`enabled`

Включает/отключает передачу собранных метрик на сервер Zabbix при значении, равном `True` или `False`, соответственно.

По умолчанию: `True` (вкл.)

`client`

Имя узла в Zabbix.

`address`

Адрес сервера Zabbix.

Значение по умолчанию: 127.0.0.1

`port`

Порт сервера Zabbix.

Значение по умолчанию: 10051

`timeout`

Максимальное время ожидания при подключении к серверу Zabbix (в секундах).

Значение по умолчанию: 15

[agent]

В разделе `[agent]` задаётся расположение службы mamonsu и включается возможность обращения к метрикам в командной строке:

`enabled`

Включает/отключает сбор метрик в командной строке с помощью команды `agent`.

По умолчанию: `True` (вкл.)

`host`

Адрес системы, в которой работает mamonsu.

Значение по умолчанию: 127.0.0.1

`port`

Порт, через который mamonsu принимает соединения.

Значение по умолчанию: 10052

[sender]

В разделе `[sender]` определяется размер очереди, в которой накапливаются данные, прежде чем они передаются серверу Zabbix:

`queue`

Максимальное количество собранных значений метрик, которое может накапливаться локально, прежде чем mamonsu передаст их на сервер Zabbix. После передачи накопленных данных очередь очищается.

Значение по умолчанию: 2048

Параметры вывода

[metric_log]

В разделе `[metric_log]` настраивается сохранение собранных значений метрик в локальных текстовых файлах. Этот раздел может содержать следующие параметры:

`enabled`

Включает/отключает сохранение собранных значений метрик в текстовом файле. Если этот параметр равен `True`, mamonsu создаёт файл `localhost.log` для сохранения значений метрик.

Значение по умолчанию: `False`

`directory`

Указывает каталог, в котором будут сохраняться данные метрик.

Каталог по умолчанию: `/var/log/mamonsu`

`max_size_mb`

Максимальный размер файла метрик, в мегабайтах. По достижении заданного размера этот файл переименовывается в `localhost.log.archive`, и создаётся новый файл `localhost.log`.

Значение по умолчанию: 1024

[log]

В разделе `[log]` настраивается запись сообщений в журнал mamonsu и могут содержаться следующие параметры:

`file`

Задаёт имя файла журнала; может также задавать полный путь.

`level`

Задаёт уровень отладки. Этот параметр может принимать значение `DEBUG`, `ERROR` или `INFO`.

Значение по умолчанию: `INFO`

`format`

Формат записываемых сообщений.

Значение по умолчанию: `[%(asctime)s] %(asctime)s - %(name)s - %(message)s`

Здесь `levelname` — уровень важности сообщения, `asctime` — текущее время, `name` — имя модуля, выдавшего это сообщение (либо `root`, если сообщение выдано не модулем), и `message` — собственно текст сообщения.

Параметры модулей

[plugins]

В разделе `[plugins]` описываются дополнительные модули, осуществляющие сбор метрик, и могут содержаться следующие параметры:

`enabled`

Включает/отключает дополнительные модули, осуществляющие сбор метрик, при значении равном `True` или `False`, соответственно.

Значение по умолчанию: `False`

`directory`

Задаёт каталог, содержащий дополнительные модули. При значении, равном `None`, использование дополнительных модулей отключается.

Значение по умолчанию: `/etc/mamonsu/plugins`

Если вы хотите настроить модуль, который вы дополнительно добавили в mamonsu после установки, для него нужно добавить раздел параметров в файл `agent.conf`.

Синтаксис, которому должен соответствовать этот раздел, показан в примере в Подразделе «Разделы отдельных модулей».

Разделы отдельных модулей

Все встроенные модули устанавливаются в составе mamonsu. Чтобы настроить такой модуль, найдите соответствующий ему раздел под заголовком `Individual Plugin Sections` (Разделы отдельных модулей) и измените значения параметров:

`enabled`

Включает/отключает встроенные модули при значении равном `True` или `False` соответственно. Эти значения чувствительны к регистру.

Значение по умолчанию: `False`

`interval`

Указывает интервал работы модуля в секундах. Параметр задаётся целым числом. Можно установить разные интервалы для разных модулей. Для модулей, собирающих метрики PostgreSQL/Postgres Pro, например autovacuum, checkpoint и т. д., это интервал, в течение которого mamonsu обращается к экземпляру Postgres Pro для сбора метрик. Для системных модулей, например diskstats, memory и т. д., это интервал, в течение которого осуществляется обращение к операционной системе. Для служебных модулей, например agentapi, logsender, zbxsender, это интервал, в течение которого вызываются их функции. Обратите внимание, что параметр `interval` модуля zbxsender связан с параметром `timeout` раздела `[zabbix]`. Максимальное время ожидания ответа сервера Zabbix, превышающее интервал работы модуля, может привести к неправильному поведению модуля zbxsender: модуль перезапускается с сообщением об ошибке, если подключение или передача данных занимают больше времени, чем значение, указанное в параметре `interval`.

Значение по умолчанию: 60

В следующем примере показаны разделы, соответствующие встроенным модулям `preparedtransaction` и `pgprobackup`:

```
[preparedtransaction]
max_prepared_transaction_time = 60
interval = 60

[pgprobackup]
enabled = false
interval = 300
backup_dirs = /backup_dir1,/backup_dir2
pg_probackup_path = /usr/bin/pg_probackup-11
```

[preparedtransaction]

Этот модуль получает возраст старейшей подготовленной транзакции в секундах, а также число всех транзакций, подготовленных для двухфазной фиксации. За дополнительными сведениями обратитесь к PREPARE TRANSACTION и Разделу 52.16.

Параметр `max_prepared_transaction_time` задаёт в секундах пороговое значения для возраста подготовленной транзакции.

Данный модуль собирает две метрики: `pgsql.prepared.count` (число подготовленных транзакций) и `pgsql.prepared.oldest` (возраст старейшей подготовленной транзакции в секундах).

Если значение `pgsql.prepared.oldest` превышает порог, заданный параметром `max_prepared_transaction_time`, срабатывает триггер со следующим сообщением: «PostgreSQL prepared transaction is too old on {host}» (Слишком старая подготовленная транзакция PostgreSQL на {узел}).

[pgprobackup]

Этот модуль обращается к программе pg_probackup для отслеживания состояния её резервных копий, а также вычисляет объём каталогов, в которых хранятся все файлы копий.

Заметьте, что этот модуль подсчитывает общий объём всех файлов в целевом каталоге. Убедитесь в том, что в целевой каталог не попадают посторонние файлы.

Параметр `backup_dirs` задаёт разделённый запятыми список путей к каталогам, для которых будут собираться метрики.

Параметр `pg_probackup_path` задаёт путь к программе pg_probackup.

По умолчанию этот модуль отключён. Чтобы включить его, задайте для параметра `enabled` значение `True`.

Данный модуль собирает две метрики: `pg_probackup.dir.size[#каталог_копий]` (объём целевого каталога) и `pg_probackup.dir.error[#каталог_копий]` (ошибки копий) для каждого указанного *`каталога_копий`*.

Если какая-либо копия имеет ошибочное состояние, например `ERROR`, `CORRUPT`, `ORPHAN`, этот модуль вызывает срабатывание триггера.

Использование

Сбор и просмотр метрик

Сразу после запуска mamonsu начинает сбор метрик системы и Postgres Pro. Команда `agent` позволяет в реальном времени получать текущие значения собираемых метрик в командной строке.

Чтобы просмотреть список имеющихся метрик, воспользуйтесь командой `agent metric-list`:

```
mamonsu agent metric-list
```

Вывод этой команды включает ключ метрики, её последнее значение и время получения этого значения. Например:

```
system.memory[active]      7921004544    1564570818
system.memory[swap_cache]      868352    1564570818
pgsql.connections[idle]           6.0    1564570818
pgsql.archive_command[failed_trying_to_archive]    0    1564570818
```

Чтобы просмотреть текущее значение определённой метрики, воспользуйтесь командой `agent metric-get`:

```
mamonsu agent metric-get имя_метрики
```

Здесь *`имя_метрики`* — ключ интересующей вас метрики, представленный в выводе команды `metric-list`. Например: `pgsql.connections[idle]`.

Вы можете просмотреть графики собранных метрик в веб-интерфейсе Zabbix в разделе **Graphs** (Графики). Подробнее об использовании Zabbix рассказывается в его официальной документации по адресу *https://www.zabbix.com/documentation/current/manual*.

Если вы решили сохранять все собираемые метрики в файле как описано в «Параметры вывода», собранные метрики можно передать на сервер Zabbix для визуализации, воспользовавшись командой upload.

Добавление дополнительных модулей

Вы можете расширить функциональность mamonsu, добавив собственные дополнительные модули следующим образом:

1. Сохраните все нужные вам дополнительные модули в одном каталоге, например, в `/etc/mamonsu/plugins`.
2. Убедитесь в том, что этот каталог указан в файле конфигурации в разделе `[plugins]`:

   ```
   [plugins]
   directory = /etc/mamonsu/plugins
   ```
3. Сгенерируйте новый шаблон Zabbix, включающий информацию дополнительных модулей:

   ```
   mamonsu export template template.xml --add-plugins=/etc/mamonsu/plugins
   ```
4. Загрузите сгенерированный `template.xml` на сервер Zabbix, как описано в Разделе «Установка и подготовка».

Оптимизация конфигурации Postgres Pro и системы

В зависимости от собранных значений метрик, mamonsu может настроить конфигурацию Postgres Pro и системы для оптимальной производительности.

Вы можете получить подробную информацию об оборудовании, операционной системе, использовании памяти и других характеристиках объекта мониторинга, а также конфигурацию Postgres Pro, выполнив команду:

```
mamonsu report
```

Чтобы только просмотреть предлагаемые для оптимизации изменения, но не применять их, запустите команду `tune` с ключом `--dry-run`:

```
mamonsu tune --dry-run
```

Чтобы предлагаемые изменения были применены, запустите команду `tune` без каких-либо параметров:

```
mamonsu tune
```

Вы можете исключить некоторые параметры из отчёта или отключить некоторые методы оптимизации, передав дополнительные аргументы. Полный список аргументов команд `report` и `tune` приведён в Разделе «Справка по командной строке».

Управление параметрами сервера Zabbix в командной строке

Приложение mamonsu позволяет работать с сервером Zabbix, используя командную строку. В частности, вы можете загружать файлы шаблонов на сервер Zabbix, создавать и удалять узлы и группы узлов Zabbix, привязывать шаблоны к узлам и получать последние значения метрик.

Чтобы настроить узел мониторинга в Zabbix с самого начала, вы можете выполнить следующие действия:

1. При желании укажите учётные данные в Zabbix в следующих переменных окружения в той системе, где настраивается мониторинг:

   - Задайте в переменных `ZABBIX_USER` и `ZABBIX_PASSWD` имя и пароль вашей учётной записи Zabbix, соответственно.
   - Задайте в переменной `ZABBIX_URL` адрес `http://zabbix/`

   Если этого не сделать, вам нужно будет указывать следующие параметры при каждом вызове любой подкоманды `mamonsu zabbix`:

   ```
   --url=http://zabbix/ --user=имя_пользователя_zabbix --
   password=пароль_пользователя_zabbix
   ```
2. Сгенерируйте новый файл шаблона и загрузите его на сервер Zabbix:

   ```
   mamonsu export template template.xml
   mamonsu zabbix template export template.xml
   ```
3. Создайте новую группу узлов:

   ```
   mamonsu zabbix hostgroup create имя_группы_узлов
   ```
4. Определите идентификаторы созданной группы и загруженного шаблона, назначаемые автоматически сервером Zabbix:

   ```
   mamonsu zabbix hostgroup id имя_группы_узлов
   mamonsu zabbix template id имя_шаблона
   ```
5. Создайте узел, связанный с этой группой, и привяжите к нему загруженный шаблон с помощью одной команды:

   ```
   mamonsu zabbix host create имя_узла ид_группы_узлов ид_шаблона адрес_mamonsu
   ```

   Здесь *`имя_узла`* — имя создаваемого узла, *`ид_группы_узлов`* и *`ид_шаблона`* — уникальные идентификаторы, полученные на предыдущем шаге, а *`mamonsu_address`* — IP-адрес системы, в которой работает mamonsu.

Создав узел в Zabbix, произведите настройку целевой системы, как описано в Разделе «Установка и подготовка».

Экспорт метрик для стандартного агента Zabbix

Используя mamonsu, вы можете преобразовать определения метрик системы и Postgres Pro в формат, который поддерживает стандартный *агент Zabbix*.

В настоящее время эта функциональность имеет следующие ограничения:

- В Windows экспорт метрик `mamonsu` не поддерживается.
- Определения метрик входящего в Postgres Pro Enterprise расширения `pg_wait_sampling` и механизма CFS не преобразуются.

Важно

Для работы с прокси Zabbix необходимо экспортировать все параметры в формат, который поддерживается стандартным агентом Zabbix.

Чтобы использовать агент Zabbix для сбора метрик mamonsu, выполните следующее:

1. Сгенерируйте файл конфигурации, совместимый со стандартным агентом Zabbix, и поместите его в `/etc/zabbix/zabbix_agent.d/`. Вы можете добавить нужный путь перед именем файла:

   ```
   mamonsu export zabbix-parameters /etc/zabbix/zabbix_agent.d/zabbix_agent.conf
   ```

   Все имеющиеся параметры команды `export zabbix-parameters` описаны в Разделе «Справка по командной строке».
2. Сгенерируйте шаблон для стандартного агента Zabbix:

   ```
   mamonsu export zabbix-template template.xml
   ```

   Все параметры команды `export zabbix-template` описаны в Разделе «Справка по командной строке».
3. Загрузите сгенерированный шаблон на сервер Zabbix, как описано в Разделе «Установка и подготовка».
4. Если вы намерены собирать метрики Postgres Pro, после загрузки шаблона измените в веб-интерфейсе следующие макросы:
   - Вместо `{$PG_CONNINFO}` укажите параметры подключения к серверу Postgres Pro, за которым вы намерены наблюдать.
   - Вместо `{$PG_PATH}` укажите путь к каталогу, где установлена программа psql.

Замечания о совместимости

Если вы хотите обновить mamonsu до версии, несовместимой с предыдущей, то ваши дальнейшие действия зависят от того, нужно ли вам сохранить собранные данные метрик.

- Если вам нужно сохранить собранные данные, выполните следующее:
  1. Установите новую версию mamonsu.
  2. Сгенерируйте новый шаблон для сервера Zabbix.
  3. Если вы выполняли специальную подготовительную процедуру в предыдущей версии mamonsu, запустите команду `bootstrap` ещё раз.
  4. Загрузите новый шаблон на сервер Zabbix.
  5. Переименуйте узел, для которого вы хотите сохранить собранные данные, и оставьте его привязку к старому шаблону.
  6. Создайте новый узел для той же системы и привяжите к нему новый шаблон.
  7. Перезапустите службу mamonsu. Она будет собирать данные для нового узла. Старый узел больше не будет использоваться, но собранные для него данные будут по-прежнему доступны.
- Если вам не нужно сохранять собранные данные, выполните следующее:
  1. Установите новую версию mamonsu.
  2. Сгенерируйте новый шаблон для сервера Zabbix.
  3. Если вы выполняли специальную подготовительную процедуру в предыдущей версии mamonsu, запустите команду `bootstrap` ещё раз.
  4. Загрузите новый шаблон на сервер Zabbix.

5. В параметрах узла Zabbix привяжите к узлу новый шаблон вместо старого.
6. Перезапустите службу mamonsu, чтобы начался сбор данных. Все данные, собранные ранее, будут потеряны.

pgpro_controldata

pgpro_controldata — вывести управляющую информацию кластера БД PostgreSQL/Postgres Pro и параметры совместимости кластера и/или сервера

Синтаксис

`pgpro_controldata` [*параметр*...]

Описание

`pgpro_controldata` выводит управляющую информацию, в частности, версию каталога, записываемую программой `initdb` при инициализации любого кластера PostgreSQL/Postgres Pro. Данная утилита также выводит сведения о журнале предзаписи, включая данные контрольной точки. Эта информация относится ко всему кластеру, а не к отдельной базе данных.

`pgpro_controldata` также помогает проверить совместимость между серверами и кластерами БД PostgreSQL/Postgres Pro. Эта программа может вывести параметры, влияющие на совместимость, и определить, совместим ли сервер с кластером, сравнивая эти параметры.

pgpro_controldata поставляется вместе с Postgres Pro Standard в виде отдельного пакета `pgpro-controldata` (подробные инструкции по установке приведены в Главе 16).

Параметры

`pgpro_controldata` принимает в командной строке параметры, описанные далее. В случае запуска без параметров `pgpro_controldata` просто выводит управляющую информацию так, как это делает pg_controldata. Заметьте, что аргументы командной строки, связанные с совместимостью, `-P` и `-S`, работают в паре как один аргумент `-C`.

Параметры общего назначения

`-B`
`--bindir`

Задаёт каталог с исполняемыми файлами PostgreSQL/Postgres Pro; требуется для получения параметров совместимости сервера.

`-D` *`каталог_данных`*
`--pgdata=`*`каталог_данных`*

Указывает размещение конфигурационных файлов кластера. Если этот ключ опущен, используется значение переменной окружения `PGDATA`.

`-V`
`--version`

Показать версию pgpro_controldata, а затем завершиться.

`-?`
`--help`

Показать справку по аргументам командной строки pgpro_controldata, а затем завершиться.

Параметры, связанные с совместимостью

`-C`
`--compatibility-check`

Вывести все параметры целевого кластера, от которых зависит совместимость выбранного сервера и кластера, и установить, совместимы ли они.

Путь к целевому каталогу данных, доступному для чтения, передаётся в аргументе `-D` или в переменной окружения `PGDATA`.

Если параметр `-B` опущен, подразумевается текущий сервер.

Эта проверка будет полноценной, только если данные кластера и сервер рассчитаны на одну архитектуру процессора и одинаковый порядок байтов.

`-P`
`--cluster-compatibility-params`

Вывести все параметры целевого кластера, которые могут повлиять на совместимость.

Путь к целевому каталогу данных, доступному для чтения, передаётся в аргументе `-D` или в переменной окружения `PGDATA`.

Эта проверка будет полноценной, только если данные кластера и сервер рассчитаны на одну архитектуру процессора и одинаковый порядок байтов.

`-S`
`--server-compatibility-params`

Вывести все параметры указанного или текущего сервера, которые могут повлиять на совместимость.

Если параметр `-B` опущен, подразумевается текущий сервер.

Переменные окружения

`PGDATA`

Каталог размещения данных кластера по умолчанию

`PG_COLOR`

Выбирает вариант использования цвета в диагностических сообщениях. Возможные значения: `always` (всегда), `auto` (автоматически) и `never` (никогда).

См. также

pg_controldata

pg_integrity_check

pg_integrity_check — вычисление и проверка контрольных сумм отслеживаемых файлов (только сертифицированная редакция)

Синтаксис

`pg_integrity_check` [*`параметр-подключения`*...] [-s | --system] [-u | --user] [-c | --catalog] [-o | --output] [-l *`filename`* | --log=*`filename`*] [--syslog] [-D *`каталог_данных`*] [-C *`filename`*] [-v | --verbose] [-? | --help]

Описание

`pg_integrity_check` — утилита, поставляемая в составе Postgres Pro Standard, которая может вычислять и проверять контрольные суммы важных объектов. Запуская `pg_integrity_check`, вы должны указать минимум один из параметров, определяющих тип отслеживаемых объектов: `-s`, `-u` или `-c` для неизменяемых файлов, дополнительных файлов или таблиц системных каталогов, соответственно.

Если вы используете ключ `-o`, `pg_integrity_check` вычисляет контрольные суммы и записывает их в файлы конфигурации в каталоге `share/security/`. Для выполнения этой команды у вас должно быть право записи в эти файлы. Ключ `-o` нельзя использовать вместе с `-s`, так как контрольные суммы неизменяемых файлов не могут перезаписываться.

Если вы не указываете ключ `-o`, `pg_integrity_check` сравнивает вычисленные контрольные суммы с соответствующими контрольными суммами в файлах конфигурации. Если контрольные суммы каких-либо из отслеживаемых объектов не совпадают, `pg_integrity_check` выводит сообщения о различиях.

Более подробно использование `pg_integrity_check` описано в Разделе 30.2.

Параметры

`параметры-подключения`

Стандартные параметры для подключения к базе данных: `-d`, `-h`, `-p`, `-U`. Вы должны указать параметры `-d` и `-U` для проверки контрольных сумм таблиц каталогов с ключом `-c`. За подробным описанием параметров подключения обратитесь к psql.

`-s`
`--system`

Проверить контрольные суммы неизменяемых файлов. В контрольных суммах неизменяемых файлов учитывается и содержимое, и атрибуты этих файлов.

`-u`
`--user`

Проверить контрольные суммы дополнительных файлов. В контрольных суммах дополнительных файлов учитывается и содержимое, и атрибуты этих файлов.

`-c`
`--catalog`

Проверить контрольные суммы для таблиц системных каталогов. Чтобы ключ `-c` работал корректно, вы также должны задать параметры подключения к базе данных. Сервер баз данных должен быть запущен и готов принимать подключения.

`-o`
`--output`

Пересчитать контрольные суммы и записать их в файл.

`-l` *`filename`*
`--log=`*`filename`*

Записывать результаты проверки контрольных сумм в файл журнала.

`--syslog`

Записывать результаты проверки контрольных сумм в syslog.

`-D` *`каталог_данных`*

Каталог данных кластера БД. Этот параметр необходим для определения имён файлов конфигурации при использовании ключа `-u`.

`-C` *`filename`*

Абсолютный путь к файлу конфигурации, который используется для проверки целостности таблиц системного каталога выбранной базы данных. При использовании с параметром `-o` `pg_integrity_check` записывает контрольные суммы в указанный файл. Если другие параметры не используются, `pg_integrity_check` проверяет данные по указанному файлу.

`-v`
`--verbose`

Выводить отладочную информацию, в том числе значения контрольных сумм.

`-?`
`--help`

Вывести справку по командной строке.

Возвращаемые значения

0 — контрольные суммы вычислены или проверены успешно.

1 — при изначальном подсчёте контрольных сумм неизменяемых файлов произошла ошибка.

2 — проверка контрольных сумм выявила изменения в одном или нескольких отслеживаемых объектах.

3 — при проверке контрольных сумм произошла неожиданная ошибка.

Примеры

Вычисление контрольных сумм для дополнительных файлов и внесение их в файл конфигурации `share/security/_var_lib_pgpro_std-17_data.user.conf`:

```
pg_integrity_check -u -o -D /var/lib/pgpro/std-17/data
```

Проверка целостности всех отслеживаемых объектов в базе данных `postgres` от имени пользователя `postgres`:

```
pg_integrity_check -s -u -c -D /var/lib/pgpro/std-17/data -d postgres -h localhost -p
 5432 -U postgres
```

I.4. Сторонние серверные приложения

В этом разделе рассматриваются сторонние серверные приложения. Обычно они запускаются на узле с сервером баз данных.

pgbouncer

pgbouncer — пул соединений Postgres Pro

Синтаксис

В системах Linux:

`pgbouncer` [-d] [-R] [-v] [-u *пользователь*] pgbouncer.ini

`pgbouncer` -V | -h

В Windows:

`pgbouncer` [-v] [-u *пользователь*] pgbouncer.ini

`pgbouncer` -V | -h

Для использования pgbouncer в виде службы Windows есть дополнительные аргументы:

`pgbouncer.exe` --regservice pgbouncer.ini

`pgbouncer.exe` --unregservice pgbouncer.ini

Описание

pgbouncer — это программа, управляющая пулом соединений Postgres Pro. Любое конечное приложение может подключиться к pgbouncer, как если бы это был непосредственно сервер Postgres Pro, и pgbouncer создаст подключение к реальному серверу, либо задействует одно из ранее установленных подключений.

Предназначение pgbouncer — минимизировать издержки, связанные с установлением новых подключений к Postgres Pro.

Чтобы не нарушать семантику транзакций при переключении подключений, pgbouncer поддерживает несколько видов пулов:

Пул сеансов

Наиболее корректный метод. Когда клиент подключается, ему назначается одно серверное подключение на всё время, пока клиент остаётся подключённым. Когда клиент отключается, это подключение к серверу возвращается в пул. Этот метод работает по умолчанию.

Пул транзакций

Подключение к серверу назначается клиенту только на время транзакции. Когда pgbouncer замечает, что транзакция завершена, это подключение возвращается в пул.

Пул операторов

Наиболее агрессивный метод. Подключение к серверу будет возвращаться в пул сразу после завершения каждого запроса. Транзакции с несколькими операторами в этом режиме запрещаются, так как они не будут работать.

Административный интерфейс pgbouncer состоит из нескольких новых команд `SHOW`, доступных при подключении к специальной «виртуальной» базе данных `pgbouncer`.

Быстрый запуск

pgbouncer поставляется вместе с Postgres Pro Standard в виде отдельного пакета `pgbouncer` (подробные инструкции по установке приведены в Главе 16). Базовая настройка и использование демонстрируются ниже.

1. Создайте файл `pgbouncer.ini`. Подробнее он описывается на странице man `pgbouncer(5)`. Например:

   ```
   [databases]
   template1 = host=localhost dbname=template1 auth_user=someuser

   [pgbouncer]
   listen_port = 6432
   listen_addr = localhost
   auth_type = md5
   auth_file = userlist.txt
   logfile = pgbouncer.log
   pidfile = pgbouncer.pid
   admin_users = someuser
   ```

2. Создайте файл `userlist.txt` со списком пользователей, которым разрешено подключение:

   ```
   "некоторый_пользователь" "его_пароль_на_сервере"
   ```

3. Запустите pgbouncer:

   ```
   $ pgbouncer -d pgbouncer.ini
   ```

   ### Примечание

   Эта команда не работает в системах Windows. Вместо этого pgbouncer должен запускаться в виде службы, которую необходимо сначала зарегистрировать следующим образом:

   ```
   pgbouncer --regservice
   ```

4. Сделайте так, чтобы ваше приложение (или клиент `psql`) подключалось к pgbouncer, а не к серверу Postgres Pro непосредственно:

   ```
   $ psql -p 6432 -U someuser template1
   ```

5. Для управления pgbouncer подключитесь к специальной административной базе данных pgbouncer и выполните `SHOW HELP;`, чтобы ознакомиться с его справкой:

   ```
   $ psql -p 6432 -U someuser pgbouncer
   pgbouncer=# SHOW HELP;
   NOTICE:  Console usage
   DETAIL:
     SHOW [HELP|CONFIG|DATABASES|FDS|POOLS|CLIENTS|SERVERS|SOCKETS|LISTS|VERSION|...]
     SET key = arg
     RELOAD
     PAUSE
     SUSPEND
     RESUME
     SHUTDOWN
     [...]
   ```

6. Если вы вносили изменения в файл `pgbouncer.ini`, его можно перезагрузить командой:

   ```
   pgbouncer=# RELOAD;
   ```

Параметры

`-d, --daemon`

Запустить в фоновом режиме. Без этого указания процесс будет работать на переднем плане. При запуске в фоновом режиме необходимо явное указание параметра `pidfile`, а также `logfile` или `syslog`. В фоновом режиме сообщения не будут записываться в stderr.

Примечание

Это не работает в Windows, там pgbouncer нужно запускать в виде службы.

`-R, --reboot`

Примечание

Этот параметр устарел. Вместо него следует применять последовательный перезапуск с несколькими процессами pgbouncer с одним и тем же принимающим портом, используя `so_reuseport`.

Выполнить перезапуск на «лету». При этом pgbouncer подключается к работающему процессу, забирает у него открытые сокеты и начинает использовать их. Если активного процесса нет, он запускается в обычном режиме.

Примечание

Это работает, только если ОС поддерживает сокеты Unix и в конфигурации определён параметр `unix_socket_dir`. Не работает в Windows. Также при этом не поддерживаются подключения TLS, они сбрасываются.

`-u` *`пользователь`*`, --user` *`пользователь`*

Переключиться на заданного пользователя при запуске.

`-v, --verbose`

Увеличить детализацию вывода. Может использоваться неоднократно.

`-q, --quiet`

Приглушить вывод: не выводить ничего в stderr. Это не влияет на уровень выводимых сообщений, а только отключает вывод в stderr. Предназначено для применения в скриптах `init.d`.

`-V, --version`

Вывести версию.

`-h, --help`

Вывести короткую справку.

`--regservice`

Win32: Зарегистрировать pgbouncer в качестве службы Windows. Имя службы будет определяться значением параметра конфигурации `service_name`.

`--unregservice`

Win32: Разрегистрировать службу Windows.

Административная консоль

Эта консоль доступна при обычном подключении к базе pgbouncer:

```
$ psql -p 6432 pgbouncer
```

Подключаться к этой консоли разрешено только пользователям, перечисленным в параметрах конфигурации `admin_users` или `stats_users`. (За исключением режима `auth_mode=any`, когда в качестве `stats_user` может подключиться любой пользователь.)

Кроме того, пользователю `pgbouncer` разрешено подключение без пароля, если это подключение устанавливается через сокет Unix и у клиента тот же uid пользователя Unix, что и у работающего процесса.

Административная консоль в настоящее время поддерживает только простой протокол запросов. Некоторые драйверы используют расширенный протокол запросов для всех команд; эти драйверы не будут работать в данном случае.

Команды вывода информации

Команды `SHOW` выводят полезную информацию. Каждая команда описана ниже.

SHOW STATS

Выводит статистику. Выводимые этой и связанными командами суммарные значения, накапливаемые с момента запуска pgbouncer, и средние значения обновляются с периодичностью `stats_period`.

`database`

База данных, для которой представлена статистика.

`total_xact_count`

Общее число транзакций SQL, прошедших через pgbouncer.

`total_query_count`

Общее число SQL-запросов, прошедших через pgbouncer.

`total_received`

Общий объём сетевого трафика (в байтах), который получил pgbouncer.

`total_sent`

Общий объём сетевого трафика (в байтах), который передал pgbouncer.

`total_xact_time`

Общее время (в микросекундах), в течение которого pgbouncer использовал подключения к Postgres Pro, выполняя запросы или простаивая.

`total_query_time`

Общее время (в микросекундах), в течение которого pgbouncer активно использовал подключения к Postgres Pro, выполняя запросы.

`total_wait_time`

Среднее время ожидания ответов сервера в течение секунды (в микросекундах). Сбрасывается, когда клиентскому подключению назначается серверное подключение.

`avg_xact_count`

Среднее число транзакций в секунду за последний период статистики.

`avg_query_count`

Среднее число запросов в секунду за последний период статистики.

`avg_recv`

Средняя скорость получения данных от клиентов (байт в секунду).

`avg_sent`

Средняя скорость передачи данных клиентам (байт в секунду).

`avg_xact_time`

Средняя длительность транзакции (в микросекундах).

`avg_query_time`

Средняя длительность запроса (в микросекундах).

`avg_wait_time`

Среднее время ожидания ответов сервера клиентами, для которых было назначено серверное подключение в текущем интервале `stats_period`, в микросекундах (в среднем в секунду за этот интервал).

SHOW STATS_TOTALS

Выводит подмножество результатов `SHOW STATS`, включающее только суммарные значения (`total_`).

SHOW STATS_AVERAGES

Выводит подмножество результатов `SHOW STATS`, включающее только средние значения (`avg_`).

SHOW TOTALS

Выводит ту же статистику, что `SHOW STATS`, но по всем базам данных в целом.

SHOW SERVERS

`type`

«S» для серверов

`user`

Имя пользователя, с которым pgbouncer подключается к серверу.

`database`

Имя базы данных.

`state`

Состояние подключения pgbouncer к серверу, один из вариантов: `active`, `idle`, `used`, `tested`, `new`, `active_cancel` или `being_canceled`.

`addr`

IP-адрес сервера Postgres Pro.

`port`

Порт сервера Postgres Pro.

`local_addr`

Исходный адрес подключения на локальной машине.

`local_port`

Исходный порт подключения на локальной машине.

`connect_time`

Время установления подключения.

`request_time`

Время выдачи последнего запроса.

`wait`

Не используется для серверных подключений.

`wait_us`

Не используется для серверных подключений.

`close_needed`

1, если соединение будет закрыто при ближайшей возможности в связи с выполнением команды `RECONNECT` либо в связи с изменением параметров соединения, вызванным перезагрузкой файла конфигурации или изменениями в DNS.

`ptr`

Адрес внутреннего объекта для данного подключения. Используется как уникальный идентификатор.

`link`

Адрес клиентского подключения, с которым связан сервер.

`remote_pid`

Идентификатор обслуживающего серверного процесса (PID). Когда подключение выполняется через сокет Unix и ОС может выдать PID процесса, это PID, полученный от ОС. В противном случае этот идентификатор извлекается из пакета для отмены, переданного сервером; это будет интересующий PID при подключении к серверу Postgres Pro, но если подключение обслуживает другой pgbouncer, это будет случайное число.

`tls`

Информация о TLS-подключении; пустая строка, если TLS не используется.

`application_name`

Строка, содержащая параметр `application_name`, заданный для связанного клиентского подключения, или пустая строка, если он не задан или если подключение отсутствует.

SHOW CLIENTS

`type`

«C» для клиентов.

`user`

Пользователь, подключённый со стороны клиента.

`database`

Имя базы данных.

`state`

Состояние клиентского подключения: `active`, `waiting`, `active_cancel_req` или `waiting_cancel_req`.

`addr`

IP-адрес клиента.

`port`

Порт клиента.

`local_addr`

Конечный адрес подключения на локальной машине.

`local_port`

Конечный порт подключения на локальной машине.

`connect_time`

Время установления подключения.

`request_time`

Время последнего запроса клиента.

`wait`

Текущая длительность ожидания (в секундах).

`wait_us`

Дробная часть текущей длительности ожидания (в микросекундах).

`close_needed`

Не используется для клиентов.

`ptr`

Адрес внутреннего объекта для данного подключения. Используется как уникальный идентификатор.

`link`

Адрес серверного подключения, с которым связан клиент.

`remote_pid`

Идентификатор процесса (PID), в случае, если клиент подключается через сокет UNIX и ОС может выдать этот идентификатор.

`tls`

Информация о TLS-подключении; пустая строка, если TLS не используется.

`application_name`

Строка, содержащая параметр `application_name`, заданный клиентом для данного подключения, или пустая строка, если он не задан или если подключение отсутствует.

SHOW POOLS

Новый пул создаётся для каждой пары сущностей (база данных, пользователь).

`database`

Имя базы данных.

`user`

Имя пользователя.

`cl_active`

Число клиентских подключений, которые либо связаны с подключениями к серверу, либо простаивают без запросов, ожидающих обработки.

`cl_waiting`

Число клиентских подключений, которые отправили запросы, но ещё не получили подключения к серверу.

`cl_active_cancel_req`

Число клиентских подключений, которые передали серверу команды отмены запроса и ожидают ответ сервера.

`cl_waiting_cancel_req`

Число клиентских подключений, для которых серверу в данный момент передаются команды отмены запросов.

`sv_active`

Число серверных подключений, связанных с клиентами.

`sv_active_cancel`

Число серверных подключений, для которых в данный момент передаются команды отмены запросов.

`sv_being_canceled`

Серверы, которые должны простаивать, но ждут, пока не будут завершены все текущие команды отмены запросов, отправленные этим серверам.

`sv_idle`

Число серверных подключений, которые не используются и могут немедленно задействоваться для запросов клиентов.

`sv_used`

Число серверных подключений, которые простаивают дольше `server_check_delay`, поэтому перед повторным использованием для них нужно выполнять `server_check_query`.

`sv_tested`

Число серверных подключений, для которых в данный момент выполняются запросы `server_reset_query` или `server_check_query`.

`sv_login`

Число серверных подключений, через которые в данный момент выполняется вход на сервер.

`maxwait`

Показывает, как долго ожидает в очереди самый первый клиент (в секундах). Если это число начинает увеличиваться, значит текущий пул серверов не справляется с запросами достаточно быстро. Причиной тому может быть перегруженный сервер, либо просто слишком маленький размер пула (параметр `pool_size`).

`maxwait_us`

Дробная часть максимального времени ожидания (в микросекундах).

`pool_mode`

Действующий режим пула.

SHOW PEER_POOLS

Для каждого настроенного узла создаётся новая запись `peer_pool`.

`database`

Идентификатор настроенной записи узла.

`cl_active_cancel_req`

Число клиентских подключений, которые передали серверу команды отмены запроса и ожидают ответ сервера.

`cl_waiting_cancel_req`

Число клиентских подключений, для которых серверу в данный момент передаются команды отмены запросов.

`sv_active_cancel`

Число серверных подключений, для которых в данный момент передаются команды отмены запросов.

`sv_login`

Число серверных подключений, через которые в данный момент выполняется вход на сервер.

SHOW LISTS

Показывает следующие внутренние сведения, в столбцах (не строках):

`databases`

Число баз данных.

`users`

Число пользователей.

`pools`

Число пулов.

`free_clients`

Число свободных клиентов.

`used_clients`

Число активных клиентов.

`login_clients`

Число клиентов в состоянии входа (`login`).

`free_servers`

Число свободных серверов.

`used_servers`

Число задействованных серверов.

`dns_names`

Количество имён DNS в кеше.

`dns_zones`

Количество зон DNS в кеше.

`dns_queries`

Количество выполняющихся запросов к DNS.

`dns_pending`

Не используется.

SHOW USERS

`name`

Имя пользователя.

`pool_mode`

Переопределение `pool_mode` для пользователя либо `NULL`, если должен использоваться режим по умолчанию.

SHOW DATABASES

`name`

Имя настроенной записи базы данных.

`host`

Узел, к которому подключается pgbouncer.

`port`

Порт, к которому подключается pgbouncer.

`database`

Реальное имя базы данных, к которой подключается pgbouncer.

`force_user`

Когда пользователь указан в строке соединения, подключение между pgbouncer и Postgres Pro должно устанавливаться от его имени, вне зависимости от пользователя на стороне клиента.

`pool_size`

Максимальное число серверных подключений.

`min_pool_size`

Минимальное число серверных подключений.

`reserve_pool`

Максимальное число дополнительных подключений для этой базы данных.

`pool_mode`

Переопределение pool_mode для базы данных либо `NULL`, если должен использоваться режим по умолчанию.

`max_connections`

Максимально возможное число подключений для этой базы, установленное либо глобально, параметром `max_db_connections`, либо на уровне базы.

`current_connections`

Текущее число подключений для этой базы.

`paused`

1, если база данных находится в состоянии паузы, иначе — 0.

`disabled`

1, если база данных находится в отключённом состоянии, иначе — 0.

SHOW PEERS

`peer_id`

Идентификатор настроенной записи узла.

`host`

Узел, к которому подключается pgbouncer.

`port`

Порт, к которому подключается pgbouncer.

`pool_size`

Максимальное число серверных подключений, которое можно установить для этого узла.

SHOW FDS

Внутренняя команда, которая показывает список файловых дескрипторов и их внутреннее состояние.

При подключении пользователя с именем `pgbouncer` через сокет Unix из процесса с UID, совпадающим с UID текущего процесса, ему передаются реальные файловые дескрипторы. Это применяется для выполнения перезагрузки «на лету».

Примечание

В Windows это не работает.

Эта команда также блокирует внутренний цикл событий, так что её не следует выполнять, когда pgbouncer используется.

`fd`

Числовое значение файлового дескриптора (ФД).

`task`

Предназначение; возможны следующие варианты: `pooler`, `client` или `server`.

`user`

Пользователь подключения, занимающего этот ФД.

`database`

База данных подключения, занимающего этот ФД.

`addr`

IP-адрес подключения, занимающего данный ФД; `unix`, если это сокет Unix.

`port`

Порт подключения, занимающего данный ФД.

`cancel`

Ключ отмены для данного подключения.

`link`

Файловый дескриптор ответной стороны сервера/клиента. `NULL`, если подключение простаивает.

SHOW SOCKETS, SHOW ACTIVE_SOCKETS

Выводит низкоуровневую информацию обо всех или только активных сокетах. В вывод включается информация, выводимая командами `SHOW CLIENTS` и `SHOW SERVERS`, а также дополнительные сведения низкого уровня.

SHOW CONFIG

Выводит текущие параметры конфигурации, по одному в строке со следующими столбцами:

`key`

Имя переменной конфигурации.

`value`

Значение переменной конфигурации.

`default`

Значение по умолчанию.

`changeable`

Значение `yes` или `no`, показывающее, можно ли изменить переменную во время работы. Если значение `no`, переменная может быть изменена только при перезагрузке. Чтобы изменить переменную во время работы, воспользуйтесь командой `SET`.

SHOW MEM

Выводит низкоуровневую информацию о размере различных блоков памяти, выделенных для внутреннего использования. Эта информация имеет динамическую природу и может меняться.

SHOW DNS_HOSTS

Выводит имена узлов, содержащиеся в кеше DNS.

`hostname`

Имя узла.

`ttl`

Сколько секунд остаётся до очередного внешнего поиска.

`addrs`

Список адресов, разделённых запятыми.

SHOW DNS_ZONES

Показывает зоны DNS в кеше.

`zonename`

Имя зоны.

`serial`

Текущий серийный номер.

`count`

Имена узлов, относящихся к этой зоне.

SHOW VERSION

Показывает строку с версией pgbouncer.

SHOW STATE

Показывает параметры состояния pgbouncer. Текущие состояния: активное, состояние паузы и приостановленное.

Команды управления процессом

PAUSE [*бд*]

pgbouncer пытается отключиться от всех серверов. При закрытии каждого подключения к серверу ожидается освобождение этого подключения в соответствии с режимом пула серверов (в режиме пула транзакций должна завершиться транзакция, в режиме пула операторов должен завершиться оператор, а в режиме пула сеансов должен отключиться клиент). Выполнение этой команды заканчивается, только когда отключатся все подключения. Эта команда должна применяться во время перезапуска базы данных.

Если указано имя базы данных, будет приостановлена работа только с ней.

Новые подключения клиентов к базе, поставленной на паузу, будут находиться в состоянии ожидания, пока не будет выполнена команда `RESUME`.

DISABLE *бд*

Запрещает любые новые подключения клиентов к указанной базе данных.

ENABLE *бд*

Разрешает новые подключения клиентов после предыдущей команды `DISABLE`.

RECONNECT *бд*

Закрывает все открытые подключения к указанной базе (или ко всем базам) по мере их освобождения в соответствии с режимом пула, не дожидаясь окончания времени жизни подключений. При этом немедленно могут быть установлены новые подключения к серверу, согласно текущим параметрам пула.

Эта команда может быть полезна, когда изменяется конфигурация подключения к серверу, например, нужно постепенно переключиться на новый сервер. Эту команду не нужно выполнять, если вы меняете строку соединения в `pgbouncer.ini` и перезагружаете конфигурацию (см. `RELOAD`) или когда меняются адреса в DNS, так как равнозначное действие будет выполнено автоматически. Эта команда может быть полезна, только если подключения pgbouncer маршрутизируются какими-то внешними средствами.

После выполнения этой команды возможен длительный период, когда будут существовать и старые подключения, и новые. Это может иметь значение только при переключении трафика между читающими репликами или при переключении между узлами в конфигурации с несколькими ведущими. Если необходимо переключить все подключения сразу, рекомендуется использовать команду `PAUSE`. Чтобы закрыть подключения экстренно (например, когда произвести переключение нужно не постепенно, а срочно), вы можете воспользоваться командой `KILL`.

KILL *бд*

Немедленно закрывает все клиентские и серверные подключения к указанной базе данных, деактивируя её.

Новые подключения клиентов к деактивированной базе будут находиться в состоянии ожидания, пока не будет выполнена команда `RESUME`.

SUSPEND

Все буферы сокетов очищаются и pgbouncer прекращает принимать данные через них. Выполнение этой команды заканчивается, только когда все буферы будут очищены. Эта команда должна применяться, когда pgbouncer перезагружается «на лету».

Новые подключения клиентов к приостановленной базе будут находиться в состоянии ожидания, пока не будет выполнена команда `RESUME`.

RESUME [*бд*]

Восстанавливает работу после предыдущей команды `KILL`, `PAUSE` или `SUSPEND`.

SHUTDOWN

Приводит к завершению процесса pgbouncer.

RELOAD

Указывает процессу pgbouncer перезагрузить файлы конфигурации и обновить значения изменяемых параметров, а именно основной файл конфигурации, а также файлы, указанные в параметрах `auth_file` и `auth_hba_file`.

pgbouncer отслеживает изменения в файле конфигурации, затрагивающие подключения к базе данных. Существующие подключения со старыми параметрами будут закрыты при ближайшем освобождении этих подключений (в соответствии с режимом пула), а новые подключения к серверу немедленно начнут использовать изменившиеся параметры соединения.

WAIT_CLOSE [*бд*]

Ожидает, пока для всех серверных подключений к указанной базе (или ко всем базам) не сбросится признак `close_needed` (см. Подраздел «SHOW SERVERS»). Эту команду можно вызвать после `RECONNECT` или `RELOAD`, чтобы, например, в скриптах переключения узлов, дождаться применения изменений конфигураций в полном объёме.

Другие команды

SET *ключ* = *аргумент*

Изменяет параметр конфигурации (см. также Подраздел «SHOW CONFIG»). Например:

```
SET log_connections = 1;
SET server_check_query = 'select 2';
```

(Заметьте, что эта команда выполняется в консоли администратора pgbouncer и задаёт параметры pgbouncer. Команда `SET`, выполненная в другой базе данных, будет передана на выполнению серверу Postgres Pro, как и любая другая SQL-команда.)

Сигналы

`SIGHUP`

Перезагрузка конфигурации. Равносильно выполнению команды `RELOAD` в консоли.

`SIGINT`

Безопасное отключение. Равносильно выполнению команд `PAUSE` и `SHUTDOWN` в консоли.

`SIGTERM`

Немедленное отключение. Равносильно выполнению команды `SHUTDOWN` в консоли.

`SIGUSR1`

Равносильно выполнению команды `PAUSE` в консоли.

`SIGUSR2`

Равносильно выполнению команды `PAUSE` в консоли.

Параметры libevent

Из документации libevent:

> Поддержку `epoll`, `kqueue`, `devpoll`, `poll` или `select` можно отключить, установив переменную окружения `EVENT_NOEPOLL`, `EVENT_NOKQUEUE`, `EVENT_NODEVPOLL`, `EVENT_NOPOLL` или `EVENT_NOSELECT`, соответственно.
>
> Если установить переменную окружения `EVENT_SHOW_METHOD`, `libevent` покажет текущий выбранный метод уведомлений в ядре.

Файл конфигурации `pgbouncer.ini`

Файл конфигурации имеет формат `ini`-файла. Названия разделов записываются между `[` и `]`. Строки, начинающиеся с `;` или `#`, считаются комментариями и игнорируются. Символы `;` и `#`, встречающиеся не в начале строки, не распознаются.

Общие параметры

`logfile`

Указывает имя файла журнала. Для работы в режиме демона (`-d`) должен быть задан либо этот параметр, либо `syslog`. Файл журнала остаётся открытым, так что после смены имени для прокрутки следует выполнить `kill -HUP` или `RELOAD;` в консоли. В Windows необходимо остановить и вновь запустить службу.

Обратите внимание, что параметр `logfile` сам по себе не отключает вывод сообщений в stderr. Отключить его можно, воспользовавшись параметром командной строки `-q` или `-d`.

По умолчанию: не задано.

`pidfile`

Указывает имя файла PID. Без этого указания работа в режиме демона (`-d`) не допускается.

По умолчанию: не задано.

`listen_addr`

Указывает список адресов (через запятую), по которым должны приниматься TCP-подключения. Вы можете указать `*`, что будет означать «принимать по всем адресам». Когда этот параметр не задан, принимаются только подключения через Unix-сокеты.

Адреса могут задаваться числами (в формате IPv4/IPv6) или именами.

По умолчанию: не задано.

`listen_port`

Номер принимающего порта. Задаётся и для сокетов TCP, и для Unix-сокетов.

По умолчанию: 6432

`unix_socket_dir`

Указывает расположение сокетов Unix. Действует и для принимающего сокета, и для подключений к серверу. Если задана пустая строка, сокеты Unix отключаются. При указании значения, начинающегося с `@`, будет создан сокет Unix в абстрактном пространстве имён (в настоящее время поддерживается в Linux и Windows).

Чтобы работала перезагрузка «на лету» (`-R`), необходимо настроить сокет Unix, который должен находиться в пространстве имён файловой системы.

По умолчанию: `/tmp` (в Windows — пустая строка) `unix_socket_mode`

Режим файловой системы для сокета Unix. Игнорируется для сокетов в абстрактном пространстве имён. Не поддерживается в Windows.

По умолчанию: 0777

`unix_socket_group`

Имя группы для сокета Unix. Игнорируется для сокетов в абстрактном пространстве имён. Не поддерживается в Windows.

По умолчанию: не задано.

`user`

Если задан, определяет, на какого пользователя Unix нужно переключиться после запуска. Работает, только если pgbouncer запускается от имени root или уже запущен от имени заданного пользователя.

Не поддерживается в Windows.

По умолчанию: не задано.

`pool_mode`

Указывает, когда подключение к серверу могут повторно использовать другие клиенты.

`session`

Сервер возвращается в пул после отключения клиента. Это вариант по умолчанию.

`transaction`

Сервер возвращается в пул после завершения транзакции.

`statement`

Сервер возвращается в пул после завершения запроса. В этом режиме не допускаются транзакции, охватывающие несколько операторов.

`max_client_conn`

Максимально допустимое число клиентских подключений.

При увеличении должно также увеличиваться ограничение на число файловых дескрипторов. Заметьте, что фактическое число занятых файловых дескрипторов будет больше, чем `max_client_conn`. Если каждый пользователь подключается к серверу под своим именем, теоретически возможный максимум равен:

```
max_client_conn + (max pool_size * total databases * total users)
```

Если пользователь задан в строке соединения (все пользователи подключаются под одним именем), теоретический максимум равен:

```
max_client_conn + (max pool_size * total databases)
```

Теоретический максимум не должен достигаться никогда, если только кто-то намеренно не предпримет специальные меры для этого. Тем не менее это значит, что число файловых дескрипторов должно ограничиваться довольно большим числом.

Поищите `ulimit` в руководстве man в вашей системе. Замечание: ограничение `ulimit` неприменимо в среде Windows.

По умолчанию: 100

`default_pool_size`

Сколько подключений к серверу возможно для пары пользователь/база. Может быть переопределено в конфигурации базы данных.

По умолчанию: 20

`min_pool_size`

Добавить в пул дополнительные соединения, если число активных подключений меньше этого числа. Даёт положительный эффект, когда нагрузка появляется внезапно после периода простоя. Это значение будет ограничено размером пула.

По умолчанию: 0 (отключено)

`reserve_pool_size`

Число дополнительно разрешённых подключений в пуле (см. `reserve_pool_timeout`). При 0 резерв отсутствует.

По умолчанию: 0 (отключено)

`reserve_pool_timeout`

Если клиент не обслуживается заданный период времени (в секундах), pgbouncer задействует дополнительные подключения из резервного пула. При 0 это не происходит.

По умолчанию: 5.0

`max_db_connections`

Не допускать больше заданного числа серверных подключений к базе данных (независимо от пользователя). При этом учитывается база данных pgbouncer, к которой подключён клиент, а не база данных Postgres Pro исходящего подключения. Данный предел можно установить для каждой базы данных в разделе `[databases]`.

Обратите внимание, что когда предел достигается, закрытие подключения клиента в одном пуле не позволяет немедленно установить другое подключение к серверу через другой пул, так как подключение первого пула по-прежнему открыто. Когда сервер закроет его (по тайм-ауту неактивности), новое подключение будет немедленно установлено для ожидающего пула.

По умолчанию: 0 (нет ограничения)

`max_user_connections`

Не допускать больше заданного числа подключений к серверу для пользователя (независимо от базы данных). При этом учитывается пользователь pgbouncer, связанный с пулом, — это может быть либо пользователь, указанный для подключения к серверу, либо, в отсутствие этого указания, пользователь, от имени которого подключился клиент. Этот предел можно установить для каждого пользователя в разделе `[users]`.

Обратите внимание, что когда предел достигается, закрытие подключения клиента в одном пуле не позволяет немедленно установить другое подключение к серверу через другой пул, так как подключение первого пула по-прежнему открыто. Когда сервер закроет его (по тайм-ауту неактивности), новое подключение будет немедленно установлено для ожидающего пула.

По умолчанию: 0 (нет ограничения)

`server_round_robin`

По умолчанию pgbouncer повторно использует подключения сервера в порядке LIFO (последнее пришло, первое ушло), так что основная загрузка распределяется по нескольким последним соединениям. Это даёт наибольшую производительность, если базу данных обслуживает один сервер. Но если за адресом (TCP, DNS или списком хостов) базы данных скрывается система балансировщика, лучше, если pgbouncer будет использовать подключения в данном режиме, обеспечивая таким образом равномерную нагрузку.

По умолчанию: 0

`track_extra_parameters`

По умолчанию pgbouncer отслеживает параметры `client_encoding`, `datestyle`, `timezone`, `standard_conforming_strings` и `application_name` для каждого клиента. Чтобы разрешить отслеживание других параметров, их можно задать в этой переменной. В таком случае pgbouncer будет хранить их в кеше переменных клиента и восстанавливать на сервере каждый раз, когда клиент становится активным.

Если нужно указать несколько значений, используется перечисление через запятую (например, `default_transaction_readonly, IntervalStyle`).

Примечание

Большинство параметров нельзя отслеживать таким образом. Можно отслеживать только те параметры, которые Postgres Pro передаёт клиенту. В Postgres Pro есть официальный список параметров, которые передаются клиенту. Расширения Postgres Pro могут изменять этот список и сами добавлять передаваемые параметры, а также могут начать передавать уже существующие параметры, которые не передаются Postgres Pro. Например, Citus 12.0+ заставляет PostgreSQL также передавать `search_path`.

Протокол `postgres` позволяет настраивать параметры непосредственно в виде параметров в стартовом пакете или внутри поля `options` стартового пакета. Параметры, указанные любым из этих методов, поддерживаются переменной `track_extra_parameters`. Однако в `track_extra_parameters` можно включить только параметры, содержащиеся в поле `options`, но не само поле.

По умолчанию: `IntervalStyle`

`ignore_startup_parameters`

По умолчанию pgbouncer принимает только параметры, которые он может отслеживать в стартовых пакетах: `client_encoding`, `datestyle`, `timezone` и `standard_conforming_strings`.

Все другие параметры вызывают ошибку. Чтобы принимались и другие параметры, их нужно указать здесь, чтобы pgbouncer знал, что они обрабатываются администратором и их можно игнорировать.

Если нужно указать несколько значений, используется перечисление через запятую (например, `options,extra_float_digits`).

Протокол `postgres` позволяет настраивать параметры непосредственно в виде параметров в стартовом пакете или внутри поля `options` стартового пакета. Параметры, указанные любым из этих методов, поддерживаются переменной `ignore_startup_parameters`. В `ignore_startup_parameters` можно даже включить само поле `options`, в результате чего любые неизвестные параметры, содержащиеся в поле `options`, будут игнорироваться.

По умолчанию: пустая строка

`peer_id`

Идентификатор узла, используемый для определения этого процесса pgbouncer в группе одноранговых процессов pgbouncer. Значение `peer_id` должно быть уникальным в пределах группы одноранговых процессов pgbouncer. Если установлено значение 0, пиринг в pgbouncer отключается. За дополнительной информацией обратитесь к разделу [peers]. Максимальное значение, которое можно указывать для `peer_id`, — 16383.

По умолчанию: 0

`disable_pqexec`

Отключает простой протокол запросов (PQexec). В отличие от расширенного протокола запросов, этот протокол допускает указание нескольких запросов в одном пакете, что оставляет место для атак с SQL-инъекцией. Отключение этого протокола может улучшить безопасность. Разумеется, это означает, что при этом смогут работать только клиенты, которые используют исключительно расширенный протокол запросов.

По умолчанию: 0

`application_name_add_host`

Добавляет адрес узла и порт клиента к имени приложения, задаваемого при установлении подключения. Это помогает идентифицировать источник плохих запросов и т. п. Это выполняется, только когда подключение устанавливается. Если свойство `application_name` будет изменено позднее командой `SET`, pgbouncer его уже не поменяет.

По умолчанию: 0

`conffile`

Показывает расположение текущего файла конфигурации. При изменении этого параметра pgbouncer будет использовать другой файл конфигурации после команд `RELOAD` / `SIGHUP`.

По умолчанию: файл, заданный в командной строке

`service_name`

Используется при регистрации службы win32.

По умолчанию: `pgbouncer`

`job_name`

Псевдоним `service_name`.

`stats_period`

Определяет, с какой периодичностью (в секундах) будут пересчитываться средние значения, выводимые различными командами `SHOW`, и как часто агрегированная статистика будет записываться в журнал (но см. `log_stats`).

По умолчанию: 60

Параметры аутентификации

pgbouncer выполняет собственную аутентификацию клиентского приложения и имеет собственную базу данных пользователей. Для управления аутентификацией и данными пользователей используются следующие параметры.

`auth_type`

Определяет, как аутентифицировать пользователей.

`cert`

Клиент должен подключаться по соединению TLS с действительным клиентским сертификатом. Имя пользователя берётся из поля `CommonName` (Общее имя) сертификата.

`md5`

Применять проверку пароля по хешу MD5. Этот метод аутентификации выбирается по умолчанию. Файл `auth_file` может содержать как зашифрованные MD5, так и открытые пароли. Даже при выборе `md5`, если пароль пользователя задан для метода SCRAM, автоматически будет применяться проверка по алгоритму SCRAM.

`scram-sha-256`

Применять проверку пароля по алгоритму SCRAM-SHA-256. Заданный параметром `auth_file` файл должен содержать зашифрованные SCRAM или открытые пароли. Учтите, что зашифрованные SCRAM пароли могут использоваться только для проверки пароли клиентов, но не для входа на сервер. Чтобы использовать SCRAM для серверных подключений, пароли необходимо задать открытым текстом.

`plain`

По каналу передаётся пароль в открытом тексте. Устаревший вариант.

`trust`

Аутентификация не выполняется. Тем не менее имя пользователя должно присутствовать в `auth_file`.

`any`

Подобен методу `trust`, но переданное имя пользователя игнорируется. Требует, чтобы для всех баз данных было настроено подключение заданного пользователя. Кроме того, база данных консоли допускает подключение любого пользователя в качестве администратора.

`hba`

Фактический тип аутентификации загружается из `auth_hba_file`. Это позволяет применять разные методы аутентификации для разных вариантов доступа. Например: для подключений через сокет Unix применять метод `peer`, а для TCP — TLS.

`pam`

Для проверки подлинности пользователей используется инфраструктура PAM (Pluggable Authentication Modules, Подключаемые модули аутентификации). Файл `auth_file` игнорируется. Этот метод несовместим с базами данных, для которых используется `auth_user`. Инфраструктуре PAM в качестве имени службы передаётся `pgbouncer`. `pam` в файле конфигурации HBA не поддерживается.

`auth_hba_file`

Файл конфигурации HBA, который используется, когда параметр `auth_type` равен `hba`.

По умолчанию: не задано.

`auth_file`

Имя файла, из которого будут загружаться имена и пароли пользователей. За подробностями обратитесь к Подразделу «Формат файла аутентификации».

Для большинства типов аутентификации (см. `auth_type`) необходимо, чтобы был задан параметр `auth_file` или `auth_user`; в противном случае в системе не будет созданных пользователей.

По умолчанию: не задано.

`auth_user`

Если задан параметр `auth_user`, пользователи, не описанные в файле `auth_file`, будут проверяться запросом `auth_query` по таблице `pg_shadow` в базе данных `auth_user`. Пароль пользователя `auth_user` будет взят из файла `auth_file`. (Если для пользователя `auth_user` не требуется пароль, задавать его в `auth_file` не нужно.)

Для прямого доступа к `pg_shadow` требуются права администратора. Поэтому рекомендуется, чтобы обычный пользователь обращался к ней, вызывая функцию SECURITY DEFINER (с контекстом безопасности определившего).

По умолчанию: не задано.

`auth_query`

Запрос для извлечения пароля пользователя из базы данных.

Для прямого доступа к `pg_shadow` требуются права администратора. Поэтому рекомендуется, чтобы обычный пользователь обращался к ней, вызывая функцию SECURITY DEFINER (с контекстом безопасности определившего).

Заметьте, что этот запрос выполняется в целевой базе данных, так что если в нём используются функции, они должны быть установлены в каждой базе.

По умолчанию: `SELECT usename, passwd FROM pg_shadow WHERE usename=$1`

`auth_dbname`

Имя базы данных в разделе [databases], которое будет использоваться для аутентификации. Этот параметр может быть глобальным или переопределяемым в строке подключения.

Параметры журнала

`syslog`

Включает/отключает запись в syslog. В Windows вместо этого используется журнал событий.

По умолчанию: 0

`syslog_ident`

Имя, с которым события передаются в syslog.

По умолчанию: `pgbouncer` (имя программы)

`syslog_facility`

Субъект, который будет указываться в событиях, отправляемых в syslog. Возможные варианты: `auth`, `authpriv`, `daemon`, `user`, `local0-7`.

По умолчанию: `daemon`

`log_connections`

Фиксировать в журнале успешные подключения.

По умолчанию: 1

`log_disconnections`

Фиксировать отключения с указаниями их причин.

По умолчанию: 1

`log_pooler_errors`

Фиксировать сообщения об ошибках, которые pgbouncer передаёт клиентам.

По умолчанию: 1

`log_stats`

Записывать агрегированную статистику в журнал, с периодичностью `stats_period`. Это может быть лишним, если ту же статистику запрашивают внешние средства мониторинга, вызывая команды `SHOW`.

По умолчанию: 1

`verbose`

Увеличивает уровень детализации. Соответствует ключу `-v` в командной строке. Например, указание `-v -v` в командной строке равносильно записи `verbose=2` в конфигурации.

По умолчанию: 0

Управление доступом к консоли

`admin_users`

Разделённый запятыми список пользователей базы данных, которым разрешено подключаться к консоли и выполнять в ней любые команды. Игнорируется с `auth_type`, равным `any`, так как в этом случае любой пользователь может подключаться как администратор.

По умолчанию: пустая строка

`stats_users`

Разделённый запятыми список пользователей баз данных, которым разрешено подключаться к консоли и выполнять команды только на чтение, а именно все команды `SHOW`, за исключением `SHOW FDS`.

По умолчанию: пустая строка

Проверки активности соединений, тайм-ауты

`server_reset_query`

Запрос, посылаемый серверу при освобождении подключения, прежде чем оно станет доступно другим клиентам. В этот момент никакая транзакция не выполняется, так что значение не должно включать команды `ABORT` или `ROLLBACK`.

Предполагается, что этот запрос очистит все изменения в состоянии сеанса базы данных, чтобы следующий клиент получил подключение в определённом состоянии. По умолчанию выполняется команда `DISCARD ALL`, которая очищает всё, но при этом следующему клиенту не остаётся никакого кешированного состояния. Её можно поменять на более мягкую, например `DEALLOCATE ALL`, просто освобождающую подготовленные операторы (если работа приложения не нарушается, когда какое-то состояние сохраняется).

Когда применяется пул транзакций, `server_reset_query` не действует, так как в этом режиме клиенты не должны использовать никакие свойства сеансов, потому что каждая транзакция завершается в отдельном соединении и таким образом получает разные состояния сеанса.

По умолчанию: `DISCARD ALL`

`server_reset_query_always`

Определяет, должен ли запрос `server_reset_query` выполняться во всех режимах пула. Когда этот параметр отключён (по умолчанию), `server_reset_query` будет запускаться только в режиме пула сеансов. Соединениям в режиме пула транзакций не должен требоваться запрос сброса состояния.

Этот параметр предназначен для аномальных конфигураций, в которых приложения не вполне корректно используют свойства сеансов, подключаясь к пулу транзакций pgbouncer. Он позволяет сменить недетерминированное неправильное поведение на детерминированное (но всё же неправильное) — клиенты всегда теряют своё состояние после каждой транзакции.

По умолчанию: 0

`server_check_delay`

Определяет, как долго должны сохраняться освобождаемые подключения в состоянии готовности к повторному использованию, без запуска запроса `server_check_query`. При значении 0 этот запрос запускается всегда.

По умолчанию: 30.0

`server_check_query`

Простой холостой запрос, проверяющий, сохраняется ли подключение к серверу.

Если это пустая строка, проверка соединения отключается.

По умолчанию: `select 1`

`server_fast_close`

Определяет, должен ли сервер в режиме пула сеансов отключаться немедленно либо после завершения текущей транзакции, если он находится в режиме `close_needed` (который включается командами `RECONNECT` и `RELOAD` или при изменениях в DNS), или необходимо дожидаться завершения сеанса. Режимы пула запросов и транзакций так и работают, поэтому включение этого параметра на эти режимы не влияет.

Если этот параметр включён, то есть подключение к серверу закрывается до завершения сеанса клиента, клиентское подключение также закрывается. Тем самым гарантируется, что прерывание сеанса не останется незамеченным для клиента.

Этот параметр позволяет ускорить вступление в силу изменений параметров соединений в случае использования пула сеансов и существования долгоживущих сеансов. Недостатком его использования может быть то, что клиентские сеансы могут прерываться при изменении конфигурации, так что клиентские приложения должны уметь переподключаться и восстанавливать состояние сеанса. Но заметьте, что никакие транзакции при этом не будут потеряны, так как прерываться будут не выполняющиеся транзакции, а только лишь простаивающие сеансы.

По умолчанию: 0

`server_lifetime`

pgbouncer будет закрывать неиспользуемое серверное подключение (то есть не подключённое ни к одному клиентскому подключению), существующее дольше заданного времени (в секундах). Ноль означает, что соединение будет использоваться только один раз, а затем будет закрываться.

По умолчанию: 3600.0

`server_idle_timeout`

Если подключение к серверу простаивает дольше заданного времени (в секундах), оно будет закрыто. При значении 0 тайм-аут отключается.

По умолчанию: 600.0

`server_connect_timeout`

Если инициализация подключения и вход на сервер не завершается за указанное время (в секундах), соединение будет закрыто.

По умолчанию: 15.0

`server_login_retry`

Время ожидания повторной попытки подключения пулом соединений (в секундах), если вход на сервер завершился ошибкой из-за сбоя подключения или аутентификации. В течение интервала ожидания новые клиенты, пытающиеся подключиться к неисправному серверу, немедленно получат сообщение об ошибке без повторной попытки подключения.

Цель такого поведения заключается в том, чтобы избавить клиентов от необходимости ожидания в очереди, пока соединение с сервером станет доступным, если сервер не работает. Однако это также означает, что если сервер ненадолго станет недоступен, например, во время перезапуска или при ошибочной конфигурации, то пройдёт как минимум такой интервал времени, прежде чем пул соединений вновь допустит подключение к этому серверу. Во избежание таких ситуаций следует управлять запланированными событиями, такими как перезапуски, используя команду `PAUSE`.

По умолчанию: 15.0

`client_login_timeout`

Если клиент подключается, но не может пройти аутентификацию за указанное время (в секундах), он будет отключён. Требуется в основном, чтобы «мёртвые» подключения не задерживали операцию `SUSPEND` и, как следствие, перезагрузку «на лету».

По умолчанию: 60.0

`autodb_idle_timeout`

Если создаваемые автоматически (через «*») пулы баз данных не используются заданное время (в секундах), они освобождаются. Минусом этого является то, что их статистика также сбрасывается.

По умолчанию: 3600.0

`dns_max_ttl`

Время кеширования результатов поиска DNS (в секундах). Фактический DNS TTL игнорируется.

По умолчанию: 15.0

`dns_nxdomain_ttl`

Время кеширования ошибок DNS и результатов `NXDOMAIN` (в секундах).

По умолчанию: 15.0

`dns_zone_check_period`

Интервал проверки серийного номера зоны.

pgbouncer может собрать список зон DNS из имён узлов (всё после первой точки) и затем периодически проверять, не изменился ли серийный номер зоны. Если он меняется, то все имена узлов, относящиеся к зоне, разрешаются заново. Если для какого-либо узла получается другой IP-адрес, его подключения признаются недействительными.

По умолчанию: 0.0 (отключено)

`resolv_conf`

Расположение пользовательского файла `resolv.conf`, в котором можно указать адреса серверов DNS а также другие параметры разрешения имён, не зависящие от глобальной конфигурации операционной системы.

Обработку этого файла выполняет используемая библиотека DNS, а не pgbouncer, поэтому, чтобы ознакомиться с его синтаксисом и указаниями, обратитесь к документации этой библиотеки.

По умолчанию: не задан (используются параметры операционной системы)

Параметры TLS

`client_tls_sslmode`

Режим TLS, выбираемый для подключений клиентов. По умолчанию подключения с использованием TLS запрещены. Когда этот режим включается, необходимо также настроить в `client_tls_key_file` и `client_tls_cert_file` ключ и сертификат, которые будет использовать pgbouncer.

`disable`

Обычный TCP. Если клиент запрашивает TLS, его запрос игнорируется. Это режим по умолчанию.

`allow`

Если клиент запрашивает TLS, он используется. В противном случае используется обычный протокол TCP. Если клиент предоставляет свой сертификат, он не проверяется.

`prefer`

То же, что и `allow`.

`require`

Клиент должен использовать TLS. В противном случае соединение клиента сбрасывается. Если клиент предоставляет свой сертификат, он не проверяется.

`verify-ca`

Клиент должен использовать TLS с годным клиентским сертификатом.

`verify-full`

То же, что и `verify-ca`.

`client_tls_key_file`

Закрытый ключ pgbouncer, применяемый для шифрования клиентских подключений.

По умолчанию: не задано.

`client_tls_cert_file`

Сертификат частного ключа. Клиенты могут проверить его.

По умолчанию: не задано.

`client_tls_ca_file`

Файл с корневым сертификатом, по которому будут проверяться клиентские сертификаты.

По умолчанию: не задано.

`client_tls_protocols`

Определяет, какие версии протокола TLS разрешены. Допустимые значения: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Краткие обозначения: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (равнозначно all).

По умолчанию: `secure`

`client_tls_ciphers`

Определяет допустимые шифры TLS, в синтаксисе OpenSSL. Допустимые краткие значения: `default`/`secure`, `compat`/`legacy`, `insecure`/`all`, `normal`, `fast`.

Распространяется только на соединения, использующие TLS версии 1.2 и ниже. В настоящее время нет параметра, который бы управлял выбором шифров при использовании TLS версии 1.3.

По умолчанию: `fast`

`client_tls_ecdhcurve`

Имя эллиптической кривой, применяемой при обмене ключами ECDH.

Допустимые значения: `none` (DH отключён), `auto` (256-битный ECDH), имя кривой.

По умолчанию: `auto`

`client_tls_dheparams`

Тип обмена ключами DHE.

Допустимые значения: `none` (DH отключён), `auto` (2048-битный DH), `legacy` (1024-битный DH).

По умолчанию: `auto`

`server_tls_sslmode`

Режим TLS для подключений к серверам Postgres Pro. Режим по умолчанию — `prefer`.

`disable`

Обычный TCP. TLS даже не запрашивается у сервера.

`prefer`

Сначала всегда запрашивается TLS-подключение к Postgres Pro. В случае отказа происходит переключение на простой TCP. Сертификат сервера не проверяется. Это режим по умолчанию.

`require`

Подключение обязательно должно устанавливаться через TLS. Если сервер не принимает его, простой TCP использоваться не будет. Сертификат сервера не проверяется.

`verify-ca`

Подключение должно устанавливаться через TLS, а сертификат сервера должен быть действительным согласно файлу `server_tls_ca_file`. Имя узла сервера по сертификату не проверяется.

`verify-full`

Подключение должно устанавливаться через TLS, а сертификат сервера должен быть действительным согласно файлу `server_tls_ca_file`. Имя узла должно соответствовать указанному в сертификате.

`server_tls_ca_file`

Файл с корневым сертификатом, по которому будут проверяться сертификаты сервера Postgres Pro.

По умолчанию: не задано.

`server_tls_key_file`

Закрытый ключ pgbouncer, с которым он будет аутентифицироваться на сервере Postgres Pro.

По умолчанию: не задано.

`server_tls_cert_file`

Сертификат закрытого ключа. Сервер Postgres Pro может проверять его.

По умолчанию: не задано.

`server_tls_protocols`

Определяет, какие версии протокола TLS разрешены. Допустимые значения: `tlsv1.0`, `tlsv1.1`, `tlsv1.2`, `tlsv1.3`. Краткие обозначения: `all` (tlsv1.0,tlsv1.1,tlsv1.2,tlsv1.3), `secure` (tlsv1.2,tlsv1.3), `legacy` (равнозначно all).

По умолчанию: `secure`

`server_tls_ciphers`

Определяет допустимые шифры TLS, в синтаксисе OpenSSL. Допустимые краткие значения: `default`/`secure`, `compat`/`legacy`, `insecure`/`all`, `normal`, `fast`.

Распространяется только на соединения, использующие TLS версии 1.2 и ниже. В настоящее время нет параметра, который бы управлял выбором шифров при использовании TLS версии 1.3.

По умолчанию: `fast`

Опасные тайм-ауты

Установка следующих тайм-аутов может приводить к неожиданным ошибкам.

`query_timeout`

Запросы, выполняющиеся дольше этого времени (в секундах), будут отменяться. Его значение следует выбирать лишь немногим меньшим параметра `statement_timeout` на сервере, чтобы это происходило только при проблемах в сети.

По умолчанию: 0.0 (отключено)

`query_wait_timeout`

Максимальное время, которое могут ожидать выполнения запросы (в секундах). Если запрос не назначается серверу за это время, клиент отключается. Нулевое значение отключает этот параметр, то есть клиенты будут ожидать бесконечно.

Этот параметр используется для предотвращения захватывания подключений «зависшими» серверами. Кроме того, он полезен, когда сервер не работает или по какой-либо причине отклоняет подключения.

По умолчанию: 120.0

`cancel_wait_timeout`

Максимальное время, которое могут ожидать выполнения команды отмены запросов (в секундах). Если команды отмены запросов не назначаются серверу за это время, клиент отключается. Нулевое значение отключает этот параметр, то есть команды отмены запросов будут ожидать бесконечно.

Этот параметр используется для предотвращения блокировки клиента, когда команды отмены запросов не могут быть переданы из-за неработоспособности сервера.

По умолчанию: 10.0

`client_idle_timeout`

Клиентские подключения, простаивающие дольше этого времени (в секундах), закрываются. Это значение должно быть больше тайм-аута подключения, установленного на стороне клиента, и применяется оно только для решения проблем с сетью.

По умолчанию: 0.0 (отключено)

`idle_transaction_timeout`

Если клиент «простаивает в транзакции» дольше этого времени (в секундах), он будет отключён.

По умолчанию: 0.0 (отключено)

`suspend_timeout`

Сколько (в секундах) ждать сброса буфера при выполнении `SUSPEND` или перезагрузке (`-R`). Если сброс не завершился, подключение сбрасывается.

По умолчанию: 10

Низкоуровневые параметры сети

`pkt_buf`

Размер внутреннего буфера для пакетов. Влияет на размер отправляемых TCP-пакетов и общее использование памяти. Собственно пакеты libpq могут быть больше этого буфера, так что нет необходимости делать его большим.

По умолчанию: 4096

`max_packet_size`

Максимальный размер пакетов Postgres Pro, который сможет пропустить через себя pgbouncer. Один пакет представляет либо один запрос, либо строку из набора результатов. Размер всего набора результатов может быть больше.

По умолчанию: 2147483647

`listen_backlog`

Значение параметра очереди для `listen()`. Определяет, сколько неотвеченных запросов на подключение будет находиться в очереди. Когда очередь заполнена, следующие новые подключения будут сбрасываться.

По умолчанию: 128

`sbuf_loopcnt`

Устанавливает, сколько циклов должны обрабатываться данные для одного подключения, после чего нужно переходить к другим. Без этого ограничения одно подключение с большим набором результатом может занять pgbouncer на долгое время. В одном цикле обрабатываются данные размером `pkt_buf` байт. Ноль убирает ограничение.

По умолчанию: 5

`so_reuseport`

Определяет, будет ли для принимающих TCP-сокетов устанавливаться параметр `SO_REUSEPORT`. В некоторых операционных системах данный параметр позволяет запускать несколько экземпляров pgbouncer на одном узле, с одним и тем же принимающим портом, при этом соединения автоматически распределяются ядром. Таким образом pgbouncer может использовать больше ядер процессора. (pgbouncer — однопоточное приложение и использует одно ядро для каждого экземпляра.)

Данный параметр имеет желаемый эффект в Linux. В системах, которые вообще не поддерживают указанный параметр сокета, включение этого параметра приведёт к ошибке.

Для всех экземпляров pgbouncer на одном узле должны задаваться разные значения как минимум для `unix_socket_dir` и `pidfile`, а также `logfile`, если он используется. Также обратите внимание, что при использовании этого параметра вы больше не сможете подключаться к конкретному экземпляру pgbouncer через TCP/IP, что может затруднить мониторинг и сбор метрик.

Чтобы отмена запросов продолжала работать, следует настроить пиринг между различными процессами pgbouncer. За более подробной информацией обратитесь к описанию параметра конфигурации `peer_id` и раздела конфигурации `[peers]`. Также в разделе Примеры есть пример с использованием пиринга и `so_reuseport`.

По умолчанию: 0

`tcp_defer_accept`

Задаёт параметр сокета `TCP_DEFER_ACCEPT`; за подробным описанием обратитесь к `man 7 tcp`. (Это логический параметр: 1 означает, что он включён. Фактическое значение при включённом параметре в настоящее время неизменяемо и составляет 45 секунд.)

В настоящее время поддерживается только в Linux.

По умолчанию: 1 в Linux, в других системах — 0

`tcp_socket_buffer`

По умолчанию: не задано.

`tcp_keepalive`

Включает базовый опрос активности со стандартными параметрами ОС.

В Linux системные параметры по умолчанию: **tcp_keepidle=7200**, **tcp_keepintvl=75**, **tcp_keepcnt=9**. Вероятно, они имеют близкие значения и в других ОС.

По умолчанию: 1

`tcp_keepcnt`

По умолчанию: не задано.

`tcp_keepidle`

По умолчанию: не задано.

`tcp_keepintvl`

По умолчанию: не задано.

`tcp_user_timeout`

Задаёт параметр сокета `TCP_USER_TIMEOUT`. Он определяет максимальное количество времени в миллисекундах, в течение которого передаваемые данные могут оставаться неподтверждёнными, прежде чем соединение TCP будет принудительно закрыто. Если установлено значение 0, используются параметры операционной системы по умолчанию.

В настоящее время поддерживается только в Linux.

По умолчанию: 0

Раздел [databases]

Раздел `[databases]` определяет имена баз данных, к которым могут подключаться клиенты pgbouncer, и указывает, куда будут перенаправляться эти подключения. Раздел содержит строки ключ=значение, такие как

```
dbname = connection string
```

, где ключ будет приниматься как имя базы данных, а значение — как строка подключения, состоящая из параметров подключения в виде пар ключ=значение, описанных ниже (аналогично libpq, но сам libpq не используется и имеет другую функциональность).

Пример:

```
foodb = host=host1.example.com port=5432
bardb = host=localhost dbname=bazdb
```

Имя базы данных может содержать символы `_0-9A-Za-z` без кавычек. Имена, содержащие другие символы, должны заключаться в двойные кавычки по правилам для идентификаторов SQL (две кавычки ("") воспринимаются внутри строки как одна).

Имя базы данных `pgbouncer` зарезервировано для административной консоли и не может использоваться здесь в качестве ключа.

«*» воспринимается как имя всех остальных баз: если точного соответствия имени для запрошенной базы данных не находится, в качестве строки подключения выбирается данное значение. Например, если есть следующая запись (и нет других переопределяющих записей):

```
* = host=foo
```

В этом случае подключение к pgbouncer с указанием базы данных `bar` будет работать так, как если бы существовала следующая запись (в качестве `dbname` по умолчанию используется имя базы, полученное от клиента):

```
bar = host=foo dbname=bar
```

Такие автоматически создаваемые записи баз данных очищаются, если они простаивают больше времени, задаваемого параметром `autodb_idle_timeout`.

`dbname`

Имя целевой базы данных.

По умолчанию: имя базы данных, полученное от клиента

`host`

Имя или IP-адрес узла, к которому нужно подключиться. Имена узлов разрешаются в момент подключения, и результат кешируется в течение времени, заданного параметром `dns_max_ttl`. Если результат разрешения имени меняется, существующие подключения к серверу автоматически закрываются при их освобождении (в соответствии с режимом пула), и изменение немедленно отражается на новых подключениях. Если DNS возвращает несколько записей, они используются по очереди.

Если значение начинается с `/`, используется сокет Unix в пространстве имён файловой системы. Если значение начинается с `@`, используется сокет Unix в абстрактном пространстве имён.

Можно указать список имён узлов или адресов через запятую. В этом случае подключения выполняются по кругу. (Если список узлов содержит имена узлов, которые, в свою очередь, разрешаются через DNS в несколько адресов, системы балансировщика работают независимо. Это зависит от реализации, которая может быть изменена.) Обратите внимание, что в списке все узлы должны быть доступны в любой момент: нет никаких механизмов для пропуска недоступных или выбора только доступных узлов из списка и т. п. (Это является существенным отличием от списка узлов в libpq.) Также учтите, что это повлияет только на выбор цели для новых подключений. Назначение клиентов для уже установленных подключений к серверу подробнее рассматривается в описании параметра `server_round_robin`.

Примеры:

```
host=localhost
host=127.0.0.1
host=2001:0db8:85a3:0000:0000:8a2e:0370:7334
host=/var/run/postgresql
host=192.168.0.1,192.168.0.2,192.168.0.3
```

По умолчанию: не задан, что подразумевает использование сокетов Unix `port`

По умолчанию: 5432

`user`

Если задано `user=`, все подключения к целевой базе данных будут выполняться с заданным именем пользователя, что означает, что для этой базы данных будет всего один пул.

В противном случае pgbouncer подключается к целевой базе данных с именем пользователя, переданным клиентом, что означает, что для каждого пользователя будет отдельный пул.

`password`

Если пароль здесь не задаётся, будет использован пароль из `auth_file` или `auth_query`.

`auth_user`

Переопределяет глобальную переменную `auth_user`, если она задана.

`pool_size`

Задаёт максимальный размер пулов для этой базы данных. Если не задано, применяется значение `default_pool_size`.

`min_pool_size`

Задаёт минимальный размер пулов для этой базы данных. Если не задано, применяется глобальное значение `min_pool_size`.

`reserve_pool`

Задаёт число дополнительных подключений для этой базы данных. Если не задано, применяется значение `reserve_pool_size`.

`connect_query`

Запрос, который будет выполняться сразу после установления соединения, но до того, как его смогут использовать какие-либо клиенты. Если при запросе возникают ошибки, они только фиксируются в журнале, другой реакции не следует.

`pool_mode`

Задаёт режим пула для данной базы данных. Если этот параметр не задаётся, применяется значение `pool_mode` по умолчанию.

`max_db_connections`

Задаёт максимум подключений для базы данных (то есть, используя все пулы этой базы данных, нельзя будет установить больше этого числа подключений к серверу).

`client_encoding`

Запрашивает у сервера использование указанной клиентской кодировки (`client_encoding`).

`datestyle`

Запрашивает у сервера использование указанного стиля даты (`datestyle`).

`timezone`

Запрашивает у сервера использование указанного часового пояса (`timezone`).

Раздел [users]

Этот раздел содержит строки ключ=значение, такие как

```
user1 = settings
```

, где в качестве ключа принимается имя пользователя, а в качестве значения — переопределяемые для него параметры конфигурации.

Пример:

```
user1 = pool_mode=session
```

Здесь доступно лишь несколько параметров.

`pool_mode`

Задаёт режим пула для всех подключений данного пользователя. Если этот параметр не задаётся, применяется значение `pool_mode` по умолчанию или заданное для базы данных.

`max_user_connections`

Задаёт максимум подключений для пользователя (то есть, используя все пулы, нельзя будет установить больше этого числа подключений к серверу).

Раздел [peers]

В этом разделе определяются одноранговые узлы, которым pgbouncer может пересылать команды отмены запросов, и куда будут перенаправляться эти команды.

Процессы pgbouncer можно объединить в группу, определив значение `peer_id` и раздел `[peers]` в конфигурациях всех процессов pgbouncer. Эти процессы могут затем пересылать команды отмены запросов своему родительскому процессу. Это необходимо для работы команд отмены запросов, когда несколько процессов pgbouncer (возможно, на разных серверах) работают с одним и тем же балансировщиком нагрузки TCP. Команды отмены запросов отправляются по разным TCP-подключениям, отличным от подключения запроса, который они отменяют, поэтому балансировщик нагрузки TCP может отправить подключение команды отмены запроса другому процессу, отличному от того, для которого оно предназначалось. Благодаря пирингу эти команды отмены запросов в конечном итоге попадают в правильный процесс.

Раздел содержит строки ключ=значение, такие как

```
peer_id = connection string
```

, где ключ будет приниматься как `peer_id`, а значение — как строка подключения, состоящая из параметров подключения в виде пар ключ=значение, описанных ниже (аналогично libpq, но сам libpq не используется и имеет другую функциональность).

Пример:

```
1 = host=host1.example.com
2 = host=/tmp/pgbouncer-2  port=5555
```

> **Примечание**
>
> Чтобы пиринг работал, `peer_id` каждого процесса pgbouncer в группе должен быть уникальным в пределах группы одноранговых узлов, а раздел `[peers]` должен содержать записи для каждого из этих идентификаторов. За примерами обратитесь к соответствующей главе. Раздел `[peers]` может (но не обязательно должен) содержать `peer_id` узла pgbouncer, для которого предназначена конфигурация. Такая запись не учитывается, но упрощает управление конфигурациями, так как позволяет использовать один и тот же раздел `[peers]` для нескольких конфигураций.

`host`

Имя или IP-адрес узла, к которому нужно подключиться. Имена узлов разрешаются в момент подключения, и результат кешируется в течение времени, заданного параметром `dns_max_ttl`.

Если DNS возвращает несколько результатов, они используются в циклическом порядке. Но в целом не рекомендуется использовать имя узла, которое разрешается для нескольких IP-адресов, поскольку тогда команда отмены запроса всё равно может быть перенаправлена не на тот узел, и её придётся пересылать снова (что разрешено не более трёх раз).

Если значение начинается с `/`, используется сокет Unix в пространстве имён файловой системы. Если значение начинается с `@`, используется сокет Unix в абстрактном пространстве имён.

Примеры:

```
host=localhost
host=127.0.0.1
host=2001:0db8:85a3:0000:0000:8a2e:0370:7334
host=/var/run/pgbouncer-1
```

`port`

По умолчанию: 6432

`pool_size`

Если не задан, используется `default_pool_size`.

Директива включения

Файл конфигурации pgbouncer может содержать директивы включения, которые указывают, что нужно прочитать и обработать дополнительный файл конфигурации. Это позволяет разделить файл конфигурации на физически отдельные части. Директивы включения выглядят примерно так:

```
%include имя_файла
```

Если файл задаётся не абсолютным путём, его путь воспринимается относительно текущего рабочего каталога.

Формат файла аутентификации

В этом разделе описывается формат файла, задаваемого параметром `auth_file`. Это текстовый файл следующего вида:

```
"username1" "password" ...
"username2" "md5abcdef012342345" ...
"username2" "SCRAM-SHA-256$число_итераций:соль$сохранённый_ключ:ключ_сервера"
```

В строке должно быть минимум два поля, заключённых в двойные кавычки. В первом поле задаётся имя пользователя, а во втором — пароль, либо открытым текстом, либо защищённый MD5 или SCRAM. Остальное содержимое строки pgbouncer игнорирует. Кавычки в этой строке можно записать, продублировав их.

Принятый в Postgres Pro формат пароля, защищённого MD5:

```
"md5" + md5(password + username)
```

Таким образом, для пользователя `admin` с паролем `1234` защищённый MD5 пароль будет следующим: `md545f2603610af569b6155c45067268c6b`.

Принятый в Postgres Pro формат пароля, защищённого SCRAM:

```
SCRAM-SHA-256$число_итераций:соль$сохранённый_ключ:ключ_сервера
```

Пароли или шифры, записанные в файле аутентификации, имеют два предназначения. Во-первых, если для клиента настроен метод аутентификации по паролю, то пароль пользователя сверяется с записанным в этом файле. Во-вторых, если сервер БД требует аутентификации по паролю (и пароль не задан явно в строке подключения), пароль для исходящего подключения к этому серверу также будет взят из этого файла. Второй вариант использования возможен при условии, что пароль записан в открытом виде или в виде MD5-хеша. Пароли в формате SCRAM могут использоваться для входа на сервер, только если клиент проходит аутентификацию по протоколу SCRAM, в определении базы pgbouncer не задаётся имя пользователя, а шифры SCRAM на сервере Postgres Pro и в pgbouncer идентичны (совпадает не только пароль, но и соль, а также количество итераций). Эти условия объясняются присущим механизму SCRAM свойством — из шифра SCRAM нельзя извлечь данные, требующиеся для аутентификации.

Файл аутентификации можно составить вручную, но также можно сгенерировать его из какого-то другого списка пользователей и паролей. В качестве примера скрипта, который генерирует файл с данными аутентификации из таблицы `pg_shadow`, может быть полезен `./etc/mkauth.py`.

Вы также можете использовать `auth_query` вместо `auth_file` и таким образом обойтись без отдельного файла аутентификации.

Формат файла HBA

Расположение файла HBA определяется параметром `auth_hba_file`. Он используется, только если для `auth_type` задано значение `hba`.

Соответствует формату Postgres Pro `pg_hba.conf`, описанному в Разделе 19.1.

- Поддерживаемые типы записей: `local`, `host`, `hostssl`, `hostnossl`.
- В поле базы данных поддерживаются варианты: `all`, `sameuser`, `@`*`файл`*, несколько имён. Не поддерживаются: `replication`, `samerole`, `samegroup`.
- В поле имени пользователя поддерживаются варианты: `all`, `@`*`файл`*, несколько имён. Не поддерживается: `+groupname`.
- В поле адреса поддерживается: `IPv4`, `IPv6`. Не поддерживаются: имена DNS, префиксы доменов.
- В поле метода аутентификации поддерживаются только те, которые поддерживает pgbouncer в `auth_type`, а также `peer` и `reject`, но исключая `any` и `pam`, которые работают только глобально. Сопоставления имён пользователей (`map=`) не поддерживаются.

Примеры

Простой пример конфигурации:

```
[databases]
template1 = host=localhost dbname=template1 auth_user=someuser

[pgbouncer]
pool_mode = session
listen_port = 6432
listen_addr = localhost
auth_type = md5
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = someuser
stats_users = stat_collector
```

Примеры базы данных:

```
[databases]

; подключение к foodb через сокет Unix
foodb =

; перенаправление bardb в базу bazdb на локальном узле
bardb = host=localhost dbname=bazdb

; обращение к целевой базе данных будет производить один пользователь
```

```
forcedb = host=localhost port=300 user=baz password=foo client_encoding=UNICODE
 datestyle=ISO
```

Пример безопасной функции для `auth_query`:

```
CREATE OR REPLACE FUNCTION pgbouncer.user_lookup(in i_username text, out uname text,
 out phash text)
RETURNS record AS $$
BEGIN
    SELECT usename, passwd FROM pg_catalog.pg_shadow
    WHERE usename = i_username INTO uname, phash;
    RETURN;
END;$$ LANGUAGE plpgsql SECURITY DEFINER;
REVOKE ALL ON FUNCTION pgbouncer.user_lookup(text) FROM public, pgbouncer;
GRANT EXECUTE ON FUNCTION pgbouncer.user_lookup(text) TO pgbouncer;
```

Примеры конфигураций для двух одноранговых процессов pgbouncer при создании многоядерной структуры pgbouncer с использованием `so_reuseport`.

Конфигурация для первого процесса:

```
[databases]
postgres = host=localhost dbname=postgres

[peers]
1 = host=/tmp/pgbouncer1
2 = host=/tmp/pgbouncer2

[pgbouncer]
listen_addr=127.0.0.1
auth_file=auth_file.conf
so_reuseport=1
unix_socket_dir=/tmp/pgbouncer1
peer_id=1
```

Конфигурация для второго процесса:

```
[databases]
postgres = host=localhost dbname=postgres

[peers]
1 = host=/tmp/pgbouncer1
2 = host=/tmp/pgbouncer2

[pgbouncer]
listen_addr=127.0.0.1
auth_file=auth_file.conf
so_reuseport=1
; only unix_socket_dir and peer_id are different
unix_socket_dir=/tmp/pgbouncer2
peer_id=2
```

pg_filedump

pg_filedump — отображает отформатированное содержимое файлов кучи, индексов или управляющего файла Postgres Pro

Синтаксис

`pg_filedump` [*параметр*...] [*файл*]

Описание

pg_filedump — утилита для представления управляющего файла и файлов кучи/индекса Postgres Pro в удобочитаемом виде. Вы можете форматировать/выводить содержимое файлов несколькими способами, описанными в разделе Параметры, а также выводить данные непосредственно в двоичном виде. Тип файла (куча/индекс) обычно определяется автоматически по содержимому блоков внутри файла. Однако чтобы прочитать содержимое файла `pg_control`, необходимо использовать параметр `-c`. По умолчанию весь файл обрабатывается с размером блока, указанным в блоке 0, и отображаются относительные адреса блоков. Значения по умолчанию можно изменить с помощью параметров времени выполнения. Некоторые параметры могут показаться странными, но они необходимы. Например, размер блока может потребоваться задать, если заголовок блока 0 был повреждён.

Установка

pg_filedump поставляется вместе с Postgres Pro Standard в виде отдельного пакета `pg-filedump-std-17` (подробные инструкции по установке приведены в Главе 16).

Параметры

По умолчанию выводится содержимое всего файла, с относительными адресами и размером блока, полученным из блока 0.

Для файлов кучи и файлов индекса действуют следующие параметры:

`-a`

Отображать абсолютные адреса. Содержимое заголовка блока всегда адресуется относительно блока.

`-b`

Выводить двоичные образы блоков в заданном диапазоне. При этом все остальные параметры форматирования не действуют.

`-d`

Показать отформатированный вывод байтов блока. При этом все остальные параметры форматирования не действуют.

`-D` *`список_атрибутов`*

Декодировать кортежи, используя заданный список типов, разделённых запятыми. Список поддерживаемых типов:

`bigint`
`bigserial`
`bool`
`char`
`charN` — char(n)
`date`
`float`
`float4`

`float8`
`int`
`json`
`macaddr`
`name`
`numeric`
`oid`
`real`
`serial`
`smallint`
`smallserial`
`text`
`time`
`timestamp`
`timetz`
`uuid`
`varchar`
`varcharN` — varchar(n)
`xid`
`xml`
`~` — игнорировать все последующие атрибуты в кортеже.

`-f`

Показать отформатированный вывод байтов блока вместе с интерпретацией.

`-h`

Показать справку.

`-i`

Выводить подробную информацию об интерпретируемых элементах.

`-k`

Проверять контрольные суммы блоков.

`-o`

Не выводить старые значения.

`-R` *`начальный_блок`* `[`*`конечный_блок`*`]`

Показать только заданные диапазоны блоков в файле. Блоки нумеруются, начиная с 0. Параметры *`начальный_блок`* и *`конечный_блок`* определяют первый и последний блок диапазона, соответственно. Если *`конечный_блок`* опускается, а задаётся только *`начальный_блок`*, будет обработан только один этот блок.

`-s` *`размер_сегмента`*

Задать размер сегмента равным значению *`размер_сегмента`*.

`-t`

Обрабатывать файлы TOAST.

`-v`

Вывести дополнительную информацию об отношениях TOAST.

`-n` *`номер_сегмента`*

Задать номер сегмента равным значению *`номер_сегмента`*.

`-S` *`размер_блока`*

Задать размер блока равным значению *`размер_блока`*.

`-x`

Форматировать элементы блока как элементы индекса.

`-y`

Форматировать элементы блока как элементы кучи.

Для управляющих файлов действуют следующие параметры:

`-c`

Интерпретировать указанный файл как управляющий файл.

`-f`

Показать отформатированный вывод байтов содержимого вместе с интерпретацией.

`-S` *`размер_блока`*

Задать размер блока равным значению *`размер_блока`*.

Дополнительные параметры:

`-m`

Интерпретировать файл как файл `pg_filenode.map` и вывести его содержимое. Все остальные параметры будут проигнорированы.

В большинстве случаев рекомендуется использовать параметры `-i` и `-f`, чтобы получить наиболее информативный вывод.

Автор

Патрик Макдональд `<patrickm@redhat.com>`

Приложение J. Внешние проекты

Postgres Pro — сложный программный проект, управлять которым довольно непросто. Поэтому мы пришли к заключению, что многие дополнения и усовершенствования Postgres Pro будет эффективнее разрабатывать отдельно от основного проекта.

J.1. Клиентские интерфейсы

В базовый дистрибутив Postgres Pro Standard включены только два клиентских интерфейса:

- libpq включён, потому что это основной интерфейс языка C и многие другие клиентские интерфейсы построены на основе него.
- ECPG включён, потому что он зависит от грамматики языка SQL на стороне сервера, и таким образом, очень чувствителен к изменениям в самом Postgres Pro.

Все остальные языковые интерфейсы разрабатываются в отдельных проектах и распространяются отдельно. Актуальный *список языковых интерфейсов* находится на странице проекта PostgreSQL wiki. Заметьте, что какие-то проекты могут выпускаться под лицензией, отличной от лицензии Postgres Pro. За дополнительной информацией о каждом языковом интерфейсе, включая условия лицензии, обратитесь к его сайту и документации.

https://wiki.postgresql.org/wiki/List_of_drivers

J.2. Средства администрирования

Для Postgres Pro доступны различные инструменты администрирования. Одним из них является *Postgres Pro Enterprise Manager* — интегрированная панель управления СУБД Postgres Pro, среди преимуществ которой единый интерфейс управления и мониторинга, централизованный доступ с иерархией прав, а также возможность сочетать работу в интерфейсе и командной строке из окна браузера. Кроме того, доступен ряд других решений, например платформа с открытым исходным кодом *pgAdmin*.

J.3. Процедурные языки

Базовый дистрибутив Postgres Pro включает несколько процедурных языков: PL/pgSQL, PL/Tcl, PL/Perl и PL/Python.

Кроме того, вне основного дистрибутива Postgres Pro разрабатываются и поддерживаются несколько процедурных языков. Актуальный список *процедурных языков* находится на странице проекта PostgreSQL wiki. Заметьте, что какие-то проекты могут выпускаться под лицензией, отличной от лицензии Postgres Pro. За дополнительной информацией о каждом процедурном языке, включая условия лицензии, обратитесь к его сайту и документации.

https://wiki.postgresql.org/wiki/PL_Matrix

J.4. Расширения

Postgres Pro проектируется так, чтобы его можно было легко расширять. Поэтому расширения, загружаемые в базу данных, могут функционировать в точности так же, как и встроенные функции. Расширения, входящие в дистрибутив Postgres Pro, описаны в Приложение F. Другие расширения разрабатываются независимо, как, например, *PostGIS*. Postgres Pro позволяет разрабатывать независимо даже решения по репликации. Например, *Slony-I* — популярное средство репликации по схеме главный/резервный, разрабатываемое отдельно от основного проекта.

Приложение K. Настройка Postgres Pro для решений 1С

Этот раздел описывает, как настроить Postgres Pro для работы с решениями 1С, чтобы получить максимальную производительность и стабильность.

K.1. Настройка локали

Перед настройкой Postgres Pro для решений 1С убедитесь, что в вашей системе установлена русская локаль и она является активной локалью пользователя, создающего кластер баз данных. Следующий пример показывает, как установить локаль `ru_RU.UTF-8` в операционных системах Debian:

```
sudo dpkg-reconfigure locales
export LANG="ru_RU.UTF-8"
/opt/pgpro/std-17/bin/pg-setup initdb
```

За подробностями обратитесь к разделам документации вашей операционной системы и 1С.

K.2. Основные требования и конфигурация сервера

Чтобы обеспечить высокую производительность для решений 1С, сервер Postgres Pro должен иметь как минимум следующие аппаратные характеристики:

- 8 - 512 ГБ ОЗУ
- 4 - 256 ядер ЦП
- RAID 0-1 или SSD

Точные рекомендуемые характеристики зависят от особенностей вашей системы и нагрузки на неё.

Рекомендуется настроить сервер Postgres Pro следующим образом:

- Отключить технологию гиперпоточности.
- Отключить режим энергосбережения. В противном случае могут непредсказуемо вырастать задержки отклика базы данных.
- Отключить подкачку для разделяемой памяти POSIX/System V.

K.3. Настройка параметров сервера

В этом документе описывается, как настроить параметры конфигурации сервера Postgres Pro для максимальной производительности и стабильности при работе с решениями 1С. Эти параметры редактируются в файле конфигурации `postgresql.conf`. Учтите, что приведённые значения параметров являются приблизительными. Они могут рассматриваться как начальная точка для дальнейшей тонкой настройки.

Для быстрой автоматической настройки Postgres Pro можно также использовать утилиту pgpro_tune. Она предоставляет пресет `1c.tune`, который задаёт оптимальные значения параметров конфигурации, специфичные для 1С. За подробностями об использовании этой утилиты обратитесь к её документации.

K.3.1. Параметры подключения

Увеличьте максимальное количество возможных одновременных подключений к серверу баз данных, используя параметр конфигурации max_connections. Решения 1С могут открывать большое количество соединений, даже если не все из них используются. Поэтому рекомендуется разрешить не менее 500 подключений. Чем больше количество клиентских сеансов, тем больше должно быть значение параметра.

```
-- max_connections = 500..10000
```

K.3.2. Параметры потребления памяти

1. Увеличьте объём памяти, который будет использоваться базой данных для кеширования данных. Для этого используйте параметр конфигурации shared_buffers. Для хорошей производительности рекомендуется устанавливать для этого параметра как минимум 25% от доступного объёма оперативной памяти.

   ```
   -- shared_buffers = ОЗУ/4
   ```

2. Увеличьте максимальный объём памяти, который будет использоваться для временных буферов в каждом сеансе базы данных, чтобы обеспечить корректную работу временных таблиц. Для этого используйте параметр конфигурации temp_buffers.

   Рекомендуемое значение должно быть между `32MB` и `256MB`. Точное значение зависит от объёма памяти на сервере и количества одновременных подключений. Чем больше подключений и меньше объём памяти, тем меньше должно быть значение параметра.

   ```
   -- temp_buffers = 32..256MB
   ```

3. Увеличьте максимальный объём памяти, который будет использоваться одной операцией запроса перед записью на временные файлы диска. Для этого используйте параметр конфигурации work_mem.

   При выборе значения параметра учтите, что сложные запросы могут одновременно выполнять несколько операций сортировки и хеширования, и при этом каждая операция может использовать указанный объём памяти. Кроме того, такие операции могут выполняться одновременно в нескольких сеансах. Таким образом, общий объём используемой памяти может быть в несколько раз больше указанного значения. Тонкую настройку параметра можно выполнить, отслеживая количество создаваемых в системе временных файлов.

   ```
   -- work_mem = ОЗУ/32..64 или 32MB..256MB
   ```

4. Увеличьте максимальный объём памяти, который будет использоваться одной операцией обслуживания базы данных, например `VACUUM` или `CREATE INDEX`. Для этого используйте параметр конфигурации maintenance_work_mem.

   Учтите, что в сеансе базы данных одновременно может выполняться только одна такая операция. Таким образом, можно задавать это значение гораздо больше, чем `work_mem`. Более высокие значения могут улучшать производительность операций очистки и восстановления базы данных из копии.

   ```
   -- maintenance_work_mem = ОЗУ/16..32 или work_mem * 4 или 256MB..4GB
   ```

5. В случае существенной фрагментации памяти укажите следующую переменную окружения в файле `/etc/systemd/system/postgresql.service`:

   ```
   -- Environment = MALLOC_MMAP_THRESHOLD_= 8192
   ```

K.3.3. Параметры использования ресурсов ядра

Увеличьте максимальное число файлов, которые могут быть одновременно открыты каждым серверным подпроцессом. Для этого используйте параметр конфигурации max_files_per_process.

Рекомендуемое значение — `10000`, но оно может быть увеличено в зависимости от нагрузки. Максимальное допустимое значение зависит от операционной системы. Если сервер достигает указанный предел, он начинает открывать и закрывать файлы, что влияет на производительность. Отслеживать открытые файлы можно с помощью команды `lsof`.

```
-- max_files_per_process = 10000
```

K.3.4. Параметры фоновой записи

1. Уменьшите задержку между раундами активности процесса фоновой записи, используя параметр конфигурации bgwriter_delay. Учтите, что слишком большое значение параметра будет увеличивать нагрузку на процесс выполнения контрольных точек и обслуживающие процессы, а слишком маленькое значение будет приводить к полной загрузке одного из ядер.

   ```
   -- bgwriter_delay = 20ms
   ```

2. Увеличьте значения для параметров конфигурации bgwriter_lru_multiplier и bgwriter_lru_maxpages, которые управляют количеством «грязных» буферов, записанных процессом фоновой записи в каждом раунде активности.

   ```
   -- bgwriter_lru_multiplier = 4.0
   -- bgwriter_lru_maxpages = 400
   ```

K.3.5. Параметры асинхронного поведения

1. Укажите количество одновременных операций ввода-вывода на диск, которое каждый отдельный сеанс Postgres Pro пытается выполнять параллельно. Для этого используйте параметр конфигурации effective_io_concurrency. В настоящее время этот параметр влияет только на сканирование по битовой карте.

   Для магнитных носителей хорошим начальным значением этого параметра будет число отдельных дисков, составляющих массив RAID 0 или RAID 1, который используется для базы данных. Однако слишком большое значение параметра будет полностью загружать дисковый массив, что приведёт к дополнительной нагрузке на процессор.

   Диски SSD часто могут обрабатывать множество параллельных запросов, так что лучшим значением может быть несколько сотен.

   ```
   -- effective_io_concurrency = 2
   ```

2. Установите для параметра конфигурации max_parallel_workers_per_gather значение 0. Это выключает параллельные запросы, которые могут быть запущены узлами `Gather` и `Gather Merge`. Параллельные запросы могут потреблять значительно больше ресурсов, чем не параллельные.

   ```
   -- max_parallel_workers_per_gather = 0
   ```

K.3.6. Параметры журнала предзаписи (WAL)

1. Оставьте для параметра конфигурации fsync значение `on` (по умолчанию). В этом случае сервер Postgres Pro пытается добиться, чтобы изменения были записаны на диск физически, выполняя системные вызовы `fsync()`. Это гарантирует, что кластер баз данных сможет вернуться в согласованное состояние после сбоя операционной системы или оборудования.

   Хотя отключение параметра `fsync` часто даёт повышение производительности, это может привести к неисправимой порче данных в случае отключения питания или сбоя системы.

   ```
   -- fsync = on
   ```

   Важное замечание: если ваш диск RAID имеет кеш и работает в режиме отложенной записи, убедитесь, что у кеша контроллера диска есть рабочая батарея. В противном случае данные, записанные в кеш, могут быть потеряны при отключении питания, и сервер Postgres Pro не сможет их восстановить.

2. Установите для параметра конфигурации synchronous_commit значение `off`. Это означает, что сервер Postgres Pro не дожидается сброса записей WAL на диск, но возвращает клиенту подтверждение об успешном выполнении.

   В отличие от fsync, указание значения `off` для этого параметра не угрожает целостности данных. Сбой операционной системы или базы данных может привести только к потере последних транзакций. Таким образом, выключение параметра `synchronous_commit` полезно, когда очень важна высокая производительность.

```
-- synchronous_commit = off
```

3. Увеличьте значения для параметров конфигурации checkpoint_timeout, min_wal_size и max_wal_size для сокращения частоты контрольных точек. Слишком частые контрольные точки приводят к значительной нагрузке на дисковую систему.

```
-- checkpoint_timeout = 15min
-- min_wal_size = 512MB..4G
-- max_wal_size = 2 * min_wal_size
```

4. Укажите параметр конфигурации commit_delay, если среднее количество транзакций в секунду (TPS) больше 1000. Этот параметр добавляет паузу перед выполнением сброса WAL и может увеличить быстродействие при фиксировании группы транзакций.

```
-- commit_delay = 1000
```

K.3.7. Параметры планирования запросов

1. Укажите параметр конфигурации effective_cache_size, чтобы определить предположение планировщика об эффективном размере дискового кеша, доступном для одного запроса.

   Производительность оптимизатора запросов зависит от выделенного объёма оперативной памяти. При высоких значениях, как правило, используется сканирование по индексу, при низких — последовательное сканирование.

   ```
   -- effective_cache_size = ОЗУ - shared_buffers
   ```

2. Укажите параметр конфигурации random_page_cost, чтобы определить приблизительную стоимость чтения одной произвольной страницы с диска.

   Значение этого параметра задаётся относительно параметра seq_page_cost, для которого по умолчанию установлено значение `1.0`. При уменьшении значения `random_page_cost` система начинает выбирать сканирование по индексу вместо последовательного сканирования. Рекомендуемое значение зависит от времени поиска дисковой системы. Чем ниже время поиска, тем ниже должно быть значение параметра (но не меньше `1.0`).

   ```
   -- random_page_cost = 1.5-2.0 (для RAID) или 1.1-1.3 (для SSD)
   ```

3. Увеличьте значение для параметра конфигурации from_collapse_limit. Планировщик будет объединять вложенные запросы с внешними запросами, если результирующий список `FROM` будет содержать не больше указанного количества элементов. Учтите, что при меньших значениях параметра сокращается время планирования, но планы запросов могут стать менее эффективными.

   ```
   -- from_collapse_limit = 20
   ```

4. Увеличьте значение параметра конфигурации join_collapse_limit. Планировщик будет переписывать явные конструкции `JOIN` (за исключением `FULL JOIN`) на списки элементов `FROM` каждый раз, когда список содержит не больше указанного количества элементов. Учтите, что при меньших значениях параметра сокращается время планирования, но планы запросов могут стать менее эффективными.

   ```
   -- join_collapse_limit = 20
   ```

5. Оставьте для параметра конфигурации geqo значение `on` (по умолчанию). Это включает генетический оптимизатор запросов (Genetic Query Optimizer, GEQO), который осуществляет планирование запросов, применяя эвристический поиск. GEQO сокращает время планирования для сложных запросов, объединяющих множество отношений, но иногда создаёт планы, которые менее эффективные, чем планы, выбираемые при обычном алгоритме полного перебора.

Для управления использованием GEQO укажите параметр конфигурации geqo_threshold. GEQO будет использоваться для планирования запросов, которые содержат не менее указанного количества элементов `FROM`.

```
-- geqo = on
-- geqo_threshold = 12
```

6. Укажите для параметра конфигурации jit значение `off` для выключения компиляции Just-in-Time (JIT). Эта компиляция имеет преимущество в первую очередь для длительных запросов, нагружающих процессор, например для аналитических запросов. Для быстрых запросов добавляемые накладные расходы будут выше, чем время, которое может сохранить компиляция JIT.

   ```
   -- jit = off
   ```

K.3.8. Параметры автоматической очистки

1. Оставьте для параметра конфигурации autovacuum значение `on` (по умолчанию) для запуска демона автоочистки. Обратите внимание, что выключение демона автоочистки будет приводить к увеличению размера базы данных и значительному ухудшению производительности.

   ```
   -- autovacuum = on
   ```

2. Увеличьте максимальное количество процессов автоочистки, которые могут выполняться одновременно, используя параметр конфигурации autovacuum_max_workers. Чем больше запросов на запись выполняется в системе, тем больше требуется процессов автоочистки.

   ```
   -- autovacuum_max_workers = ядра ЦП/4..2 (но не менее 4)
   ```

3. Сократите минимальную задержку между раундами автоочистки, используя параметр конфигурации autovacuum_naptime. Если значение параметра слишком большое, может не хватить времени для очистки всех требуемых таблиц. Это приведёт к увеличению размера базы данных и значительному снижению производительности. Однако слишком маленькое значение приведёт к бесполезной нагрузке.

   ```
   -- autovacuum_naptime = 20s
   ```

K.3.9. Параметры клиентских подключений

1. Укажите каталог, в котором будут создаваться временные таблицы и индексы временных таблиц, используя параметр конфигурации temp_tablespaces. Обычно решения 1С используют множество временных таблиц. Для увеличения производительности при работе с такими таблицами расположите этот каталог на отдельных быстрых дисках.

   Перед этим сначала необходимо создать табличное пространство, используя команду `CREATE TABLESPACE`. Если характеристики целевых дисков отличаются от основных дисков, укажите в этой команде соответствующее значение для параметра `random_page_cost`.

   ```
   -- temp_tablespaces = 'имя_табличного_пространства'
   ```

2. Укажите для параметра конфигурации row_security значение `off`, чтобы выдавать ошибку, если результаты запроса фильтруются политикой безопасности на уровне строк.

   ```
   -- row_security = off
   ```

K.3.10. Параметры управления блокировками

Увеличьте значение для параметра конфигурации max_locks_per_transaction до `256`. Оно задаёт количество объектов для каждого серверного процесса или подготовленной транзакции, которые могут быть одновременно заблокированы.

Обычно решения 1С используют множество временных таблиц. Каждый обслуживающий процесс обычно содержит много временных таблиц. При закрытии соединения Postgres Pro пытается удалить все временные таблицы в одной транзакции, при этом транзакция может использовать множество блокировок. Если число блокировок превысит значение `max_locks_per_transaction`, транзакция прервётся, оставляя множество потерянных временных таблиц.

```
-- max_locks_per_transaction = 256
```

K.3.11. Параметры для совместимости с 1С

1. Установите для параметра конфигурации standard_conforming_strings значение `off`, чтобы включить экранирование символом обратной косой черты (\) для всех строк.

   ```
   -- standard_conforming_strings = off
   ```

2. Установите для параметра конфигурации escape_string_warning значение `off`, чтобы выключить предупреждение об использовании символа обратной косой черты.

   ```
   -- escape_string_warning = off
   ```

K.3.12. Путь к подкаталогу `pg_stat_tmp`

Измените путь по умолчанию к подкаталогу `PGDATA/pg_stat_tmp`, чтобы разместить его отдельно от каталога кластера. Этот подкаталог содержит временные файлы для подсистемы статистики. Эти файлы интенсивно изменяются, что создаёт значительную нагрузку на дисковую систему. Рекомендуется размещать этот подкаталог в `tmpfs` (хранилище временных файлов).

K.4. Дополнительные модули

K.4.1. Встроенные модули по умолчанию

Чтобы обеспечить поддержку решений 1С, дистрибутив Postgres Pro содержит следующие встроенные модули, которые включены по умолчанию и не требуют дополнительной настройки:

- fasttrun — предоставляет транзакционно-небезопасную функцию для усечения временных таблиц, предотвращающую увеличение размера `pg_class`.
- fulleq — предоставляет дополнительный оператор равенства для совместимости с Microsoft SQL Server.
- mchar — предоставляет дополнительные типы данных `MCHAR` и `MVARCHAR` для совместимости с Microsoft SQL Server.

K.4.2. dbcopies_decoding

dbcopies_decoding — модуль 1С для обновления копий баз данных, который включён по умолчанию. Для использования механизма логической репликации этого модуля установите для параметра `wal_level` значение `logical`.

За более подробной информацией об этом модуле обратитесь в поддержку 1С.

K.4.3. plantuner

Модуль plantuner позволяет оптимизировать планы выполнения запросов. Чтобы настроить его для решений 1С, выполните следующее:

1. Добавьте имя библиотеки в переменную `shared_preload_libraries` в файле `postgresql.conf`.

   ```
   shared_preload_libraries = 'plantuner'
   ```

2. Установите для параметра `plantuner.fix_empty_table` значение `on` для улучшения планирования запросов с недавно созданными пустыми таблицами.

   ```
   plantuner.fix_empty_table = on
   ```

K.4.4. online_analyze

Модуль online_analyze предоставляет набор функций, которые немедленно обновляют статистику для задействованных таблиц после операций `INSERT`, `UPDATE`, `DELETE`, `SELECT INTO` или `COPY`.

Чтобы настроить этот модуль, выполните шаги ниже.

1. Добавьте имя библиотеки в переменную `shared_preload_libraries` в файле `postgresql.conf`.

   ```
   shared_preload_libraries = 'online_analyze'
   ```
2. Включите этот модуль.

   ```
   online_analyze.enable = on
   ```
3. Укажите минимальное число изменений строк, после которого может начаться немедленный анализ.

   ```
   online_analyze.threshold = 50
   ```
4. Укажите процент от размера таблицы, при котором начнётся немедленный анализ.

   ```
   online_analyze.scale_factor = 0.1
   ```
5. Выполняйте команду `ANALYZE` без `VERBOSE`.

   ```
   online_analyze.verbose = off
   ```
6. Включите немедленный анализ для временных таблиц.

   ```
   online_analyze.table_type = 'temporary'
   ```
7. Укажите минимальный интервал времени между вызовами `ANALYZE` для каждой таблицы, в миллисекундах.

   ```
   online_analyze.min_interval = 10000
   ```

Приложение L. Ограничения Postgres Pro

В Таблице L.1 описываются различные жёсткие ограничения Postgres Pro. Однако раньше этих абсолютных лимитов на практике могут достигаться другие, например, предел производительности или доступного объёма дискового хранилища.

Таблица L.1. Ограничения Postgres Pro

Объект	Верхний предел	Комментарий
размер базы данных	без ограничений	
количество баз данных	4 294 950 911	
отношений в базе данных	1 431 650 303	
размер отношения	32 ТБ	со значением `BLCKSZ` по умолчанию, равным 8192 байта
строк в таблице	ограничивается количеством кортежей, которое может уместиться в 4 294 967 295 страниц	
столбцов в таблице	1600	дополнительно ограничивается размером кортежа, который может уместиться в одной странице; см. примечание ниже
столбцов в наборе результатов	1664	
размер поля	1 ГБ	
индексов в таблице	без ограничений	ограничивается максимальным количеством отношений в базе данных
столбцов в индексе	32	может быть увеличена при перекомпиляции Postgres Pro
ключей секционирования	32	может быть увеличена при перекомпиляции Postgres Pro
длина идентификатора	63 байта	может быть увеличена при перекомпиляции Postgres Pro
аргументы функции	100	может быть увеличена при перекомпиляции Postgres Pro
параметры запроса	65535	

Максимальное количество столбцов таблицы дополнительно уменьшается в связи с тем, что сохраняемый кортеж должен умещаться в одной странице размером 8192 байта. Например, если не учитывать размер заголовка, кортеж, состоящий из 1600 столбцов `int`, будет занимать 6400 байт и поместится в странице кучи, тогда как 1600 столбцов `bigint` займут 12800 байт и в одной странице не поместятся. Поля переменной длины, например типов `text`, `varchar` и `char`, могут храниться отдельно, в таблице TOAST, когда их значения достаточно велики для этого. При этом внутри кортежа кучи должен остаться только 18-байтовый указатель. Для более коротких значений полей переменной длины используется заголовок из 1 или 4 байт, и само значение сохраняется внутри кортежа в куче.

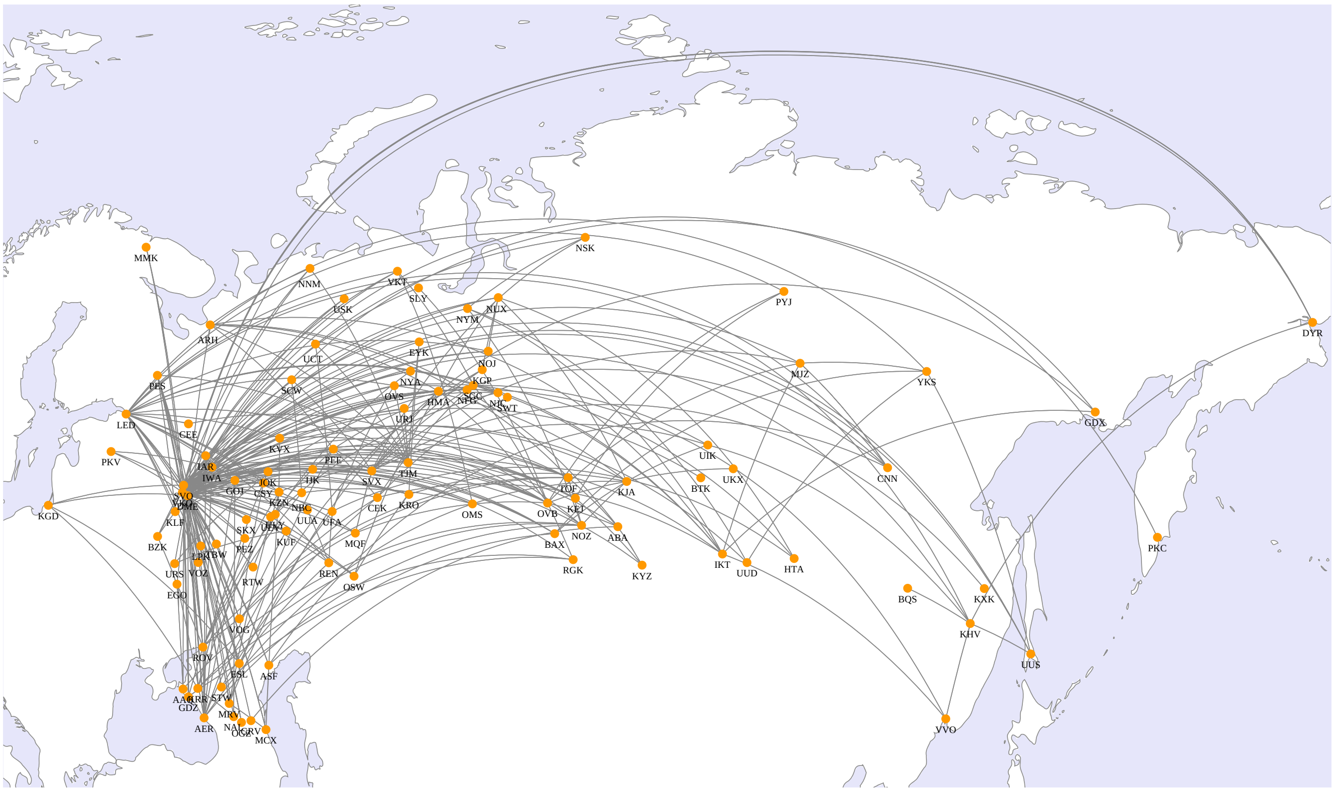
Максимальное количество столбцов может также зависеть от числа столбцов, удалённых из таблицы. И хотя значения удалённых столбцов для создаваемых впоследствии кортежей не хранятся, а только помечаются как NULL в специальной битовой карте, эта карта тоже занимает место.

Теоретически каждая таблица может хранить до 2^32 отделённых значений. Более подробно такие значения описаны в Разделе 63.2. Данное ограничение вызвано использованием 32-битных OID для идентификации каждого такого значения. В реальности же лимит значительно ниже, поскольку как только всё пространство для OID занято, найти свободный OID становится сложно, что замедляет работу операторов INSERT/UPDATE. Обычно это происходит только с таблицами объёмом в несколько терабайт. Возможным обходным решением может быть секционирование.

Приложение M. Демонстрационная база данных «Авиаперевозки»

Представляем вам демонстрационную базу данных для Postgres Pro. В этом приложении к документации описана схема данных, состоящая из восьми таблиц и нескольких представлений. В качестве предметной области выбраны авиаперевозки по России. Базу данных можно загрузить с *нашего сайта*, см. Раздел M.1.

Рисунок M.1. Воздушное сообщение в России



База данных может использоваться, например:

- для самостоятельного изучения языка запросов SQL;
- для подготовки книг, пособий и учебных курсов по языку SQL;
- для демонстрации возможностей Postgres Pro в статьях и заметках.

При разработке демонстрационной базы данных мы преследовали несколько целей:

- схема данных должна быть достаточно простой, чтобы быть понятной без особых пояснений;
- в то же время схема данных должна быть достаточно сложной, чтобы позволять строить осмысленные запросы;
- база данных должна быть наполнена данными, напоминающими реальные, с которыми будет интересно работать.

Демонстрационная база данных распространяется под *лицензией PostgreSQL*.

Свои замечания и пожелания направляйте нам по адресу *edu@postgrespro.ru*.

M.1. Установка

Демонстрационная база данных доступна на *edu.postgrespro.ru* в трёх версиях, которые отличаются только объёмом данных:

- *demo-small.zip* (21 МВ) — — данные по полётам за один месяц (размер БД около 300 МБ)
- *demo-medium.zip* (62 МБ) — данные по полётам за три месяца (размер БД около 700 МБ)
- *demo-big.zip* (232 МБ) — данные по полётам за год (размер БД около 2,5 ГБ)

Небольшая база годится для того, чтобы писать запросы, и при этом не займёт много места на диске. База большого размера позволит почувствовать, как ведут себя запросы на больших объёмах данных, и задуматься об оптимизации.

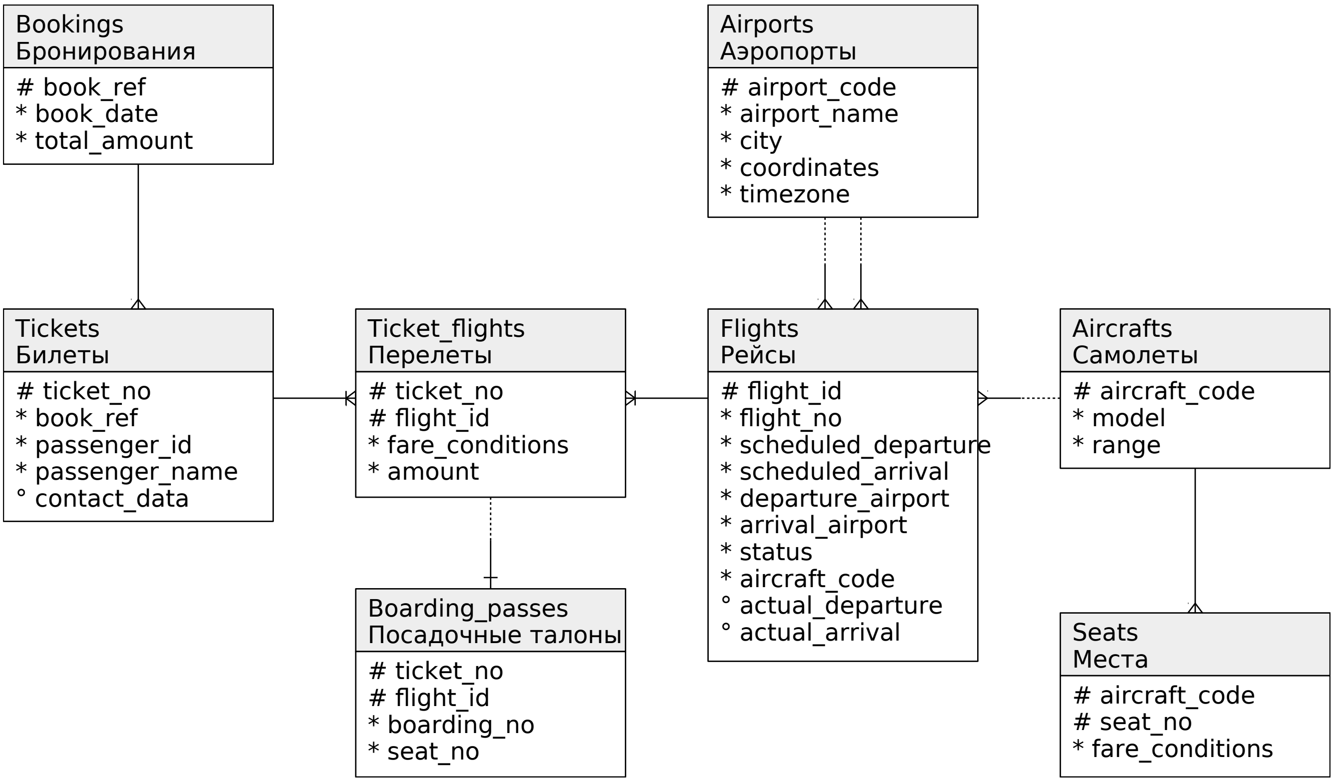
Файлы содержат SQL-скрипт, создающий базу данных `demo` и наполняющий её данными (фактически, это резервная копия, созданная утилитой pg_dump). Владельцем базы данных `demo` станет пользователь СУБД, выполнявший скрипт. Например, чтобы создать небольшую базу данных, запустите скрипт от имени пользователя `postgres` в программе psql:

```
psql -f demo_small_YYYYMMDD.sql -U postgres
```

Обратите внимание, что если база данных `demo` уже существует, она будет удалена и создана заново!

M.2. Диаграмма схемы данных

Рисунок M.2. Диаграмма схемы Bookings



M.3. Описание схемы

Основной сущностью является бронирование (`bookings`).

В одно бронирование можно включить несколько пассажиров, каждому из которых выписывается отдельный билет (`tickets`). Билет имеет уникальный номер и содержит информацию о пассажире. Как таковой пассажир не является отдельной сущностью. Как имя, так и номер документа пассажира могут меняться с течением времени, так что невозможно однозначно найти все билеты одного человека; для простоты можно считать, что все пассажиры уникальны.

Билет включает один или несколько перелетов (`ticket_flights`). Несколько перелетов могут включаться в билет в случаях, когда нет прямого рейса, соединяющего пункты отправления и назначения (полет с пересадками), либо когда билет взят «туда и обратно». В схеме данных нет жёсткого ограничения, но предполагается, что все билеты в одном бронировании имеют одинаковый набор перелетов.

Каждый рейс (`flights`) следует из одного аэропорта (`airports`) в другой. Рейсы с одним номером имеют одинаковые пункты вылета и назначения, но будут отличаться датой отправления.

При регистрации на рейс пассажиру выдаётся посадочный талон (`boarding_passes`), в котором указано место в самолете. Пассажир может зарегистрироваться только на тот рейс, который есть у него в билете. Комбинация рейса и места в самолете должна быть уникальной, чтобы не допустить выдачу двух посадочных талонов на одно место.

Количество мест (`seats`) в самолете и их распределение по классам обслуживания зависит от модели самолета (`aircrafts`), выполняющего рейс. Предполагается, что каждая модель самолета имеет только одну компоновку салона. Схема данных не контролирует, что места в посадочных талонах соответствуют имеющимся в самолете (такая проверка может быть сделана с использованием табличных триггеров или в приложении).

M.4. Объекты схемы

M.4.1. Список отношений

```
       Имя        |      Тип      | Small  | Medium |   Big  |       Описание
------------------+---------------+--------+--------+--------+-------------------------
 aircrafts        | представление |        |        |        | Самолеты
 aircrafts_data   | таблица       | 16 kB  |  16 kB |  16 kB | Самолеты (переводы)
 airports         | представление |        |        |        | Аэропорты
 airports_data    | таблица       | 56 kB  |  56 kB |  56 kB | Аэропорты (переводы)
 boarding_passes  | таблица       | 31 MB  | 102 MB | 427 MB | Посадочные талоны
 bookings         | таблица       | 13 MB  |  30 MB | 105 MB | Бронирования
 flights          | таблица       |  3 MB  |   6 MB |  19 MB | Рейсы
 flights_v        | представление |        |        |        | Рейсы
 routes           | представление |        |        |        | Маршруты
 seats            | таблица       | 88 kB  |  88 kB |  88 kB | Места
 ticket_flights   | таблица       | 64 MB  | 145 MB | 516 MB | Перелеты
 tickets          | таблица       | 47 MB  | 107 MB | 381 MB | Билеты
```

M.4.2. Представление `bookings.aircrafts`

Каждая модель воздушного судна идентифицируется своим трёхзначным кодом (`aircraft_code`). Указывается также название модели (`model`) и максимальная дальность полета в километрах (`range`).

Значение поля `model` определяется в зависимости от выбранного языка. За подробностями обратитесь к Подразделу M.4.15.

```
    Столбец     |   Тип    | Модификаторы |             Описание
----------------+----------+--------------+-----------------------------------
 aircraft_code  | char(3)  | not null     | Код самолета, IATA
 model          | text     | not null     | Модель самолета
 range          | integer  | not null     | Максимальная дальность полета, км
Определение представления:
 SELECT ml.aircraft_code,
    ml.model ->> lang() AS model,
    ml.range
   FROM aircrafts_data ml;
```

M.4.3. Таблица `bookings.aircrafts_data`

Это базовая таблица для представления `aircrafts`. Поле `model` этой таблицы содержит переводы моделей самолётов на разные языки, в формате JSONB. В большинстве случаев к этой таблице не следует обращаться напрямую.

```
    Столбец    |   Тип   | Модификаторы |             Описание
---------------+---------+--------------+-----------------------------------
 aircraft_code | char(3) | not null     | Код самолета, IATA
 model         | jsonb   | not null     | Модель самолета
 range         | integer | not null     | Максимальная дальность полета, км
Индексы:
    PRIMARY KEY, btree (aircraft_code)
Ограничения-проверки:
    CHECK (range > 0)
Ссылки извне:
    TABLE "flights" FOREIGN KEY (aircraft_code)
        REFERENCES aircrafts_data(aircraft_code)
    TABLE "seats" FOREIGN KEY (aircraft_code)
        REFERENCES aircrafts_data(aircraft_code) ON DELETE CASCADE
```

M.4.4. Представление `bookings.airports`

Аэропорт идентифицируется трехбуквенным кодом (`airport_code`) и имеет своё имя (`airport_name`).

Для города не предусмотрено отдельной сущности, но введено поле с названием города (`city`), позволяющее найти аэропорты одного города. Это представление также включает координаты аэропорта (`coordinates`) и часовой пояс (`timezone`).

Значения полей `airport_name` и `city` определяются в зависимости от выбранного языка. За подробностями обратитесь к Подразделу M.4.15.

```
   Столбец    |   Тип   | Модификаторы |                 Описание
--------------+---------+--------------+--------------------------------------------
 airport_code | char(3) | not null     | Код аэропорта
 airport_name | text    | not null     | Название аэропорта
 city         | text    | not null     | Город
 coordinates  | point   | not null     | Координаты аэропорта (долгота и широта)
 timezone     | text    | not null     | Часовой пояс аэропорта
Определение представления:
 SELECT ml.airport_code,
    ml.airport_name ->> lang() AS airport_name,
    ml.city ->> lang() AS city,
    ml.coordinates,
    ml.timezone
   FROM airports_data ml;
```

M.4.5. Таблица `bookings.airports_data`

Это базовая таблица для представления `airports`. Она содержит переводы значений `airport_name` и `city` на разные языки, в формате JSONB. В большинстве случаев к этой таблице не следует обращаться напрямую.

```
   Столбец    |   Тип   | Модификаторы |                 Описание
```

```
--------------+---------+-------------+-----------------------------------------
 airport_code | char(3) | not null    | Код аэропорта
 airport_name | jsonb   | not null    | Название аэропорта
 city         | jsonb   | not null    | Город
 coordinates  | point   | not null    | Координаты аэропорта (долгота и широта)
 timezone     | text    | not null    | Часовой пояс аэропорта
Индексы:
    PRIMARY KEY, btree (airport_code)
Ссылки извне:
    TABLE "flights" FOREIGN KEY (arrival_airport)
        REFERENCES airports_data(airport_code)
    TABLE "flights" FOREIGN KEY (departure_airport)
        REFERENCES airports_data(airport_code)
```

M.4.6. Таблица `bookings.boarding_passes`

При регистрации на рейс, которая возможна за сутки до плановой даты отправления, пассажиру выдаётся посадочный талон. Он идентифицируется также, как и перелёт — номером билета и номером рейса.

Посадочным талонам присваиваются последовательные номера (`boarding_no`) в порядке регистрации пассажиров на рейс (этот номер будет уникальным только в пределах данного рейса). В посадочном талоне указывается номер места (`seat_no`).

```
   Столбец   |    Тип     | Модификаторы |         Описание
-------------+------------+--------------+--------------------------
 ticket_no   | char(13)   | not null     | Номер билета
 flight_id   | integer    | not null     | Идентификатор рейса
 boarding_no | integer    | not null     | Номер посадочного талона
 seat_no     | varchar(4) | not null     | Номер места
Индексы:
    PRIMARY KEY, btree (ticket_no, flight_id)
    UNIQUE CONSTRAINT, btree (flight_id, boarding_no)
    UNIQUE CONSTRAINT, btree (flight_id, seat_no)
Ограничения внешнего ключа:
    FOREIGN KEY (ticket_no, flight_id)
        REFERENCES ticket_flights(ticket_no, flight_id)
```

M.4.7. Таблица `bookings.bookings`

Пассажир заранее (`book_date`, максимум за месяц до рейса) бронирует билет себе и, возможно, нескольким другим пассажирам. Бронирование идентифицируется номером (`book_ref`, шестизначная комбинация букв и цифр).

Поле `total_amount` хранит общую стоимость включённых в бронирование перелетов всех пассажиров.

```
   Столбец    |      Тип      | Модификаторы |         Описание
--------------+---------------+--------------+---------------------------
 book_ref     | char(6)       | not null     | Номер бронирования
 book_date    | timestamptz   | not null     | Дата бронирования
 total_amount | numeric(10,2) | not null     | Полная сумма бронирования
Индексы:
    PRIMARY KEY, btree (book_ref)
Ссылки извне:
    TABLE "tickets" FOREIGN KEY (book_ref) REFERENCES bookings(book_ref)
```

M.4.8. Таблица `bookings.flights`

Естественный ключ таблицы рейсов состоит из двух полей — номера рейса (`flight_no`) и даты отправления (`scheduled_departure`). Чтобы сделать внешние ключи на эту таблицу компактнее, в качестве первичного используется суррогатный ключ (`flight_id`).

Рейс всегда соединяет две точки — аэропорты вылета (`departure_airport`) и прибытия (`arrival_airport`). Такое понятие, как «рейс с пересадками» отсутствует: если из одного аэропорта до другого нет прямого рейса, в билет просто включаются несколько необходимых рейсов.

У каждого рейса есть запланированные дата и время вылета (`scheduled_departure`) и прибытия (`scheduled_arrival`). Реальные время вылета (`actual_departure`) и прибытия (`actual_arrival`) могут отличаться: обычно не сильно, но иногда и на несколько часов, если рейс задержан.

Статус рейса (`status`) может принимать одно из следующих значений:

`Scheduled`

Рейс доступен для бронирования. Это происходит за месяц до плановой даты вылета; до этого запись о рейсе не существует в базе данных.

`On Time`

Рейс доступен для регистрации (за сутки до плановой даты вылета) и не задержан.

`Delayed`

Рейс доступен для регистрации (за сутки до плановой даты вылета), но задержан.

`Departed`

Самолет уже вылетел и находится в воздухе.

`Arrived`

Самолет прибыл в пункт назначения.

`Cancelled`

Рейс отменён.

```
       Столбец       |     Тип     | Модификаторы |         Описание
---------------------+-------------+--------------+-----------------------------
 flight_id           | serial      | not null     | Идентификатор рейса
 flight_no           | char(6)     | not null     | Номер рейса
 scheduled_departure | timestamptz | not null     | Время вылета по расписанию
 scheduled_arrival   | timestamptz | not null     | Время прилёта по расписанию
 departure_airport   | char(3)     | not null     | Аэропорт отправления
 arrival_airport     | char(3)     | not null     | Аэропорт прибытия
 status              | varchar(20) | not null     | Статус рейса
 aircraft_code       | char(3)     | not null     | Код самолета, IATA
 actual_departure    | timestamptz |              | Фактическое время вылета
 actual_arrival      | timestamptz |              | Фактическое время прилёта
Индексы:
    PRIMARY KEY, btree (flight_id)
    UNIQUE CONSTRAINT, btree (flight_no, scheduled_departure)
Ограничения-проверки:
    CHECK (scheduled_arrival > scheduled_departure)
    CHECK ((actual_arrival IS NULL)
       OR  ((actual_departure IS NOT NULL AND actual_arrival IS NOT NULL)
            AND (actual_arrival > actual_departure)))
    CHECK (status IN ('On Time', 'Delayed', 'Departed',
                      'Arrived', 'Scheduled', 'Cancelled'))
Ограничения внешнего ключа:
```

```
    FOREIGN KEY (aircraft_code)
        REFERENCES aircrafts(aircraft_code)
    FOREIGN KEY (arrival_airport)
        REFERENCES airports(airport_code)
    FOREIGN KEY (departure_airport)
        REFERENCES airports(airport_code)
Ссылки извне:
    TABLE "ticket_flights" FOREIGN KEY (flight_id)
        REFERENCES flights(flight_id)
```

M.4.9. Таблица `bookings.seats`

Места определяют схему салона каждой модели. Каждое место определяется своим номером (`seat_no`) и имеет закреплённый за ним класс обслуживания (`fare_conditions`) — `Economy`, `Comfort` или `Business`.

```
      Столбец     |     Тип     | Модификаторы |      Описание
-----------------+-------------+--------------+--------------------
 aircraft_code   | char(3)     | not null     | Код самолета, IATA
 seat_no         | varchar(4)  | not null     | Номер места
 fare_conditions | varchar(10) | not null     | Класс обслуживания
Индексы:
    PRIMARY KEY, btree (aircraft_code, seat_no)
Ограничения-проверки:
    CHECK (fare_conditions IN ('Economy', 'Comfort', 'Business'))
Ограничения внешнего ключа:
    FOREIGN KEY (aircraft_code)
        REFERENCES aircrafts(aircraft_code) ON DELETE CASCADE
```

M.4.10. Таблица `bookings.ticket_flights`

Перелёт соединяет билет с рейсом и идентифицируется их номерами.

Для каждого перелета указываются его стоимость (`amount`) и класс обслуживания (`fare_conditions`).

```
     Столбец     |      Тип       | Модификаторы |    Описание
-----------------+----------------+--------------+---------------------
 ticket_no       | char(13)       | not null     | Номер билета
 flight_id       | integer        | not null     | Идентификатор рейса
 fare_conditions | varchar(10)    | not null     | Класс обслуживания
 amount          | numeric(10,2)  | not null     | Стоимость перелета
Индексы:
    PRIMARY KEY, btree (ticket_no, flight_id)
Ограничения-проверки:
    CHECK (amount >= 0)
    CHECK (fare_conditions IN ('Economy', 'Comfort', 'Business'))
Ограничения внешнего ключа:
    FOREIGN KEY (flight_id) REFERENCES flights(flight_id)
    FOREIGN KEY (ticket_no) REFERENCES tickets(ticket_no)
Ссылки извне:
    TABLE "boarding_passes" FOREIGN KEY (ticket_no, flight_id)
        REFERENCES ticket_flights(ticket_no, flight_id)
```

M.4.11. Таблица `bookings.tickets`

Билет имеет уникальный номер (`ticket_no`), состоящий из 13 цифр.

Билет содержит идентификатор пассажира (`passenger_id`) — номер документа, удостоверяющего личность, — его фамилию и имя (`passenger_name`) и контактную информацию (`contact_data`).

Ни идентификатор пассажира, ни имя не являются постоянными (можно поменять паспорт, можно сменить фамилию), поэтому однозначно найти все билеты одного и того же пассажира невозможно.

```
     Столбец    |     Тип     | Модификаторы |          Описание
----------------+-------------+--------------+-----------------------------
 ticket_no      | char(13)    | not null     | Номер билета
 book_ref       | char(6)     | not null     | Номер бронирования
 passenger_id   | varchar(20) | not null     | Идентификатор пассажира
 passenger_name | text        | not null     | Имя пассажира
 contact_data   | jsonb       |              | Контактные данные пассажира
Индексы:
    PRIMARY KEY, btree (ticket_no)
Ограничения внешнего ключа:
    FOREIGN KEY (book_ref) REFERENCES bookings(book_ref)
Ссылки извне:
    TABLE "ticket_flights" FOREIGN KEY (ticket_no) REFERENCES tickets(ticket_no)
```

M.4.12. Представление `bookings.flights_v`

Над таблицей `flights` создано представление `flights_v`, содержащее дополнительную информацию:

- расшифровку данных об аэропорте вылета — `departure_airport`, `departure_airport_name`, `departure_city`
- расшифровку данных об аэропорте прибытия — `arrival_airport`, `arrival_airport_name`, `arrival_city`
- местное время вылета — `scheduled_departure_local`, `actual_departure_local`
- местное время прибытия — `scheduled_arrival_local`, `actual_arrival_local`
- продолжительность полета — `scheduled_duration`, `actual_duration`.

```
          Столбец          |     Тип     |              Описание
---------------------------+-------------+--------------------------------------
 flight_id                 | integer     | Идентификатор рейса
 flight_no                 | char(6)     | Номер рейса
 scheduled_departure       | timestamptz | Время вылета по расписанию
 scheduled_departure_local | timestamp   | Время вылета по расписанию,
                           |             | местное время в пункте отправления
 scheduled_arrival         | timestamptz | Время прилёта по расписанию
 scheduled_arrival_local   | timestamp   | Время прилёта по расписанию,
                           |             | местное время в пункте прибытия
 scheduled_duration        | interval    | Планируемая продолжительность полета
 departure_airport         | char(3)     | Код аэропорта отправления
 departure_airport_name    | text        | Название аэропорта отправления
 departure_city            | text        | Город отправления
 arrival_airport           | char(3)     | Код аэропорта прибытия
 arrival_airport_name      | text        | Название аэропорта прибытия
 arrival_city              | text        | Город прибытия
 status                    | varchar(20) | Статус рейса
 aircraft_code             | char(3)     | Код самолета, IATA
 actual_departure          | timestamptz | Фактическое время вылета
 actual_departure_local    | timestamp   | Фактическое время вылета,
                           |             | местное время в пункте отправления
```

```
 actual_arrival           | timestamptz | Фактическое время прилёта
 actual_arrival_local     | timestamp   | Фактическое время прилёта,
                          |             | местное время в пункте прибытия
 actual_duration          | interval    | Фактическая продолжительность полета
```

M.4.13. Представление `bookings.routes`

Таблица рейсов (`bookings.flights`) содержит избыточность: из неё можно было бы выделить информацию о маршруте (номер рейса, аэропорты отправления и назначения), которая не зависит от конкретных дат рейсов.

Именно такая информация составляет представление `routes`.

```
        Столбец          |    Тип     |              Описание
-------------------------+------------+-------------------------------------
 flight_no               | char(6)    | Номер рейса
 departure_airport       | char(3)    | Код аэропорта отправления
 departure_airport_name  | text       | Название аэропорта отправления
 departure_city          | text       | Город отправления
 arrival_airport         | char(3)    | Код аэропорта прибытия
 arrival_airport_name    | text       | Название аэропорта прибытия
 arrival_city            | text       | Город прибытия
 aircraft_code           | char(3)    | Код самолета, IATA
 duration                | interval   | Продолжительность полета
 days_of_week            | integer[]  | Дни недели, когда выполняются рейсы
```

M.4.14. Функция `bookings.now`

Демонстрационная база содержит временной «срез» данных — так, как будто в некоторый момент была сделана резервная копия реальной системы. Например, если некоторый рейс имеет статус `Departed`, это означает, что в момент резервного копирования самолет вылетел и находился в воздухе.

Позиция «среза» сохранена в функции `bookings.now()` function. Ей можно пользоваться в запросах там, где в обычной жизни использовалась бы функция `now()`.

Кроме того, значение этой функции определяет версию демонстрационной базы данных. Актуальная версия на текущий момент — от 15 августа 2017 г.

M.4.15. Функция `bookings.lang`

Некоторые поля в демонстрационной базе содержат текст на английском и русском языках. Переводы на другие языки отсутствуют, но их несложно добавить. Функция `bookings.lang` возвращает значение параметра *`bookings.lang`*, то есть язык, на котором будут выдаваться значения этих полей.

Эта функция используется в представлениях `aircrafts` и `airports` и не предназначена для непосредственного использования в запросах.

M.5. Использование

M.5.1. Схема `bookings`

Схема `bookings` содержит все объекты демонстрационной базы. При подключении к базе параметр конфигурации `search_path` автоматически принимает значение `bookings, public`, так что указывать имя схемы явно необязательно.

Однако для функции `bookings.now` в любом случае необходимо явно указывать схему, чтобы отличать её от стандартной функции `now`.

M.5.2. Переводы

По умолчанию значения различных переводимых полей выдаются на русском языке. Это поля `airport_name` и `city` представления `airports`, а также поле `model` представления `aircrafts`.

Вы можете выбрать другой язык для отображения этих полей (хотя в данной базе представлен только английский перевод). Например, для переключения на английский установите для параметра *`bookings.lang`* значение `en`. Также может быть удобно выбрать язык на уровне базы данных:

```
ALTER DATABASE demo SET bookings.lang = en;
```

Чтобы это изменение вступило в силу, необходимо переподключиться к базе данных. Другие варианты установления параметров конфигурации описаны в Разделе 18.1.

В следующих примерах для переводимых полей выбран русский язык.

M.5.3. Примеры запросов

Чтобы лучше познакомиться с содержимым демонстрационной базы данных, посмотрим на результаты нескольких простых запросов.

Результаты, представленные ниже, были получены для версии с небольшой базой данных (demo-small) от 15 августа 2017 г. Если в вашей системе запросы выдают другие данные, проверьте версию демонстрационной базы (функция `bookings.now`). Незначительные отклонения могут быть связаны с местным временем, отличным от московского, и настройками локализации.

Все рейсы выполняются несколькими типами самолетов:

```
SELECT * FROM aircrafts;
```

```
 aircraft_code |        model        | range
---------------+---------------------+-------
 773           | Boeing 777-300      | 11100
 763           | Boeing 767-300      |  7900
 SU9           | Sukhoi SuperJet-100 |  3000
 320           | Airbus A320-200     |  5700
 321           | Airbus A321-200     |  5600
 319           | Airbus A319-100     |  6700
 733           | Boeing 737-300      |  4200
 CN1           | Cessna 208 Caravan  |  1200
 CR2           | Bombardier CRJ-200  |  2700
(9 строк)
```

Для каждого типа самолета поддерживается список мест в салоне. Например, вот где можно разместиться в небольшом самолете Cessna 208 Caravan:

```
SELECT   a.aircraft_code,
         a.model,
         s.seat_no,
         s.fare_conditions
FROM     aircrafts a
         JOIN seats s ON a.aircraft_code = s.aircraft_code
WHERE    a.model = 'Cessna 208 Caravan'
ORDER BY s.seat_no;
```

```
 aircraft_code |       model        | seat_no | fare_conditions
```

```
---------------+--------------------+---------+-----------------
 CN1           | Cessna 208 Caravan | 1A      | Economy
 CN1           | Cessna 208 Caravan | 1B      | Economy
 CN1           | Cessna 208 Caravan | 2A      | Economy
 CN1           | Cessna 208 Caravan | 2B      | Economy
 CN1           | Cessna 208 Caravan | 3A      | Economy
 CN1           | Cessna 208 Caravan | 3B      | Economy
 CN1           | Cessna 208 Caravan | 4A      | Economy
 CN1           | Cessna 208 Caravan | 4B      | Economy
 CN1           | Cessna 208 Caravan | 5A      | Economy
 CN1           | Cessna 208 Caravan | 5B      | Economy
 CN1           | Cessna 208 Caravan | 6A      | Economy
 CN1           | Cessna 208 Caravan | 6B      | Economy
(12 строк)
```

Самолеты большего размера имеют больше посадочных мест с разными классами обслуживания:

```
SELECT   s2.aircraft_code,
         string_agg (s2.fare_conditions || '(' || s2.num::text || ')',
                     ', ') as fare_conditions
FROM     (
          SELECT   s.aircraft_code, s.fare_conditions, count(*) as num
          FROM     seats s
          GROUP BY s.aircraft_code, s.fare_conditions
          ORDER BY s.aircraft_code, s.fare_conditions
         ) s2
GROUP BY s2.aircraft_code
ORDER BY s2.aircraft_code;
```

```
 aircraft_code |             fare_conditions
---------------+-----------------------------------------
 319           | Business(20), Economy(96)
 320           | Business(20), Economy(120)
 321           | Business(28), Economy(142)
 733           | Business(12), Economy(118)
 763           | Business(30), Economy(192)
 773           | Business(30), Comfort(48), Economy(324)
 CN1           | Economy(12)
 CR2           | Economy(50)
 SU9           | Business(12), Economy(85)
(9 строк)
```

База данных содержит список аэропортов практически всех крупных городов России. В большинстве городов есть только один аэропорт. Исключение составляют:

```
SELECT   a.airport_code as code,
         a.airport_name,
         a.city,
         a.coordinates
FROM     airports a
WHERE    a.city IN (
            SELECT   aa.city
            FROM     airports aa
            GROUP BY aa.city
            HAVING   COUNT(*) > 1
         )
ORDER BY a.city, a.airport_code;
```

```
 code |      airport_name       |   city    |           coordinates
------+-------------------------------------+-----------+-------------------------
 DME  | Домодедово              | Москва    | (37.9062995910645,55.4087982177734)
 SVO  | Шереметьево             | Москва    | (37.4146,55.972599)
 VKO  | Внуково                 | Москва    | (37.2615013123,55.5914993286)
 ULV  | Баратаевка              | Ульяновск | (48.2266998291,54.2682991028)
 ULY  | Ульяновск-Восточный     | Ульяновск | (48.8027000427246,54.4010009765625)
(5 строк)
```

Чтобы понять, откуда и куда можно улететь, удобно использовать материализованное представление `routes`, в котором агрегируется информация о всех рейсах. Вот, например, куда, в какие дни недели и за какое время можно долететь из Волгограда:

```
SELECT r.arrival_city as city,
       r.arrival_airport as code,
       r.arrival_airport_name as airport_name,
       r.days_of_week,
       r.duration
FROM   routes r
WHERE  r.departure_city = 'Волгоград';
```

```
    city        | code |            airport_name             |  days_of_week   |
 duration
----------------+------+-------------------------------------+-----------------
+----------
 Москва         | SVO  | Шереметьево                         | {1,2,3,4,5,6,7} |
 01:15:00
 Челябинск      | CEK  | Челябинск                           | {1,2,3,4,5,6,7} |
 01:50:00
 Ростов-на-Дону | ROV  | Ростов-на-Дону                      | {1,2,3,4,5,6,7} |
 00:30:00
 Москва         | VKO  | Внуково                             | {1,2,3,4,5,6,7} |
 01:10:00
 Чебоксары      | CSY  | Чебоксары                           | {1,2,3,4,5,6,7} |
 02:45:00
 Томск          | TOF  | Богашёво                            | {1}             |
 03:50:00
(6 строк)
```

База данных была сформирована на момент времени, возвращаемый функцией `bookings.now()`:

```
SELECT bookings.now() as now;
```

```
          now
------------------------
 2017-08-15 18:00:00+03
```

Относительно именно этого момента времени все рейсы делятся на прошедшие и будущие:

```
SELECT   status,
         count(*) as count,
         min(scheduled_departure) as min_scheduled_departure,
         max(scheduled_departure) as max_scheduled_departure
FROM     flights
GROUP BY status
```

```
ORDER BY min_scheduled_departure;

  status   | count | min_scheduled_departure | max_scheduled_departure
-----------+-------+-------------------------+------------------------
 Arrived   | 16707 | 2017-07-16 01:50:00+03  | 2017-08-15 17:25:00+03
 Cancelled |   414 | 2017-07-19 11:35:00+03  | 2017-09-14 20:55:00+03
 Departed  |    58 | 2017-08-15 09:55:00+03  | 2017-08-15 17:50:00+03
 Delayed   |    41 | 2017-08-15 15:15:00+03  | 2017-08-16 17:25:00+03
 On Time   |   518 | 2017-08-15 17:55:00+03  | 2017-08-16 18:00:00+03
 Scheduled | 15383 | 2017-08-16 18:05:00+03  | 2017-09-14 20:40:00+03
(6 строк)
```

Найдем ближайший рейс, вылетающий из Екатеринбурга в Москву. Использовать для такого запроса таблицу `flight` не очень удобно, так как в ней нет информации о городах отправления и прибытия. Поэтому воспользуемся представлением `flights_v`:

```
\x
SELECT   f.*
FROM     flights_v f
WHERE    f.departure_city = 'Екатеринбург'
AND      f.arrival_city = 'Москва'
AND      f.scheduled_departure > bookings.now()
ORDER BY f.scheduled_departure
LIMIT    1;

-[ RECORD 1 ]-------------+-----------------------------------
flight_id                 | 10927
flight_no                 | PG0226
scheduled_departure       | 2017-08-16 08:10:00+03
scheduled_departure_local | 2017-08-16 10:10:00
scheduled_arrival         | 2017-08-16 09:55:00+03
scheduled_arrival_local   | 2017-08-16 09:55:00
scheduled_duration        | 01:45:00
departure_airport         | SVX
departure_airport_name    | Кольцово
departure_city            | Екатеринбург
arrival_airport           | SVO
arrival_airport_name      | Шереметьево
arrival_city              | Москва
status                    | On Time
aircraft_code             | 773
actual_departure          |
actual_departure_local    |
actual_arrival            |
actual_arrival_local      |
actual_duration           |
```

Обратите внимание, что в представлении `flights_v` указано не только московское время, но и местное время в аэропортах вылета и прилета.

M.5.4. Бронирования

Каждое бронирование может включать несколько билетов, по одному на каждого пассажира. Билет, в свою очередь, может включать несколько перелетов. Полная информация о бронировании находится в трёх таблицах: `bookings`, `tickets` и `ticket_flights`.

Найдём несколько бронирований с самой высокой стоимостью:

```
SELECT   *
FROM     bookings
ORDER BY total_amount desc
LIMIT    10;

 book_ref |       book_date        | total_amount
----------+------------------------+--------------
 3B54BB   | 2017-07-05 17:08:00+03 |   1204500.00
 3AC131   | 2017-07-31 01:06:00+03 |   1087100.00
 65A6EA   | 2017-07-03 06:28:00+03 |   1065600.00
 D7E9AA   | 2017-08-08 05:29:00+03 |   1062800.00
 EF479E   | 2017-08-02 15:58:00+03 |   1035100.00
 521C53   | 2017-07-08 09:25:00+03 |    985500.00
 514CA6   | 2017-07-27 05:07:00+03 |    955000.00
 D70BD9   | 2017-07-05 12:47:00+03 |    947500.00
 EC7EDA   | 2017-07-02 16:13:00+03 |    946800.00
 8E4370   | 2017-07-28 02:04:00+03 |    945700.00
(10 строк)
```

Посмотрим, из каких билетов состоит бронирование с кодом `521C53`:

```
SELECT ticket_no,
       passenger_id,
       passenger_name
FROM   tickets
WHERE  book_ref = '521C53';

   ticket_no   | passenger_id |   passenger_name
---------------+--------------+--------------------
 0005432661914 | 8234 547529  | IVAN IVANOV
 0005432661915 | 2034 201228  | ANTONINA KUZNECOVA
(2 строки)
```

Если нас интересует, какие перелеты включены в билет Антонины Кузнецовой, то это можно узнать запросом:

```
SELECT   to_char(f.scheduled_departure, 'DD.MM.YYYY') AS when,
         f.departure_city || ' (' || f.departure_airport || ')' AS departure,
         f.arrival_city || ' (' || f.arrival_airport || ')' AS arrival,
         tf.fare_conditions AS class,
         tf.amount
FROM     ticket_flights tf
         JOIN flights_v f ON tf.flight_id = f.flight_id
WHERE    tf.ticket_no = '0005432661915'
ORDER BY f.scheduled_departure;

    when    |      departure       |       arrival        |  class   |  amount
------------+----------------------+----------------------+----------+-----------
 29.07.2017 | Москва (SVO)         | Анадырь (DYR)        | Business | 185300.00
 02.08.2017 | Анадырь (DYR)        | Khabarovsk (KHV)     | Business |  92200.00
 03.08.2017 | Хабаровск (KHV)      | Благовещенск (BQS)   | Business |  18000.00
 08.08.2017 | Благовещенск (BQS)   | Хабаровск (KHV)      | Business |  18000.00
 12.08.2017 | Хабаровск (KHV)      | Анадырь (DYR)        | Economy  |  30700.00
 17.08.2017 | Анадырь (DYR)        | Москва (SVO)         | Business | 185300.00
(6 строк)
```

Как видим, высокая стоимость бронирования объясняется большим количеством перелётов на дальние расстояния бизнес-классом.

Часть перелётов в этом билете имеет более ранние даты, чем значение `bookings.now()`: это значит, что они уже выполнены. А последний полет ещё предстоит. После регистрации на рейс выписывается посадочный талон с указанием места в самолете. Мы можем посмотреть какие именно места занимала Антонина (обратите внимание на внешнее левое соединение с таблицей `boarding_passes`):

```
SELECT   to_char(f.scheduled_departure, 'DD.MM.YYYY') AS when,
         f.departure_city || ' (' || f.departure_airport || ')' AS departure,
         f.arrival_city || ' (' || f.arrival_airport || ')' AS arrival,
         f.status,
         bp.seat_no
FROM     ticket_flights tf
         JOIN flights_v f ON tf.flight_id = f.flight_id
         LEFT JOIN boarding_passes bp ON tf.flight_id = bp.flight_id
                                     AND tf.ticket_no = bp.ticket_no
WHERE    tf.ticket_no = '0005432661915'
ORDER BY f.scheduled_departure;
```

```
    when    |      departure       |       arrival        |  status   | seat_no
------------+----------------------+----------------------+-----------+---------
 29.07.2017 | Москва (SVO)         | Анадырь (DYR)        | Arrived   | 5C
 02.08.2017 | Анадырь (DYR)        | Хабаровск (KHV)      | Arrived   | 1D
 03.08.2017 | Хабаровск (KHV)      | Благовещенск (BQS)   | Arrived   | 2C
 08.08.2017 | Благовещенск(BQS)    | Хабаровск(KHV)       | Arrived   | 2D
 12.08.2017 | Хабаровск(KHV)       | Анадырь(DYR)         | Arrived   | 20B
 17.08.2017 | Анадырь(DYR)         | Москва(SVO)          | Scheduled |
(6 строк)
```

M.5.5. Новое бронирование

Попробуем отправить Александра Николаевича Радищева по маршруту, который принес ему известность. Разумеется бесплатно и бизнес-классом. Предварительно найдем «завтрашний» рейс, а также позаботимся о возвращении через неделю.

```
BEGIN;

INSERT INTO bookings (book_ref, book_date, total_amount)
VALUES      ('_QWE12', bookings.now(), 0);

INSERT INTO tickets (ticket_no, book_ref, passenger_id, passenger_name)
VALUES      ('_000000000001', '_QWE12', '1749 051790', 'ALEKSANDR RADISHCHEV');

INSERT INTO ticket_flights (ticket_no, flight_id, fare_conditions, amount)
VALUES      ('_000000000001', 8525, 'Business', 0),
            ('_000000000001', 4967, 'Business', 0);

COMMIT;
```

Мы начинаем идентификаторы с подчёркивания, чтобы не пересекаться с диапазоном значений, присутствующих в базе.

Сразу зарегистрируемся на завтрашний рейс:

```
INSERT INTO boarding_passes (ticket_no, flight_id, boarding_no, seat_no)
VALUES      ('_000000000001', 8525, 1, '1A');
```

Проверим информацию о созданном бронировании:

```
SELECT   b.book_ref,
         t.ticket_no,
         t.passenger_id,
         t.passenger_name,
         tf.fare_conditions,
         tf.amount,
         f.scheduled_departure_local,
         f.scheduled_arrival_local,
         f.departure_city || ' (' || f.departure_airport || ')' AS departure,
         f.arrival_city || ' (' || f.arrival_airport || ')' AS arrival,
         f.status,
         bp.seat_no
FROM     bookings b
         JOIN tickets t ON b.book_ref = t.book_ref
         JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
         JOIN flights_v f ON tf.flight_id = f.flight_id
         LEFT JOIN boarding_passes bp ON tf.flight_id = bp.flight_id
                                     AND tf.ticket_no = bp.ticket_no
WHERE    b.book_ref = '_QWE12'
ORDER BY t.ticket_no, f.scheduled_departure;
```

```
-[ RECORD 1 ]-------------+---------------------
book_ref                  | _QWE12
ticket_no                 | _000000000001
passenger_id              | 1749 051790
passenger_name            | ALEKSANDR RADISHCHEV
fare_conditions           | Business
amount                    | 0.00
scheduled_departure_local | 2017-08-16 09:45:00
scheduled_arrival_local   | 2017-08-16 10:35:00
departure                 | Санкт-Петербург (LED)
arrival                   | Москва (SVO)
status                    | On Time
seat_no                   | 1A
-[ RECORD 2 ]-------------+---------------------
book_ref                  | _QWE12
ticket_no                 | _000000000001
passenger_id              | 1749 051790
passenger_name            | ALEKSANDR RADISHCHEV
fare_conditions           | Business
amount                    | 0.00
scheduled_departure_local | 2017-08-23 10:20:00
scheduled_arrival_local   | 2017-08-23 11:10:00
departure                 | Москва (SVO)
arrival                   | Санкт-Петербург (LED)
status                    | Scheduled
seat_no                   |
```

Надеемся, что эти несколько простых примеров помогли составить представление о содержимом демонстрационной базы данных.

Приложение N. Сокращения

Ниже перечислены сокращения, часто используемые в документации Postgres Pro, и в обсуждениях, связанных с Postgres Pro.

AM

Access Method, Метод доступа

ANSI

American National Standards Institute, Американский национальный институт стандартов

API

Application Programming Interface, *Интерфейс программирования приложений*

ASCII

American Standard Code for Information Interchange, *Американский стандартный код для обмена информацией*

BKI

Backend Interface, Внутренний интерфейс

CA

Certificate Authority, *Центр сертификации*

CIDR

Classless Inter-Domain Routing, *Бесклассовая междоменная маршрутизация*

CPAN

Comprehensive Perl Archive Network, Всеобъемлющая сеть архивов Perl

CRL

Certificate Revocation List, Список отозванных сертификатов

CSV

Comma Separated Values, *Значения, разделённые запятыми*

CTE

Common Table Expression, Общее табличное выражение

CVE

Common Vulnerabilities and Exposures, Общие уязвимости и риски

DBA

Database Administrator, *Администратор баз данных*

DBI

Database Interface (Perl), Интерфейс баз данных (Perl)

DBMS

Database Management System, *Система управления базами данных*

DDL

Data Definition Language, *Язык описания данных*, включающий такие команды SQL, как `CREATE TABLE`, `ALTER USER`

DML

Data Manipulation Language, *Язык обработки данных*, включающий такие SQL-команды, как `INSERT`, `UPDATE`, `DELETE`

DST

Daylight Saving Time, *Летнее время*

ECPG

Embedded C for Postgres Pro, Встроенный C для Postgres Pro

ESQL

Embedded SQL, *Встраиваемый SQL*

FAQ

Frequently Asked Questions, *Часто задаваемые вопросы*

FSM

Free Space Map, Карта свободного пространства

GEQO

Genetic Query Optimizer, Генетический оптимизатор запросов

GIN

Generalized Inverted Index, Обобщённый инвертированный индекс

GiST

Generalized Search Tree, Обобщённое дерево поиска

Git

Git

GMT

Greenwich Mean Time, *Среднее время по Гринвичу*

GSSAPI

Generic Security Services Application Programming Interface, *Универсальный интерфейс программирования приложений служб безопасности*

GUC

Grand Unified Configuration, Главная унифицированная конфигурация, подсистема Postgres Pro, управляющая конфигурацией сервера

HBA

Host-Based Authentication, Аутентификация по сетевым узлам

HOT

Кортежи только в куче

IEC

International Electrotechnical Commission, *Международная электротехническая комиссия*

IEEE

Institute of Electrical and Electronics Engineers, Институт инженеров по электротехнике и электронике

IPC

Inter-Process Communication, *Межпроцессное взаимодействие*

ISO

International Organization for Standardization, Международная организация по стандартизации

ISSN

International Standard Serial Number, *Международный стандартный серийный номер*

JDBC

Java Database Connectivity, *Соединение с базами данных на Java*

JIT

Just-in-Time compilation, *JIT-компиляция*

JSON

>JavaScript Object Notation, *Представление объектов в JavaScript*

LDAP

Lightweight Directory Access Protocol, *Облегчённый протокол доступа к каталогам*

LSN

Последовательный номер в журнале

MCF

Частота использования одного из самых частых значений

MCV

Самое частое значение, одно из значений, которое чаще всего встречается в определённом столбце таблицы

MITM

Man-in-the-middle attack, *Атака посредника*

MSVC

Microsoft Visual C

MVCC

Multi-Version Concurrency Control, Многоверсионное управление конкурентным доступом

NLS

National Language Support, *Поддержка национальных языков*

ODBC

Open Database Connectivity, *Открытое соединение с базами данных*

OID

Object Identifier, Идентификатор объекта

OLAP

Online Analytical Processing, *Интерактивная аналитическая обработка*

OLTP

Online Transaction Processing, *Интерактивная обработка транзакций*

ORDBMS

Object-Relational Database Management System, *Объектно-реляционная система управления базами данных*

PAM

Pluggable Authentication Modules, *Подключаемые модули аутентификации*

PGSQL

Postgres Pro

PGXS

Postgres Pro Extension System, Система расширений Postgres Pro

PID

Process Identifier, *Идентификатор процесса*

PITR

Point-In-Time Recovery, Восстановление на момент времени (Непрерывное архивирование)

PL

Procedural Languages, Процедурные языки (на стороне сервера)

POSIX

Portable Operating System Interface, *Переносимый интерфейс операционных систем*

RDBMS

Relational Database Management System, *Реляционная система управления базами данных*

RFC

Request For Comments, *Рабочее предложение*

SGML

Standard Generalized Markup Language, *Стандартный обобщённый язык разметки*

SNI

Server Name Indication, *Server Name Indication*, *RFC 6066*

SPI

Server Programming Interface, Интерфейс программирования сервера

SP-GiST

Space-Partitioned Generalized Search Tree, Обобщённое дерево поиска с секционированием пространства

SQL

Structured Query Language, *Язык структурированных запросов*

SRF

Set-Returning Function, Функция, возвращающая множество

SSH

Secure Shell, *Защищённая оболочка*

SSL

Secure Sockets Layer, *Уровень защищённых сокетов*

SSPI

Security Support Provider Interface, Интерфейс поставщика поддержки безопасности

SYSV

Unix System V

TCP/IP

Transmission Control Protocol (TCP) / Internet Protocol (IP), *Протокол управления передачей/Межсетевой протокол*

TID

Tuple Identifier, Идентификатор кортежа

TLS

Transport Layer Security, *Протокол защиты транспортного уровня*

TOAST

The Oversized-Attribute Storage Technique, Методика хранения сверхбольших атрибутов

TPC

Transaction Processing Performance Council, Совет по оценке производительности обработки транзакций

URL

Uniform Resource Locator, *Универсальный указатель ресурса*

UTC

Coordinated Universal Time, *Универсальное координированное время*

UTF

Unicode Transformation Format, Формат преобразования Unicode

UTF8

Eight-Bit Unicode Transformation Format, *Восьмибитный формат преобразования Unicode*

UUID

Universally Unique Identifier, Универсальный уникальный идентификатор

WAL

Write-Ahead Log, Журнал предзаписи

XID

Transaction Identifier, Идентификатор транзакции

XML

Extensible Markup Language, *Расширяемый язык разметки*

Приложение O. Глоссарий

Ниже представлен перечень терминов с описанием их значений в контексте PostgreSQL и реляционных баз данных вообще.

ACID	Atomicity (*Атомарность*), Consistency (*Согласованность*), Isolation (*Изолированность*) и Durability (*Надёжность*). Этот набор свойств транзакций в базе данных должен гарантировать корректность операций при параллельном выполнении, а также в случае ошибок, при отключении питания и т. п.
DELETE	SQL-команда, которая удаляет *строки* из данной *таблицы* или *отношения*. Подробности в описании DELETE.
GMT	См. UTC.
GRANT	SQL-команда, дающая *пользователю* или *роли* доступ к определённым объектам в *базе данных*. Подробности в описании GRANT.
INSERT	SQL-команда, добавляющая в *таблицу* новые данные. Подробности в описании INSERT.
LSN	См. Последовательный номер в журнале.
MERGE	Команда SQL для добавления, изменения или удаления *строк* с определёнными условиями в заданной *таблице* с использованием данных из исходного *отношения*. Подробности в описании MERGE.
NULL	Понятие неопределённости, выражаемой как NULL, является ключевым в теории реляционных баз данных. Оно обозначает отсутствие какого-либо определённого значения.
REVOKE	Команда, лишающая указанные *роли* доступа к определённому набору объектов *базы данных*. Подробности в описании REVOKE.
ROLLBACK	Команда, отменяющая все операции, выполненные с начала *транзакции*. Подробности в описании ROLLBACK.
SELECT	SQL-команда, позволяющая запросить данные из *базы*. Обычно команды `SELECT` не должны изменять состояние *базы* никаким образом, однако выполняемые ими запросы могут вызывать *функции*, в качестве побочного эффекта меняющие данные. Подробности в описании SELECT.
SQL-объект	Любой объект, который может быть создан командой `CREATE`. Большинство объектов относятся к одной базе данных, поэтому они обычно называются *локальными объектами*.

	Большинство локальных объектов находятся в определённых *схемах* в соответствующей базе, как например *отношения* (любых видов), *подпрограммы* (любых видов), типы данных и т. д. Имена объектов одного вида в отдельной схеме должны быть уникальными. Кроме того, существуют локальные объекты, находящиеся вне схем, например *расширения*, *приведения типов данных* и *обёртки сторонних данных*. Имена таких объектов одного вида должны быть уникальными в базе данных. Другие виды объектов, а именно *роли*, *табличные пространства*, источники репликации, подписки логической репликации, а также сами базы данных не являются локальными SQL-объектами, так как они существуют вне какой-либо отдельной базы; они называются *глобальными объектами*. Имена таких объектов должны быть уникальными во всём кластере баз данных. Подробности в Разделе 21.1.
TOAST	Механизм, который обеспечивает хранение больших атрибутов строк таблицы, размещая их в дополнительной *TOAST-таблице*. У каждого отношения с большими атрибутами имеется собственная TOAST-таблица. Подробности в Разделе 63.2.
UPDATE	SQL-команда, вносящая изменения в уже существующие *строки* определённой *таблицы*. Эта команда не добавляет и не удаляет строки. Подробности в описании UPDATE.
UTC	Всемирное координированное время (UTC) — основная глобальная система отсчёта времени, близкая к времени нулевого меридиана. Часто ошибочно называют средним временем по Гринвичу (GMT).
WAL	См. Журнал предзаписи.
Автоочистка (autovacuum)	Набор фоновых процессов, регулярно выполняющих операции *очистки* и *анализа*. *Вспомогательный процесс*, который координирует работу и существует всегда (если не отключена автоочистка), называется «autovacuum launcher» (*процесс запуска автоочистки*), а каждый процесс, который запускается для выполнения задач, называется «autovacuum worker» (*рабочий процесс автоочистки*). Подробности в Подразделе 23.1.6.
Агрегатная функция (подпрограмма)	*Функция*, которая сводит к одному значению (*агрегирует*) множество входных значений, например, подсчитывая их количество, складывая их или вычисляя среднее. Подробности в Разделе 9.21. См. также Оконная функция (подпрограмма).
Анализ (операция)	Сбор статистики по данным в *таблицах* и других *отношениях*, помогающий *планировщику запросов* находить оптимальные способы выполнения *запросов*. (Не путайте этот термин с указанием `ANALYZE` команды EXPLAIN.) Подробности в ANALYZE.

Аналитическая функция	См. Оконная функция (подпрограмма).
Атомарность	Свойство *транзакции*, которое заключается в том, что в результате транзакции все её операции выполняются вместе, либо не выполняются совсем. Кроме того, частичные изменения исключены в случае сбоя системы в процессе выполнения транзакции. Это одно из свойств ACID.
Атомарный	Применительно к *данным* означает, что элемент данных нельзя разделить на меньшие части. Применительно к *транзакциям в базах данных* см. *атомарность*.
Атрибут	Элемент с определённым именем и типом данных, содержащийся в *кортеже*.
База данных	Именованный набор *локальных SQL-объектов*. Подробности в Разделе 21.1.
Базовая резервная копия	Двоичная копия всех файлов *кластера базы данных*, которая создаётся программой pg_basebackup. При наличии файлов WAL эту копию можно использовать в качестве отправной точки для восстановления, трансляции журналов или потоковой репликации.
Блокировка	Механизм, позволяющий некоторому процессу ограничивать или запрещать параллельный доступ к ресурсу.
Ведомый (резервный) сервер	См. Реплика.
Ведущий (главный) сервер	См. Ведущий (сервер).
Ведущий (сервер)	Когда несколько *баз данных* связываются вместе посредством *репликации*, *сервер*, служащий первоисточником информации, называется *ведущим* или *главным*.
Владелец кластера	Пользователь операционной системы, которому принадлежит *каталог данных* и от имени которого запускается процесс `postgres`. Необходимо, чтобы этот пользователь существовал до создания нового *кластера базы данных*. В операционных системах с пользователем `root` указанному пользователю запрещается быть владельцем кластера.
Внешний ключ	Вид *ограничения*, определяемый для одного или нескольких *столбцов* в *таблице* и требующий, чтобы значения в этих *столбцах* указывали на ноль или одну *строку* в другой (в редких случаях, той же) *таблице*.
Временная таблица	*Таблица*, существующая либо в рамках *сеанса*, либо в рамках *транзакции*, в соответствии с выбранным при её создании режимом. Такие таблицы не являются *журналируемыми*, и их содержимое не видно в других сеансах. Временные таблицы часто применяются, когда нужно сохранить промежуточные результаты в ходе сложной операции. Подробности в описании CREATE TABLE.
Вспомогательный процесс	Процесс, относящийся к *экземпляру* СУБД и отвечающий за определённую фоновую задачу для этого экземпляра. К числу вспомогательных процессов относятся *процесс запуска автоочистки* (но не рабочие

	процессы автоочистки), *процесс фоновой записи*, *процесс контрольных точек*, *процесс протоколирования*, *процесс запуска*, *процесс архивации WAL*, *процесс-приёмник WAL* (но не *процессы-передатчики WAL*), *процесс создания сводок WAL* и *процесс записи WAL*.
Домен	Пользовательский тип данных, основанный на другом нижележащем типе данных. Он действует так же, как нижележащий тип, за исключением возможного ограничения набора допустимых значений. Подробности в Разделе 8.18.
Журнал предзаписи	Журнал, в котором отслеживаются изменения в *кластере баз данных*, производимые при выполнении операций пользователями и самой системой. Он содержит множество отдельных *записей WAL*, вносимых последовательно в *файлы WAL*.
Журналируемое отношение	*Таблица* считается *журналируемой*, если вносимые в неё изменения проходят через *WAL*. По умолчанию все обычные таблицы являются журналируемыми. Таблицу можно сделать *нежурналируемой* либо во время создания, либо выполнив команду `ALTER TABLE`.
Запись	См. Кортеж.
Запись WAL	Низкоуровневое описание отдельного изменения данных. Оно содержит достаточно информации для воспроизведения (*повторного применения*) этого изменения в случае сбоя системы до переноса изменения в базу. В записях WAL используется непечатаемый двоичный формат. Подробности в Разделе 27.6.
Запись журнала	Устаревшее название *записи WAL*.
Запрос	Указание, передаваемое клиентом *обслуживающему процессу*, в результате выполнения которого меняются данные в базе или выдаются результаты для клиента.
Идентификатор транзакции	Числовой уникальный идентификатор, последовательно назначаемый каждой транзакции при первой операции, меняющей данные. Часто он сокращённо обозначается *xid*. Когда эти идентификаторы хранятся на диске, каждый занимает всего 32 бита, поэтому только около 4 миллиардов пишущих транзакций могут получить уникальные идентификаторы. Чтобы система баз данных могла работать дольше, дополнительно используется *эпоха*, которая также имеет размер 32 бита. Когда счётчик транзакций достигает максимально возможного значения, он сбрасывается до `3` (меньшие значения зарезервированы) и значение эпохи увеличивается на 1. В некоторых контекстах значения эпохи и xid рассматриваются вместе как одна 64-битная величина; за подробностями обратитесь к Разделу 64.1. Подробности в Разделе 8.19.
Изолированность	Свойство, которое заключается в том, что результаты транзакции не видны для *параллельных транзакций*, пока она не будет зафиксирована. Это одно из свойств ACID. Подробности в Разделе 13.2.
Индекс (отношение)	*Отношение*, содержащее производную информацию от исходных данных *таблицы* или *материализованного представления*. Его внутренняя структура ориентирована на быстрый поиск и извлечение исходных данных.

	Подробности в описании CREATE INDEX.
Инкрементальная резервная копия	Специальная *базовая резервная копия*, которая для некоторых файлов может содержать только те страницы, которые изменились с момента создания предыдущей резервной копии, а не полное содержимое каждого файла. Как и базовые резервные копии, она создаётся инструментом pg_basebackup. Для восстановления инкрементальных резервных копий используется инструмент pg_combinebackup, который объединяет инкрементальные резервные копии с базовой. После этого для приведения *кластера базы данных* в согласованное состояние может использоваться *WAL*. Подробности в Подразделе 24.3.3.
Карта видимости (слой)	Хранимая структура, содержащая метаинформацию о каждой странице данных в основном слое таблицы. Для каждой страницы в ней отводится два бита: первый (`all-visible`, полностью видимая) показывает, что все кортежи в странице видны всем транзакциям, а второй (`all-frozen`, полностью замороженная) показывает, что все кортежи в странице помечены как замороженные.
Карта свободного пространства (слой)	Хранимая структура, содержащая метаинформацию о каждой странице данных в основном слое таблицы. В ней указывается, какой объём доступен для новых кортежей в каждой странице, при этом сама она специально оптимизирована для эффективного поиска свободного места для нового кортежа заданного размера. Подробности в Разделе 63.3.
Каталог	В стандарте SQL этот термин обозначает то, что в терминологии PostgreSQL называется *базой данных*. (Не путайте это понятие с понятием *системного каталога*). Подробности в Разделе 21.1.
Каталог данных	Главный каталог *сервера* в файловой системе, содержащий все файлы данных и подкаталоги, связанные с *кластером баз данных* (за исключением *табличных пространств* и, возможно, *WAL*). Обычно на каталог данных ссылается переменная окружения `PGDATA`. Каталог данных в совокупности со всеми дополнительными табличными пространствами образует хранилище данных *кластера*. Подробности в Разделе 63.1.
Класс (устаревшее)	См. Отношение.
Кластер баз данных	Набор баз данных и глобальных SQL-объектов, а также их общих статических и динамических метаданных. Иногда также называется просто *кластером*. Кластер баз данных создаётся с использованием программы initdb. В PostgreSQL термин *кластер* иногда обозначает также экземпляр СУБД. (Не путайте этот термин с SQL-командой `CLUSTER`.) См. также *владелец кластера* — владелец кластера в операционной системе, и *начальный суперпользователь* — владелец кластера PostgreSQL.

Клиент (процесс)	Любой процесс, возможно, удалённый, который *подключается* к *экземпляру* сервера для взаимодействия с *базой данных* и осуществляет *сеанс* использования СУБД.
Ключ	Средство идентификации *строки* в *таблице* или другом *отношении* по значениям, содержащимся в одном или нескольких *атрибутах* этого отношения.
Конкурентный доступ	Концепция, определяющая возможность одновременного выполнения нескольких независимых операций в одной *базе данных*. В PostgreSQL конкурентный доступ реализуется механизмом *многоверсионного управления конкурентным доступом*.
Контрольная точка	Точка в последовательности записей *WAL*, в которой гарантируется, что в файлы данных кучи и индекса попала вся информация из *общей памяти*, изменённая до контрольной точки; для обозначения этой точки в WAL записывается и сбрасывается *запись контрольной точки*. Контрольной точкой также называется процедура, в ходе которой выполняются все действия, необходимые для достижения контрольной точки в приведённом выше определении. Эта процедура запускается при выполнении предопределённых условий, например по истечении заданного времени или после помещения в журнал некоторого объёма записей; также её можно вызвать командой `CHECKPOINT`. Подробности в Разделе 27.5.
Кортеж	Упорядоченный набор *атрибутов*. Порядок атрибутов может определяться *таблицей* (или другим *отношением*), в которой содержится кортеж. В этом случае кортеж часто называют *строкой* таблицы. Он также может определяться структурой результирующего множества; такие кортежи иногда называют *записями*.
Куча	Содержит значения атрибутов *строк* (то есть непосредственно данные) *отношения*. Куча размещается в одном или нескольких *файловых сегментах* в *основном слое* отношения.
Материализованное представление (отношение)	*Отношение*, которое определяется оператором `SELECT` (подобно обычному *представлению*) и при этом содержит данные как обычная *таблица*. Его содержимое нельзя изменить операциями `INSERT`, `UPDATE`, `DELETE` и `MERGE`. Подробности в описании CREATE MATERIALIZED VIEW.
Материализованные данные	Свойство данных, означающее, что они были вычислены заранее и сохранены для последующего использования, а не вычисляются каждый раз «на лету». Это используется в *материализованных представлениях*, в которых данные, полученные из запроса, определяющего представление, сохраняются на диске отдельно от первоисточника данных. Этот термин также используется применительно к некоторым многоэтапным запросам, в которых данные, полученные на некотором этапе, сохраняются в памяти (а могут быть также вытеснены на диск), благодаря чем они могут быть многократно прочитаны на следующем этапе выполнения.
Метод доступа	Интерфейсы, которые Postgres Pro использует для доступа к данным в таблицах и индексах. Этот уровень абстракции позволяет добавлять поддержку новых типов хранения данных.

	Подробности в Главе 59 и Главе 60.
Многоверсионное управление конкурентным доступом (MVCC)	Механизм, позволяющий нескольким *транзакциям* читать и записывать одни и те же строки и при этом не ждать друг друга. В PostgreSQL для реализации MVCC создаются копии (*версии*) *кортежей*, когда их данные меняются; после того, как все транзакции, которым были нужны старые версии, завершаются, эти версии удаляются.
Надёжность	Гарантия того, что после *фиксирования транзакции* произведённые ей изменения не будут потеряны даже в случае системной ошибки или краха сервера. Это одно из свойств ACID.
Начальный суперпользователь	Первый *пользователь*, инициализированный в *кластере баз данных*. Этот пользователь владеет всеми таблицами системного каталога в каждой базе данных. А также это роль, от имени которой автоматически выдаются все разрешения. По вышеуказанным причинам эту роль нельзя удалять. Эта роль также ведёт себя как обычный *суперпользователь баз данных*, и её статус суперпользователя не может быть удалён.
Нежурналируемое отношение	*Отношение*, изменения в котором не отражаются в журнале *WAL*. Для таких отношений отсутствует возможность репликации и восстановления после сбоя. Нежурналируемые таблицы полезны в основном, когда требуется эффективно передать промежуточные рабочие данные между процессами. Все *временные таблицы* являются нежурналируемыми.
Область данных	См. Каталог данных.
Обслуживающий процесс (backend)	Процесс *экземпляра* сервера, заключающий в себе реализацию *сеанса клиента* и обрабатывающий его запросы. (Не путайте это понятие с понятиями *фоновый рабочий процесс* и *фоновый процесс записи*).
Общая память	Область ОЗУ, совместно используемая процессами, относящимися к одному *экземпляру* сервера. В неё отображаются блоки файлов *базы данных*, а также она служит временным хранилищем для *записей WAL* и содержит дополнительную общую информацию. Заметьте, что общая память относится ко всему экземпляру, а не к отдельной базе данных в нём. Самая большая часть общей памяти — *общие буферы*, в которые в виде страниц отображаются блоки файлов данных. Страница, изменённая в буфере, но ещё не сохранённая в файловой системе, называется «грязной». Подробности в Подразделе 18.4.1.
Обёртка сторонних данных	Способ представления данных, находящихся не в локальной *базе данных*, таким образом, что они отображаются как содержащиеся в локальных *таблицах*. Обёртка сторонних данных позволяет определить *сторонний сервер* и *сторонние таблицы*. Подробности в описании CREATE FOREIGN DATA WRAPPER.

Ограничение	Условие, определяющее допустимость значений данных в *таблице* или атрибутов в *домене*. Подробности в Разделе 5.5.
Ограничение уникальности	Вид *ограничения*, определённого для *отношения*, который ограничивает значения, допустимые в одном или нескольких столбцах таким образом, чтобы данное значение или набор значений могли присутствовать в отношении только один раз — то есть ни в какой другой строке отношения не может быть значений, равных данным. Так как *значения NULL* не считаются равными друг другу, ограничение уникальности не нарушается, если в отношении есть несколько строк со значениями NULL.
Ограничение-проверка	Вид *ограничения*, определяемый для *отношения* и ограничивающий множество значений, допустимых в одном или нескольких *атрибутах*. Проверка-ограничение может обращаться к любому атрибуту в той же строке отношения, но не может обращаться к другим строкам этого же отношения или к другим отношениям. Подробности в Разделе 5.5.
Оконная функция (подпрограмма)	Вид *функций*, используемых в *запросах*, который применяется к *секции результирующего множества* запроса; результат такой функции зависит от значений в *строках* одной секции или рамки. В качестве оконных могут использоваться и *агрегатные функции*, но оконные функции способны, например, определять ранг для каждой строки в секции. Также они называются *аналитическими функциями*. Подробности в Разделе 3.5.
Оптимизатор	См. Планировщик запросов.
Отношение	Общий термин, который охватывает все объекты в *базе данных*, имеющие имя и упорядоченный список *атрибутов*. Отношениями являются *таблицы*, *последовательности*, *представления*, *сторонние таблицы*, *материализованные представления*, составные типы и *индексы*. В более общем смысле отношением является набор кортежей; например, результат выполнения запроса тоже отношение. В PostgreSQL название *класс* является устаревшим синонимом *отношения*.
Очистка (VACUUM)	Процесс удаления устаревших *версий кортежей* из таблиц или материализованных представлений и другая тесно связанная с ним обработка данных, потребность в которой продиктована реализацией *MVCC* в PostgreSQL. Очистку можно запустить вручную, воспользовавшись командой `VACUUM`, но она также может осуществляться автоматически процессами *автоочистки*. Подробности в Разделе 23.1.
Параллельное выполнение	Возможность распределять обработку частей *запроса* между несколькими процессами, увеличивая тем самым эффективность использования многоядерных компьютеров.
Первичный ключ	Особый вид *ограничения уникальности*, определяемого для *таблицы* или другого *отношения*, который дополнительно гарантирует, что все

	атрибуты в *первичном ключе* отличны от *null*. Как можно понять из названия, в таблице может быть только один первичный ключ, хотя в ней может быть несколько ограничений уникальности, так же не допускающих значения null в атрибутах.
Переработка	См. Файл WAL.
Планировщик запросов	Компонент PostgreSQL, предназначенный для определения (*планирования*) наиболее эффективного способа выполнения *запросов*. Также называется *оптимизатором запросов*, *оптимизатором* или просто *планировщиком*.
Подпрограмма	Определённый набор инструкций, сохранённый в СУБД, который может выполняться по запросу. Подпрограммы могут быть написаны на самых разных языках программирования. Понятие подпрограмм включает *функции* (в том числе функции, возвращающие множества, и *триггерные функции*), *агрегатные функции* и *процедуры*. PostgreSQL уже содержит большое количество подпрограмм и при этом поддерживает добавление пользовательских подпрограмм.
Пользователь	*Роль* с *правом LOGIN* (см. Раздел 20.2).
Последовательность (отношение)	Вид отношения, позволяющий генерировать значения, чаще всего это последовательные неповторяющиеся числа. Последовательности часто применяются для генерирования идентификаторов в *первичном ключе*.
Последовательный номер в журнале (LSN)	Байтовое смещение в *WAL*, монотонно увеличивающееся с каждой новой *записью WAL*. Подробности в описании `pg_lsn` и Раздел 27.6.
Представление	*Отношение*, которое определяется оператором `SELECT` но само по себе не хранится. При каждом обращении к представлению его определение подставляется в запрос, как если бы пользователь просто вставил это определение в запрос вместо имени представления. Подробности в описании CREATE VIEW.
Приведение	Преобразование *элемента данных* из его текущего типа в другой тип данных. Подробности в CREATE CAST.
Процедура (подпрограмма)	Вид подпрограммы, характеризующийся отсутствием возвращаемых значений. Кроме того, процедуры могут содержать операторы управления транзакциями, например `COMMIT` и `ROLLBACK`. Вызываются процедуры командой `CALL`. Подробности в описании CREATE PROCEDURE.
Процесс postmaster	Самый первый процесс *экземпляра* сервера. Он запускает *вспомогательные процессы* и управляет ими, а также создаёт *обслуживающие процессы* по требованию. Подробности в Разделе 17.3.
Процесс архивации WAL	*Вспомогательный процесс*, который сохраняет копии *файлов WAL* для осуществления резервного копирования или для поддержания актуального состояния *реплик*. Он может быть отключён.

	Подробности в Разделе 24.3.
Процесс записи WAL	*Вспомогательный процесс*, который переносит *записи WAL* из *общей памяти* в *файлы WAL*. Подробности в описании Разделе 18.5.
Процесс запуска	*Вспомогательный процесс*, воспроизводящий WAL во время восстановления после сбоя и в *физической реплике*. (Такое имя объясняется историческими причинами: процесс запуска был назван так до того, как была реализована репликация; имя отражает задачу этого процесса — запуск сервера после сбоя.)
Процесс контрольных точек (checkpointer)	*Вспомогательный процесс*, отвечающий за выполнение процедуры *контрольной точки*.
Процесс протоколирования	*Вспомогательный процесс*, который записывает сообщения о событиях в базе данных в свой текущий *файл журнала*. Данный файл сменяется другим, когда выполняется некоторый критерий (ограничение по времени или объёму). Также этот процесс называется *syslogger*. Он может быть отключён. Подробности в Разделе 18.8.
Процесс создания сводок WAL	*Вспомогательный процесс*, отвечающий за создание сводок данных WAL для *инкрементальных резервных копий*. Подробности в описании Подраздел 18.5.7.
Процесс фоновой записи	*Вспомогательный процесс*, записывающий «грязные» *страницы данных* из *общей памяти* в файловую систему. Он периодически активируется на короткое время, чтобы создаваемая им значительная нагрузка на подсистему ввода-вывода распределялась во времени с целью исключения пиков нагрузки, блокирующих другие процессы. Подробности в Подразделе 18.4.5.
Процесс-передатчик WAL	Специальный *обслуживающий процесс*, передающий WAL по сети. Принимать его может *процесс-приёмник WAL* в *реплике*, pg_receivewal или любая другая клиентская программа, понимающая протокол репликации.
Процесс-приёмник WAL	*Вспомогательный процесс*, который выполняется на *реплике* и получает от *основного сервера* WAL для воспроизведения *процессом запуска*. Подробности в Разделе 25.2.
Раздувание	Увеличение объёма, занимаемого не текущими версиями строк, в том числе объёма неиспользуемого (свободного) места и объёма, занимаемого устаревшими версиями строк.
Расширение	Дополнительный программный модуль, который может быть установлен в *экземпляре* сервера для расширения его функциональности. Подробности в Разделе 36.17.
Результирующее множество	*Отношение*, передаваемое из *обслуживающего процесса клиенту* после выполнения команды SQL, обычно `SELECT`, но также это могут быть команды `INSERT`, `UPDATE`, `DELETE` или `MERGE` с предложением `RETURNING`.

То, что результирующее множество является отношением, означает, что один запрос может фигурировать в определении другого запроса в качестве *подзапроса*.

Реплика	*База данных*, связанная с *ведущей* базой и содержащая копию некоторых или всех данных последней. Прежде всего такие базы создаются для расширения возможностей доступа к данным, а также для обеспечения доступности данных в случае потери *ведущего сервера*.
Репликация	Перенос информации с одного *сервера* на другой называется *репликацией*. Она может реализовываться как *физическая репликация*, когда с одного сервера на другой переносятся все изменения на уровне файлов, или как *логическая репликация*, когда передаются изменения в определённом подмножестве данных, представленные на более высоком уровне.
Роль	Набор прав доступа к объектам *экземпляра* сервера. Принадлежность к роли тоже может считаться правом, которым можно наделять другие роли. Это часто используется, когда нужно дать одинаковые права множеству *пользователей* или организовать распределение прав удобным образом. Подробности в описании CREATE ROLE.
Сеанс	Состояние, в котором клиент может взаимодействовать с обслуживающим процессом, используя установленное *соединение*.
Сегмент	См. Файловый сегмент.
Сегмент WAL	См. Файл WAL.
Секционированная таблица (отношение)	*Отношение*, которое по смыслу не отличается от *таблицы*, но его содержимое хранится распределённо, в нескольких *секциях*.
Секция	Одно из нескольких отдельных (не пересекающихся) подмножеств большего множества. Применительно к *секционированной таблице*: одна из таблиц, содержащих часть данных секционированной таблицы, которая называется *родительской*. Секция сама по себе является таблицей, поэтому запросы могут обращаться к ней напрямую; при этом секция тоже может быть секционированной таблицей, таким образом могут создаваться иерархии. Применительно к *оконным функциям* в *запросе* секция соответствует определённому пользователем критерию, группирующему *строки результирующего множества запроса* для обработки такой функцией.
Сервер	Компьютер, на котором работают *экземпляры* PostgreSQL. *Сервером* может называться как физическая, так и *виртуальная машина* или контейнер. Иногда сервером также называют экземпляр сервера баз данных.
Сервер баз данных	См. Экземпляр СУБД.
Сетевой узел (host)	Компьютер, взаимодействующий с другими компьютерами по сети. Иногда так называют *сервер*. Также это название применимо к компьютеру, на котором работают *клиентские процессы*.

Система накопительной статистики	Система, которая собирает статистическую информацию о действиях *экземпляра* сервера. Она может быть отключена. Подробности в Разделе 26.2.
Системный каталог	Набор *таблиц*, которые описывают структуру всех *SQL-объектов* сервера. Системный каталог располагается в схеме `pg_catalog`. Его таблицы содержат данные во внутреннем представлении и не считаются полезными для конечных пользователей. В более понятном виде часть этой информации доступна в ряде системных *представлений*, также находящихся в схеме `pg_catalog`. Кроме того, имеется схема `information_schema` (см. Главу 35), содержащая дополнительные таблицы и представления, из которых можно получить информацию (частично ту же, но также и некоторую другую), предоставляемую в соответствии с требованиями *стандарта SQL*. Подробности в Разделе 5.10.
Слой	Отдельный набор файлов-сегментов, в которых хранится отношение. В *основном слое* находятся собственно данные отношения. Помимо него существует два дополнительных слоя для метаданных: *карта свободного пространства* и *карта видимости*. У *нежурналируемых отношений* также имеется *слой инициализации*.
Согласованность	Свойство *базы*, суть которого в том, что данные базы всегда удовлетворяют *ограничениям целостности*. Транзакциям позволяется временно нарушать такие ограничения, но если к моменту завершения транзакции эти нарушения не устраняются, транзакция автоматически *откатывается*. Это одно из свойств ACID.
Соединение	Установленное подключение клиентского процесса к *обслуживающему* процессу, обычно сетевое, в рамках которого существует *сеанс*. Иногда этот термин употребляется как синоним сеанса. Подробности в Разделе 18.3.
Соединение (join)	Операция (и ключевое слово SQL), выполняемая в *запросах* для объединения данных из нескольких *отношений*.
Сопоставление пользователей	Устанавливаемое в *обёртке сторонних данных* соответствие учётной записи в локальной *базе данных* другой учётной записи в удалённой системе. Подробности в описании CREATE USER MAPPING.
Ссылочная целостность	Средство ограничения данных в *отношении* по *внешнему ключу* таким образом, чтобы им обязательно соответствовали данные в другом *отношении*.
Стандарт SQL	Совокупность документов, определяющих язык SQL.
Столбец	*Атрибут*, относящийся к *таблице* или *представлению*.
Сторонний сервер	Именованный набор *сторонних таблиц*, для обращения к которым используется одна *обёртка сторонних данных* и общие параметры конфигурации. Подробности в описании CREATE SERVER.
Сторонняя таблица (отношение)	*Отношение*, *строки* и *столбцы* которого представляются содержащимися в обычной *таблице*, но запросы к нему проходят через *обёрт-*

	ку сторонних данных. В ответ эта обёртка возвращает *результирующие множества*, структурированные согласно определению *сторонних таблиц*. Подробности в описании CREATE FOREIGN TABLE.
Страница данных	Основная структура, в которой хранятся данные отношений. Все страницы имеют одинаковый размер. Страницы данных обычно хранятся на диске в определённых файлах и могут быть прочитаны в *общие буферы*, а затем претерпеть изменения, в результате которых они станут *грязными*. При записи на диск они вновь становятся чистыми. Новые страницы, изначально существующие только в памяти, также считаются грязными, пока не будут записаны на диск.
Стратегия доступа к буферу	Некоторые операции обращаются к большому количеству *страниц*. *Стратегия доступа к буферу* помогает предотвратить вытеснение из *общих буферов* слишком большого количества страниц в результате этих операций. Стратегия доступа к буферу устанавливает ссылки на ограниченное количество *общих буферов* и использует их циклически. Когда для операции требуется новая страница, целевой буфер выбирается из буферов в стратегическом цикле, что может потребовать сброса «грязных» данных страницы и, возможно, также несброшенного *WAL* в постоянное хранилище. Стратегии доступа к буферу используются для различных операций, таких как последовательное сканирование больших таблиц, `VACUUM`, `COPY`, `CREATE TABLE AS SELECT`, `ALTER TABLE`, `CREATE DATABASE`, `CREATE INDEX` и `CLUSTER`.
Строка таблицы	См. Кортеж.
Суперпользователь	В данной документации это синоним *суперпользователя базы данных*.
Суперпользователь баз данных	Роль, имеющая *статус суперпользователя* (см. Раздел 20.2). Часто также называется *суперпользователь*.
Схема	Пространство имён *SQL-объектов*, принадлежащих одной *базе данных*. Каждый отдельный SQL-объект должен располагаться ровно в одной схеме. Все системные SQL-объекты располагаются в схеме `pg_catalog`. В более общем смысле *схема* — это совокупность всех описаний данных (определений *таблиц*, *ограничений*, комментариев и т. д.) в определённой *базе* или в её подмножестве. Подробности в Разделе 5.10.
Таблица	Набор *кортежей*, имеющих общую структуру данных (одинаковое количество *атрибутов* в том же порядке, имеющих одинаковые имена и типы). Таблица является наиболее распространённым видом *отношения* в PostgreSQL. Подробности в описании CREATE TABLE.
Табличное пространство	Именованное расположение в файловой системе сервера. Все *SQL-объекты*, для которых требуется хранилище, должны располагаться в некотором табличном пространстве (при этом их определения раз-

	мещаются в *системном каталоге*). Изначально в кластере баз данных есть только одно табличное пространство, которое называется `pg_default` и содержит по умолчанию все SQL-объекты. Подробности в Разделе 21.6.
Точка перезапуска	Вариант *контрольной точки*, выполняемый на *реплике*. Подробности в Разделе 27.5.
Точка сохранения	Специальная отметка в последовательности операций *транзакции*. Изменения данных, произведённые после этой отметки, могут быть отменены, таким образом будет получено состояние на момент сохранения. Подробности в описании SAVEPOINT.
Транзакции/сек (TPS)	Среднее количество транзакций, выполняющихся за секунду, подсчитанное по всем сеансам за время наблюдения. Эта величина позволяет оценить производительность экземпляра сервера.
Транзакция	Совокупность команд, которые должны выполниться как одна *атомарная* команда: они завершаются успешно либо отменяются все вместе, а результат их выполнения не виден в других *сеансах* до завершения транзакции и, возможно, даже позже (в зависимости от уровня изоляции). Подробности в Разделе 13.2.
Триггер	*Функция*, которая вызывается при выполнении определённой операции (`INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`) с некоторым *отношением*. Триггер выполняется в рамках той же *транзакции*, что и вызвавший его оператор, и если в функции триггера произойдёт ошибка, этот оператор тоже не будет выполнен успешно. Подробности в описании CREATE TRIGGER.
Файл WAL	Также называется *сегментом WAL* или *файлом сегмента WAL*. Представляет собой один из последовательно нумеруемых файлов, служащих хранилищем *WAL*. Все эти файлы имеют одинаковый предопределённый размер и записываются по порядку. В файле WAL перемежаются изменения, производимые в нескольких параллельных сеансах. В случае сбоя системы эти файлы также по порядку считываются и все записанные в них изменения воспроизводятся, в результате чего восстанавливается последнее состояние системы до сбоя. Каждый файл WAL может быть освобождён после того, как во время *контрольной точки* все изменения из него будут переписаны в соответствующие файлы данных. Освобождённый файл может быть просто удалён либо переименован (или *переработан*) так, чтобы его можно было использовать в будущем. Подробности в Разделе 27.6.
Файл журнала	Файлы журналов содержат предназначенные для человека текстовые сообщения о событиях. Например, в журнале могут фиксироваться ошибки входа, длительные запросы и т. д. Подробности в Разделе 23.3.
Файловый сегмент	Физический файл, содержащий данные для определённого *отношения*. Размер сегментов ограничивается параметром конфигурации (обычно

	ограничение равно 1 гигабайту), поэтому отношения большего объёма разбиваются на несколько сегментов. Подробности в Разделе 63.1. (Не путайте это понятие с подобным понятием *сегмент WAL*).
Фиксирование	Акт успешного завершения *транзакции* в *базе данных*, обеспечивающий *надёжность* и видимость её результатов для других транзакций. Подробности в COMMIT.
Фоновый рабочий процесс	Процесс внутри *экземпляра* сервера, выполняющий системный или пользовательский код. Обеспечивает инфраструктуру для различной функциональности PostgreSQL, в частности для *логической репликации* и *параллельных запросов*. Кроме того, отдельные рабочие процессы могут создаваться *расширениями*. Подробности в Главе 46.
Функция (подпрограмма)	Вид подпрограммы, который получает ноль или более аргументов, выдаёт ноль или более выходных значений и может выполняться только в рамках одной транзакции. Функции могут вызываться в ходе выполнения запросов, например, `SELECT`. Функции особого рода могут выдавать *множества*; они называются *функциями, возвращающими множества*. Функции также вызываются при срабатывании *триггеров*. Подробности в описании CREATE FUNCTION.
Экземпляр СУБД	Группа *обслуживающих* и *вспомогательных процессов*, использующих общую область разделяемой памяти. Экземпляром СУБД управляет один *процесс postmaster*, и относится к этому экземпляру ровно один *кластер баз данных* со всеми его базами. На одном *сервере* могут работать несколько экземпляров СУБД, если их TCP-порты не конфликтуют. Конкретный экземпляр реализует все функциональные возможности СУБД: читает и записывает файлы, работает с общей памятью, обеспечивает свойства ACID, принимает *подключения клиентских процессов*, проверяет права доступа, выполняет восстановление после сбоя, осуществляет репликацию и т. д.
Элемент данных	Внутреннее представление одного значения некоторого типа данных SQL.
Эпоха	См. Идентификатор транзакции.

Приложение P. Поддержка цветового оформления

Большинство программ из пакета Postgres Pro могут выводить в консоль цветной текст. В этом приложении описано, как это настраивается.

P.1. Когда используется цветной вывод

Использованием цветного вывода управляет переменная окружения `PG_COLOR`, принимающая следующие значения:

1. `always` — цвет используется всегда.
2. `auto` — цвет используется, если поток вывода связан с устройством терминала.
3. При других значениях цвет не используется.

P.2. Настройка цветового оформления

Собственно цветовая палитра настраивается в переменной окружения `PG_COLORS`, в которой задаётся разделённый двоеточиями список пар *`ключ=значение`*. Ключи определяют, для какого именно текста задаётся цвет, а значения устанавливают цвет в формате SGR, который может интерпретировать терминал.

В настоящее время поддерживаются следующие ключи:

`error`

выделяет текст «ошибки» в соответствующих сообщениях

`warning`

выделяет текст «предупреждения»

`note`

выделяет текст «подробностей» и «подсказок» в таких сообщениях

`locus`

выделяет в сообщениях информацию о положении (например, имя программы или имя файла)

Значение по умолчанию: `error=01;31:warning=01;35:locus=01;36:locus=01` (`01;31` = яркий красный, `01;35` = яркий пурпурный, `01;36` = яркий голубой, `01` = цвет по умолчанию, яркий).

> **Подсказка**
>
> Такой формат указания цветов применяется и в других программных продуктах, в частности в GCC, GNU coreutils и GNU grep.

Приложение Q. Устаревшая или переименованная функциональность

Иногда в PostgreSQL меняется функциональность, переименовываются функции, параметры и файлы, или документация перемещается в другое место. Данный раздел помогает читателям документации старых версий, а также пришедшим по внешним ссылкам, найти необходимую им информацию в новом месте.

Q.1. Содержимое файла `recovery.conf` включено в `postgresql.conf`

В Postgres Pro 11 и более ранних версиях существовал файл конфигурации `recovery.conf` , предназначенный для управления репликами и резервными серверами. Начиная с Postgres Pro 12, этот файл не поддерживается. Информация о данном изменении представлена в Замечаниях к выпуску PostgreSQL 12.

В Postgres Pro версии 12 и выше восстановление архивов, потоковая репликация и PITR настраиваются с использованием обычных параметров конфигурации сервера. Соответствующие параметры, как и любые другие, можно задать командой ALTER SYSTEM или внести в файл `postgresql.conf`.

Сервер не запустится, если обнаружит файл `recovery.conf`.

В Postgres Pro версии 15 и ниже был параметр `promote_trigger_file`, а в версиях ниже 12 — `trigger_file`. Теперь вместо них для повышения ведомого сервера используется команда `pg_ctl promote` или функция `pg_promote()`.

Параметр `standby_mode` удалён. Его роль теперь выполняет файл `standby.signal` в каталоге данных. Подробности в Работа резервного сервера.

Q.2. Замена термина «default roles»

В PostgreSQL версии 13 и ниже использовался термин «default roles» (роли по умолчанию). Однако, поскольку эти роли фактически не могут быть изменены и создаются при инициализации системы, более корректно называть их «predefined roles» (предопределённые роли). В текущей документации подробную информацию об этих ролях можно найти в Разделе 20.5, а описание данного изменения — в Замечаниях к выпуску PostgreSQL 14.

Q.3. `pg_xlogdump` переименована в `pg_waldump`

В Postgres Pro 9.6 и более ранних версиях существовала команда `pg_xlogdump` (предназначенная для чтения файлов журнала предзаписи (WAL)). Эта команда была переименована в `pg_waldump`, о ней рассказывается в описании pg_waldump, а информация о данном изменении изложена в Замечаниях к выпуску PostgreSQL 10.

Q.4. `pg_resetxlog` переименована в `pg_resetwal`

В Postgres Pro 9.6 и более ранних версиях существовала команда `pg_resetxlog` (предназначенная для восстановления файлов журнала предзаписи (WAL)). Эта команда была переименована в `pg_resetwal`; о ней рассказывается в описании pg_resetwal, а информация о данном изменении представлена в Замечаниях к выпуску PostgreSQL 10.

Q.5. `pg_receivexlog` переименована в `pg_receivewal`

В Postgres Pro 9.6 и более ранних версиях существовала команда `pg_receivexlog` (предназначенная для получения файлов журнала предзаписи (WAL)). Эта команда была переименована в `pg_receivewal`; о ней рассказывается в описании pg_receivewal, а информация о данном изменении представлена в Замечаниях к выпуску PostgreSQL 10.

Библиография

Здесь представлены ссылки на справочные материалы и книги, посвящённые SQL и PostgreSQL.

Некоторые статьи и технические описания, написанные первой командой разработки POSTGRES, можно найти на *сайте* факультета компьютерных наук Калифорнийского университета в Беркли.

Справочники по языку SQL

[bowman01] *The Practical SQL Handbook*. Using SQL Variants. Fourth Edition. Judith Bowman, Sandra Emerson, Marcy Darnovsky. ISBN 0-201-70309-2. Addison-Wesley Professional. 2001.

[date97] *A Guide to the SQL Standard*. A user's guide to the standard database language SQL. Fourth Edition. C. J. Date Hugh Darwen. ISBN 0-201-96426-0. Addison-Wesley. 1997.

[date04] *An Introduction to Database Systems*. Eighth Edition. C. J. Date. ISBN 0-321-19784-4. Addison-Wesley. 2003.

[elma04] *Fundamentals of Database Systems*. Fourth Edition. Ramez Elmasri Shamkant Navathe. ISBN 0-321-12226-7. Addison-Wesley. 2003.

[melt93] *Understanding the New SQL*. A complete guide. Jim Melton Alan R. Simon. ISBN 1-55860-245-3. Morgan Kaufmann. 1993.

[ull88] *Principles of Database and Knowledge-Base Systems*. Classical Database Systems. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

[sqltr-19075-6] *SQL Technical Report*. Part 6: SQL support for JavaScript Object Notation (JSON). First Edition. 2017.

Документация, посвящённая PostgreSQL

[sim98] *Enhancement of the ANSI SQL Implementation of PostgreSQL*. Stefan Simkovics. Department of Information Systems, Vienna University of Technology. Vienna, Austria. November 29, 1998.

[yu95] *The Postgres95. User Manual*. A. Yu J. Chen. University of California. Berkeley, California. Sept. 5, 1995.

[fong] *The design and implementation of the POSTGRES query optimizer*. Zelaine Fong. University of California, Berkeley, Computer Science Department.

Заметки и статьи

[ports12] «*Serializable Snapshot Isolation in PostgreSQL*». D. Ports K. Grittner. VLDB Conference, August 2012.

[berenson95] «*A Critique of ANSI SQL Isolation Levels*». H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. ACM-SIGMOD Conference on Management of Data, June 1995.

[olson93] *Partial indexing in POSTGRES: research project*. Nels Olson. UCB Engin T7.49.1993 O676. University of California. Berkeley, California. 1993.

[ong90] «A Unified Framework for Version Modeling Using Production Rules in a Database System». L. Ong J. Goh. *ERL Technical Memorandum M90/33*. University of California. Berkeley, California. April, 1990.

[rowe87] «*The POSTGRES data model*». L. Rowe M. Stonebraker. VLDB Conference, Sept. 1987.

[seshadri95] «*Generalized Partial Indexes*». P. Seshadri A. Swami. Eleventh International Conference on Data Engineering, 6–10 March 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, California. 1995. 420–7.

[ston86] «*The design of POSTGRES*». M. Stonebraker L. Rowe. ACM-SIGMOD Conference on Management of Data, May 1986.

[ston87a] «The design of the POSTGRES. rules system». M. Stonebraker, E. Hanson, C. H. Hong. IEEE Conference on Data Engineering, Feb. 1987.

[ston87b] «*The design of the POSTGRES storage system*». M. Stonebraker. VLDB Conference, Sept. 1987.

[ston89] «*A commentary on the POSTGRES rules system*». M. Stonebraker, M. Hearst, S. Potamianos. *SIGMOD Record 18(3)*. Sept. 1989.

[ston89b] «*The case for partial indexes*». M. Stonebraker. *SIGMOD Record 18(4)*. Dec. 1989. 4–11.

[ston90a] «*The implementation of POSTGRES*». M. Stonebraker, L. A. Rowe, M. Hirohama. *Transactions on Knowledge and Data Engineering 2(1)*. IEEE. March 1990.

[ston90b] «*On Rules, Procedures, Caching and Views in Database Systems*». M. Stonebraker, A. Jhingran, J. Goh, S. Potamianos. ACM-SIGMOD Conference on Management of Data, June 1990.

[ston92] « *An overview of the Sequoia 2000 project* ». M. Stonebraker. *Digest of Papers COMPCON Spring 1992*. 1992. 383–388.

Предметный указатель